

A multiagent system for the reliable execution of automatically composed ad-hoc processes*

Walter Binder · Ion Constantinescu · Boi Faltings · Klaus Haller · Can Türker

Published online: 24 February 2006
Springer Science + Business Media, Inc. 2006

Abstract This article presents an architecture to automatically create ad-hoc processes for complex value-added services and to execute them in a reliable way. The uniqueness of ad-hoc processes is to support users not only in standardized situations like traditional workflows do, but also in unique non-recurring situations. Based on user requirements, a service composition engine generates such ad-hoc processes, which integrate individual services in order to provide the desired functionality. Our infrastructure executes ad-hoc processes by transactional agents in a peer-to-peer style. The process execution is thereby performed under transactional guarantees. Moreover, the service composition engine is used to re-plan in the case of execution failures.

*The work presented in this article was supported by the Swiss National Science Foundation as part of the project MAGIC (FNRS-68155), as well as by the Swiss National Funding Agency OFES as part of the European projects KnowledgeWeb (FP6-507482) and DIP (FP6-507483). Authors in alphabetic order.

W. Binder (✉) · I. Constantinescu · B. Faltings
Ecole Polytechnique Fédérale de Lausanne (EPFL),
Artificial Intelligence Laboratory,
CH-1015 Lausanne,
Switzerland
e-mail: walter.binder@epfl.ch

I. Constantinescu
e-mail: ion.constantinescu@epfl.ch

B. Faltings
e-mail: boi.faltings@epfl.ch

C. Türker
Functional Genomics Center Zurich (FGCZ)
UNI/ETH Zurich,
Winterthurerstr. 190,
CH-8057 Zurich,
Switzerland
e-mail: tuerker@fgcz.ethz.ch

Keywords Service composition · Ad-hoc processes · Process execution · Process expansion · Failure handling

1. Introduction

Users benefit from ad-hoc processes and distributed implementations of their execution environment, because for the first time information systems support users not only in standardized situations like workflow systems do. Instead, ad-hoc processes can provide support in a ubiquitous way for everybody, everywhere, and anytime.

An example for an ad-hoc process is planning an evening. The user states his preferences (e.g., comedy movie, restaurant with French cuisine). Then, an ad-hoc process reserves a table in a restaurant and a ticket for a movie. In the scenario depicted in Fig. 1, an ad-hoc process representing a personal agent (PA) tries to find a cinema showing a “good” comedy and a good French restaurant. For this purpose, the PA contacts a movie recommendation service in order to discover a good comedy, as well as a yellow page directory to select a French restaurant. Afterwards, the PA searches for a cinema which plays the chosen movie and uses a recommendation service to ensure that the selected restaurant has a good rating. Finally, the PA returns a restaurant/cinema combination to the user. As it can be seen in this example, this new generation of ad-hoc processes can support users in their everyday life where unique situations appear.

In this way, ad-hoc processes imply a shift in the usage of process technology: Whereas employees in their offices can be assumed to be static, this does not hold for mobile users

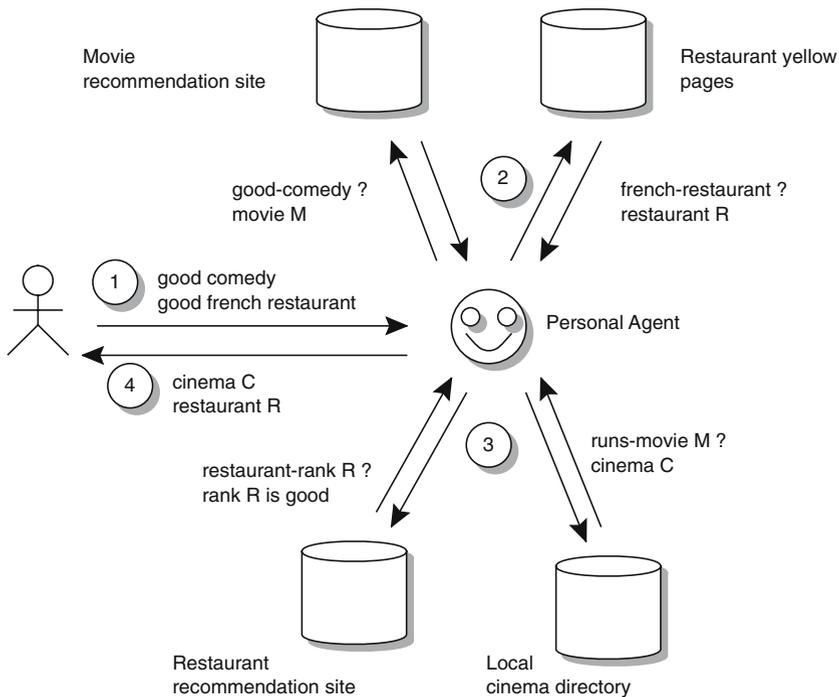


Fig. 1 Exemplary scenario: Planning an evening

who change their location. Consequently, ad-hoc processes respect the nomadicity of users by implementing location-aware services, e.g., time table information systems considering the actual location of the user. Centralized approaches are not suitable for ad-hoc process execution.

Dynamic process changes are required for ubiquitous applications. In case of the evening planner, for example, the user might miss the metro and therefore not reach the cinema in time. Also, the ad-hoc process might figure out that there is no cinema in town. Both require that the ad-hoc process is able to dynamically adapt, for instance, with the help of a decision support system such that nevertheless a valid plan is constructed for the evening.

The main contribution of this article is a novel, integrated infrastructure for the automated creation and reliable execution of ad-hoc processes. The system comprises a service composition planner to generate and dynamically adapt ad-hoc processes, as well as a process engine to ensure the reliable execution of processes with the aid of transactional agents. To the best of our knowledge, we present the first integrated system that takes the fully automated composition of services, which are incrementally retrieved from a directory, and the reliable execution of the resulting plan under transactional control into account.

This article is structured as follows: Section 2 discusses the overall architecture. We assume that individual services are advertised in service directories. Our infrastructure comprises three principal components: Directories that hold the service descriptions, a service composition planner that computes execution plans to fulfill user requirements (such as the aforementioned evening planning task), as well as an engine that executes composition plans in a distributed way using ad-hoc processes. In this article we focus on the service composition planner and on the execution engine. We also discuss their inter-dependencies. Scalable directories that facilitate service composition have been explored in previous work [6]. Section 3 outlines how service capabilities and user requests are represented. In Section 4, we discuss the service composition component of our architecture. Section 5 concentrates on the execution of ad-hoc processes by transactional (mobile) agents. It describes the execution infrastructure with the repositories required on each peer. Furthermore, we explain how we enforce the isolation property [11]. The isolation property guarantees that the system looks stable for each transactional process executing in the system. Finally, Sections 6 and 7 conclude this article.

2. Overview of the system architecture

This section briefly presents the overall architecture of our system and illustrates the interplay of the different components. Figure 2 gives an overview of the architecture. On the left side, there are the peers which are the service providers. For example, there might be peers providing services for reserving tickets for a cinema movie. On the right side, there are the

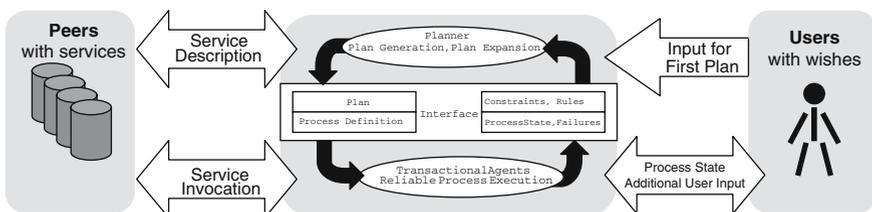


Fig. 2 Architecture

users with their requirements. In between is our planning and execution infrastructure, which fulfills the user requirements by invoking services using ad-hoc processes.

The users inform the planner about their initial goals such that the planner can compose a first plan. This composition is based on the user requirements and on the services advertised by the peers. The result of the composition is a plan, which can also be seen as the definition of a process to be executed in the system. The process definition is compiled into a transactional agent. This agent is responsible for the reliable execution of the process. During the execution, the transactional agent invokes services on peers or presents results to the user and obtains new information from him. However, reliable execution not necessarily implies that the process always performs in the intended way. It can also mean that failures appear which of course must be detected. The process has to roll back until it reaches an expansion step which invokes the planner. Then, the planner modifies the plan. For this task, the planner may access the process state and the failure description, which can also be considered as a constraint for the planner.

3. Service and query descriptions

We represent services and queries in the standard way (W3C. Web services description language (WSDL) version 2.0, <http://www.w3.org/TR/wsd120>) as two sets of parameters (inputs and outputs). A parameter is defined through its name and a type that can be primitive (W3C. XML Schema Part 2: Datatypes, Second Edition <http://www.w3.org/TR/xmlschema-2/>) (e.g., a decimal in a given range) or a class/ontological type (W3C. OWL web ontology languages 1.0 reference, <http://www.w3.org/TR/owl-ref/>) Both primitive and class types are represented as sets of numeric intervals, which enables efficient subsumption testing. For instance, the generic type *Color* may be encoded as the interval [1,3], whereas the specific colors (subtypes) *Red*, *Green*, and *Blue* may be represented as the single-point subintervals [1,1], [2,2], and [3,3]. For more details on the encoding of classes/ontologies as numeric intervals see [7].

Input and output parameters of service descriptions have the following semantics:

- In order to invoke the service, a value must be provided for each of the service input parameters and it has to be consistent with the respective parameter type. For primitive data types, the invocation value must be in the range of allowed values; in the case of classes, the invocation value must be subsumed by the parameter type.
- Upon successful invocation, the service provides a value for each output parameter. Each of these values is consistent with the respective parameter type.

Service composition queries are represented in a similar way but have different semantics:

- The query inputs are the parameters available to the composition (e.g., provided by the user). Each of these input parameters may be either a concrete value of a given type, or just the type information. In the second case, the composition plan has to be able to handle all possible values for the given input parameter type.
- The query outputs are the parameters that a successful composition must provide and the parameter types define what ranges of values can be handled. The composition plan must provide a value for each query output parameter, which has to be in the range of the respective query output parameter type.

For manipulating service or query descriptions, we use the following functions:

- $in(X)$ —returns the set of input parameter names of a service or query description X .
- $out(X)$ —returns the set of output parameter names of a service or query description X .
- $type(P, X)$ —returns the type of a parameter named P in the frame of a service or query description X as the set of intervals of all possible values for P . The \subseteq operator in conjunction with this function represents a range inclusion in the case that P has a primitive data type or subsumption in the case P is defined through a class or concept description (W3C. OWL web ontology language 1.0 reference, <http://www.w3.org/TR/owl-ref/>). The operator \cap in conjunction with this function represents a range intersection in the case that P has a primitive data type or in the case of a class/concept description it represents the subclass common to both arguments of the operator (possibly the bottom class “Nothing”).

We assume that all service and query descriptions X are well-formed in that they cannot have the same parameter both as input and output: $in(X) \cap out(X) = \emptyset$. The rationale behind this assumption is that if a description had an overlap between input and output parameters, this would lead to two undesirable cases: either the two parameters would have the same type, in which case the output parameter is redundant, or they would have different types, in which case the service description is inconsistent.

Parameter names (properties in the case of OWL-S (OWL-S.DAML Services, <http://www.daml.org/services/owl-s/>) or strings in the case of WSDL (W3C. Web services description language (WSDL) version 2.0, <http://www.w3.org/TR/wsdl20/>)) attach also some semantic information to the parameters.¹ Thus, in our composition algorithms we not only consider type compatibility between parameters but also semantic compatibility.

4. Type-compatible service composition

Our approach to service composition is based on the idea of chaining services, starting with the input parameters provided by a query. Forward chaining techniques are used by different types of reasoning systems, in particular for planning [2] and more recently for service composition [8,24].

In Section 4.1 we explain the principle of forward chaining and the kind of type matches that are involved between query and service descriptions. In Section 4.2 we also argue for interleaving service composition with the dynamic discovery of relevant service descriptions from a separate directory. In Sections 4.3 and 4.4 we introduce two service composition algorithms, which are evaluated in Section 4.5.

4.1. Service composition using forward chaining

As described in Section 3, a service composition query is specified in terms of a set of available input parameters and a set of required output parameters. A composition solution is a plan defining the order in which to invoke services and how to pass parameter values between them such that after execution of the plan, values are known for all output parameters required by the query.

Our service composition algorithms are based on forward chaining. Informally, the idea of forward chaining is to iteratively apply a possible service S to a set of input parameters pro-

¹ For WSDL this is not explicitly specified by the standard, but we assume that two parameters with the same name are semantically equivalent.

vided by a query Q (i.e., all inputs required by S have to be available). If applying S does not solve the problem (i.e., still not all the outputs required by the query Q are available), a new query Q' can be computed from Q and S and the whole process is iterated. This part of our framework corresponds to planning techniques currently used for service composition [24].

Now we consider the conditions for a service S to be applied to the inputs available from a query Q using forward chaining: for each input required by the S , there has to be a compatible parameter in the inputs provided by Q . Compatibility has to be achieved both for names (that have to be semantically equivalent) and for types, where the range provided by Q has to be at least as specific as the one accepted by S :

$$(\forall P \in in(S)) (P \in in(Q) \wedge type(P, Q) \subseteq type(P, S))$$

We call this kind of matching between the inputs of query Q and service S *forward chaining with complete type matches*. Concerning inputs, forward chaining with complete type matches corresponds to the *plugin* match identified by Paolucci, Kawamura, Payne, and Sycara [20].

Frequently, forward chaining with complete type matches is too restrictive and fails to find a solution, because the types accepted by the available services may partially overlap the type specified in the query. For example, a query for restaurant recommendation services across all Switzerland may specify that the integer parameter zip code could be in the range [1000,9999], whereas an existing service providing recommendations for the French speaking part of Switzerland accepts only integers in the range [1000–2999] for the zip code parameter. Nonetheless, there may be several recommendation services for different parts of Switzerland that together could cover the whole range given in the query. Hence, a service composition algorithm may create a plan with different execution paths, depending on the concrete value provided for the zip code parameter at execution time. That is, the plan would include a *switch* in order to select the appropriate execution path.

In order to support *partial type matches*, we can relax the type inclusion to a type overlap:

$$(\forall P \in in(S)) (P \in in(Q) \wedge type(P, Q) \cap type(P, S) \neq \emptyset)$$

We call this kind of matching between the inputs of query Q and service S *forward chaining with partial type matches*. It corresponds to the *overlap* resp. *intersection* match identified by Li and Horrocks [18] and Constantinescu and Faltings [7].

Figure 3 illustrates how partially matching services can be aggregated in order to cover a given input range. In this example, the query $Q1$ provides the input parameters X and Y ,

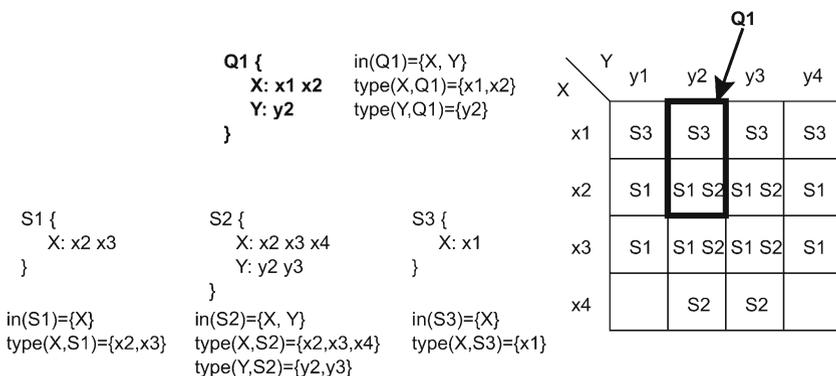


Fig. 3 Composing services with partial type matches

where X may be either x_1 or x_2 , and Y can only be y_2 . Using forward chaining with complete type matches, none of the three service S_1 , S_2 , S_3 are applicable: S_1 and S_2 cannot handle the case $X = x_1$, while S_3 cannot deal with $X = x_2$. However, a combination of S_3 with S_1 or S_2 can handle all possible input parameter values provided by the query Q_1 . Hence, a service composition algorithm based on forward chaining with partial type matches may find a solution in a situation where forward chaining with complete type matches would fail.

4.2. Interleaving service composition with discovery

In contrast to prevailing service composition techniques [24], which require all service descriptions to be loaded into a reasoning engine before starting the composition process, our composition algorithms involve dynamic directory access in order to incrementally retrieve relevant service descriptions. In previous work [6] we presented a directory system that provided specific support for the kind of queries issued by service composition algorithms. Details concerning our directory system are not in the scope of this article, we refer to [6].

The design of our service composition algorithms has been motivated by two aspects that are specific to large-scale service directories operating in open environments:

1. *Large result sets*: For each query, the directory may return a large number of service descriptions.
2. *Costly directory accesses*: Accessing the directory (possibly remotely) can be expensive, as the directory is a shared resource.

Our composition algorithms address these issues by interleaving discovery and composition, as explained in Sections 4.3 and 4.4. We describe our service composition algorithms in Java-based pseudo-code. We assume a `Directory` interface with the method `getNextNewService(Query, boolean)`, which returns a service description matching a given query that has not been returned by a prior invocation of the method during the same run of the composition algorithm. The matching algorithm compares the available inputs of the given query with the required inputs of service descriptions in the directory. If the second argument is false, only complete type matches are taken into account; if it is true, also partial type matches are considered. The matching conditions regarding inputs have been explained in Section 4.1. If there are multiple matching service descriptions in the directory, it returns the service description that provides most of the outputs required by the given query (ranking). `getNextNewService(Query, boolean)` returns null if there are no further matching service descriptions.

4.3. Forward chaining with complete type matches

Figure 4 illustrates a decision algorithm that computes whether a given service composition problem (a query Q) can be solved by forward chaining with complete type matches. The type `Query` represents a query and the type `Service` a service description.

The algorithm uses a search structure `SearchStruct` that keeps track of currently available inputs and of the required outputs. The constructor takes a query Q as argument and initializes the available inputs with $in(Q)$ resp. the required outputs with $out(Q)$. The methods `getAvailableInputs()` resp. `getRequiredOutputs()` returns the currently available inputs resp. the required outputs. The `addService(Service)` method adds the output parameters of a given service description to the available inputs of the search structure, taking name and type of the parameters into consideration. `isSolution()` returns true, if

```

boolean solveFwdComplete(Directory dir, Query Q) {
    return solveFwdComplete(dir, new SearchStruct(Q));
}

boolean solveFwdComplete(Directory dir, SearchStruct ss) {
    Service s = null;
    do {
        if (ss.isSolution()) return true;
        Query dirq = new Query(ss.getAvailableInputs(), ss.getRequiredOutputs());
        s = dir.getNextNewService(dirq, false); // only complete type matches
        if (s != null) ss.addService(s);
    } while (s != null);
    return false;
}

```

Fig. 4 Algorithm: Forward chaining with complete type matches

for each required output there is a matching available input (the type of the required output parameter has to subsume the type of the available input parameter).

The algorithm terminates, because the directory is finite, i.e., at some point `getNextNewService(Query, boolean)` returns null. In our actual implementation, a composition plan is computed from the information stored in the `SearchStruct` instance.

4.4. Forward chaining with partial type matches

Conceptually, our service composition algorithm using forward chaining with partial type matches has three steps (see Fig. 5 for an informal presentation of the algorithm):

1. *Discover complete matches* using the algorithm presented before (see Fig. 4).
2. *Discover full coverage*: Discovery of partially matching services in order to fully cover the available inputs. This step yields a set of sub-problems (represented as queries) to be solved.
3. *Discover correct switch*: Solving of all sub-problems by recursive invocation of the composition algorithm.

A decision algorithm for forward chaining with partial type matches is presented in Fig. 6. In addition to the search structure `SearchStruct` introduced previously, we use a switch generator `SwitchGen`. Upon failure of the composition algorithm supporting only complete type matches (invocation of `solveFwdComplete(Directory, SearchStruct)`), a new `SwitchGen` instance is initialized with the set of available inputs obtained from the failed algorithm.

`SwitchGen` provides two main functionalities corresponding to the second and third steps of the algorithm: Handling the coverage of all possible combinations of available input values and ensuring that for each combination of values, the subsequently composed services provide a solution to the original query. `isFullCoverage()` determines whether the possible input range is fully covered by the services discovered so far. While this is not the case, new services are retrieved from the directory.

When a full coverage is obtained, the `SwitchGen` instance provides a number of sub-problems that all have to be successfully solved in order to get a solution for each possible combination of input values. The sub-problems are represented as an array of `Query` objects returned by `getSubProblems()`. While there is an unsolved sub-problem, additional services are discovered and added to the `SwitchGen` instance by the method `addService()`.

In order to improve algorithm performance, we keep track of successfully solved sub-problems (dynamic programming). This technique avoids re-computing solutions for already

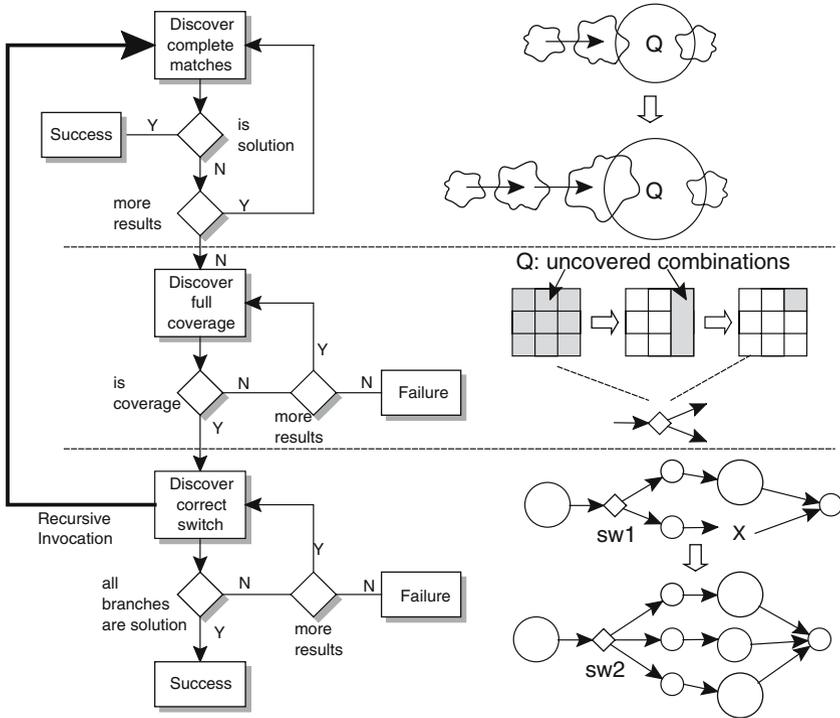


Fig. 5 Flow of algorithm for forward chaining with partial type matches

```

boolean solveFwdPartial(Directory dir, Query Q) {
    return solveFwdPartial(dir, Q, new HashSet());
}

boolean solveFwdPartial(Directory dir, Query q, Set solvable) {
    SearchStruct ss = new SearchStruct(q);
    if (solveFwdComplete(dir, ss)) return true;
    SwitchGen sg = new SwitchGen(ss.getAvailableInputs());
    Query dirq = new Query(ss.getAvailableInputs(), ss.getRequiredOutputs());
    Service s = null;
    do {
        if (sg.isFullCoverage()) {
            boolean isSolvable = true;
            Query[] subProblems = sg.getSubProblems();
            for (int i = 0; i < subProblems.size && isSolvable; i++) {
                if (!solvable.contains(subProblems[i])) {
                    if (solveFwdPartial(dir, subProblems[i], solvable))
                        solvable.add(subProblems[i]);
                    else isSolvable = false;
                }
            }
            if (isSolvable) return true;
        }
        s = dir.getNextNewService(dirq, true); // allow partial type matches
        if (s != null) sg.addService(s);
    } while (s != null);
    return false;
}
    
```

Fig. 6 Algorithm: Forward chaining with partial type matches

solved sub-problems after discovery and addition of a new service description. We use the set `solvable`, which is passed to recursive invocations of `solveFwdPartial(Directory, Query, Set)`, in order to keep track of the sub-problems solved so far.

The algorithm terminates, because the directory is finite and the sub-problems generated by the `SwitchGen` instance have more constrained input ranges. Moreover, the `solvable` set prevents repeated processing of the same solved query. Our actual implementation generates a composition plan from the information stored in the active `SearchStruct` and `SwitchGen` instances.

4.5. Evaluation

We implemented a testbed in order to simulate large-scale service directories as well as service composition problems. The testbed offers several simulation models; for the evaluation presented here, we used the following “media model”:

A user wants to find music providers for an album of good pop music; he is interested only in a complete album. This scenario involves a recommendation site (e.g., `billboard.com`) to find the title of a good album. Next, a library site (e.g., `cdcovers.cc`) is needed to discover the set of melodies for the album. Finally, the melodies are obtained from a provider. In this context, partial type matches allow to select providers that offer only a subset of the melodies of an album. The generated switch combines those providers that together have all the melodies of an album. For solving this kind of composition problems, we need at least a recommendation, an album description, and a provider for the melodies. Hence, a successful service composition involves at least three directory accesses.

With our testbed, we generated random service descriptions and composition problems. For each specific type of service (e.g., album recommendation, album description, melody provider) and for queries (e.g., find good pop album in mp3 format) we had pre-defined sets of parameters. In order to generate a concrete service description or query, we selected a random subtype from a set of possible types for each of the pre-defined parameters.

We solved the queries using the two algorithms presented in Sections 4.3 and 4.4 measuring the average number of directory accesses as well as the failure rate. Each measurement represents 50 runs of our service composition algorithms on randomly generated queries. Figure 7 shows the average number of directory accesses. Both algorithms scale well with

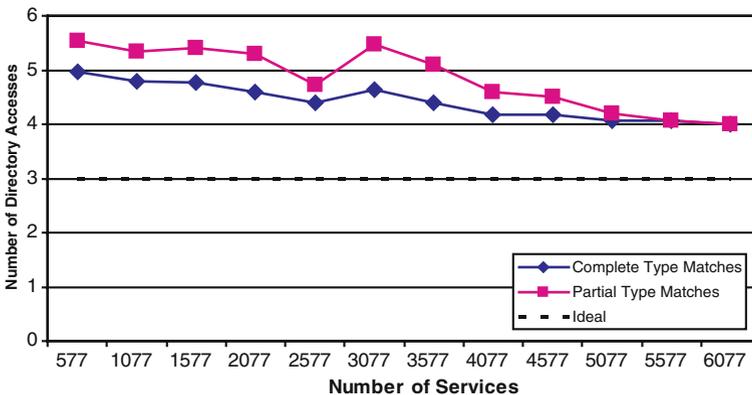


Fig. 7 Media domain: Algorithm performance

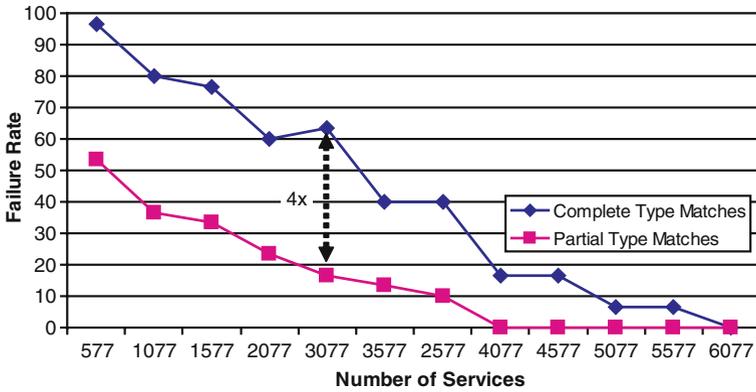


Fig. 8 Media domain: Algorithm failure rate

the size of the directory, as there is no significant increase in the number of directory accesses with increasing directory size.

The algorithm using partial type matches performs comparable with the one supporting only complete type matches. The overhead induced by the support for partial type matches is not significant and decreases as the directory gets saturated with services. This is due to the fact that having more choices makes it easier to solve the coverage problem regarding partial type matches.

The most important result concerns the number of extra problems that can be solved by supporting partial type matches, as illustrated in Fig. 8. The graph shows that the failure rate is up to factor 4 higher, if partial type matches are not considered. This result indicates that the support for partial type matches enables solving many composition problems that cannot be solved by an algorithm supporting only complete type matches. Measurements from other simulation models, which had to be omitted due to space limitations, confirm this result.

5. Process execution

We start the discussion on process execution with an overview of our process execution infrastructure, called AMOR. Section 5.2 then sketches how transactional guarantees are enforced for ad-hoc processes. Thereafter, Section 5.3 discusses how an execution plan is expanded dynamically. Finally, Section 5.4 illustrates an example application.

5.1. Processes and execution infrastructure

In the AMOR prototype [13–15], ad-hoc processes are executed based on transactional (mobile) agents. After the user has specified his requirements and the planner has generated an initial plan, this plan is compiled into a transactional agent. Basically, the planner delivers an XML file which contains the order of the service invocations to be performed. Each service invocation of the plan is mapped to a code fragment which is encapsulated by a Java class *Step*. This class provides a wrapper to integrate the particular service invocation in the AMOR execution infrastructure. Furthermore, the step classes together with the ordering of the steps are packed into a transaction agent class representing this specific plan. Thus, AMOR supports *ad-hoc processes* which satisfy a user need that might not emerge again.

Though the application focus of ad-hoc processes is different from the one of workflows,² their definitions are conceptually the same. Ad-hoc processes come along with transactional guarantees. Thus, we use the term transactional processes throughout this Section equivalent to ad-hoc processes.

Ad-hoc processes are composed of a set of steps. A step can be of one of the following types:

- *Activities* correspond to service invocations in the plan. An activity changes the transactional process state or the state of the overall system outside of a transactional process. We distinguish two types. One type is formed by the *service invocation steps*. They invoke services of one or more peers and may retrieve or manipulate data. For instance, they book a seat in a cinema for a certain movie. *User interaction steps* present data to the users respectively allow users to give information to the ad-hoc process, e.g., which kind of movies the user prefers.
- *Expansion steps* mark situations when the planner cannot define during the planning stage how to continue. Thus, the planning and execution cycle (Fig. 10) starts over again. When the ad-hoc process reaches an expansion step, it stops the execution, hands over the control to the planner, and delivers the actual execution state and failure information over to the planner. The planner generates a plan and then returns the steps to the transactional process, such that the latter one can expand its process description accordingly. Placing user interaction steps strategically before expansion steps, the planner can use them to get information from the user.
- *Control flow* steps define parallel as well as alternative execution paths. Alternative execution paths can be used for implementing expansion steps. Section 5.3 discusses this aspect. Here, we are particularly interested in their possibility to support planners which are able to generate plan variants. If the planner knows that there is user input or influence from the outside world *and* the planner can guess which are the options most likely to be needed, the planner can plan the most probable variants. Additionally, the planner specifies under which conditions which variant is taken. The benefit of plans with alternatives is that the planner is not invoked so often, because each alternative step usually corresponds to an invocation of the planner. Furthermore, when the planner generates a plan it might already gain knowledge about less favorable options. Alternatives allow to materialize this knowledge. Besides the alternative paths, parallel paths can be specified to define paths being executed concurrently. Each path is executed by a replica of the ad-hoc process instance (i.e., the mobile agent). From the specification point of view, a parallel section starts with a *fork* step and ends at a *join* step.

AMOR executes ad-hoc processes in a peer-to-peer fashion without any central instance or component like a workflow engine. Basically, the ad-hoc process (dynamically) decides which service provider on which computer it contacts. In case of a user interaction step, the ad-hoc process figures out on which computer the user is working at the moment. Then, the ad-hoc process instance moves from the peer it currently resides on to the peer the service is provided respectively the user is working on. Internally, AMOR implements ad-hoc processes as mobile agents.

In AMOR, each peer is equipped with a local coordination layer (see Fig. 9). The layer maintains meta-data required by ad-hoc processes to invoke local services. Firstly, the service repository holds information about locally available services. This allows to find concrete service instances implementing the service types specified by the planner. Secondly, the network

² The term workflow is used for environments where few types of workflows are executed very often. Our composition planner generates plans for individual situations. Typically, each plan is executed only once.

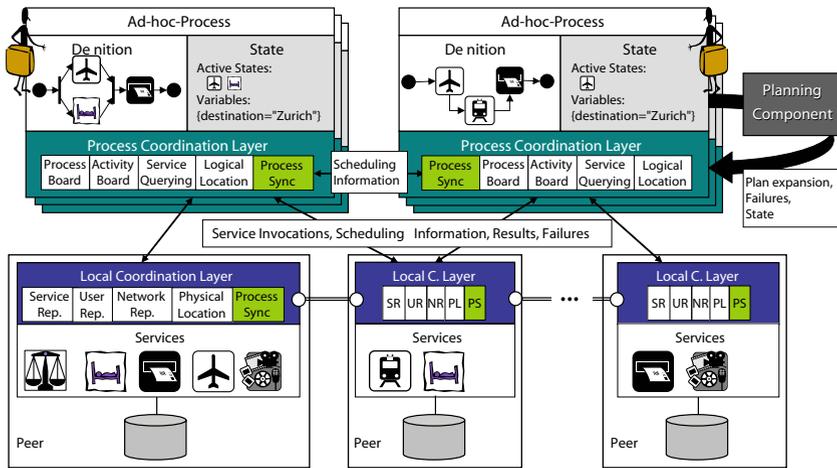


Fig. 9 The AMOR process execution infrastructure

repository manages connections to other peers such that the peers are able to establish ad-hoc communities. The two repositories together allow ad-hoc processes to discover services locally as well as remotely (see [13] for technical details) which is required for executing a service invocation step. If many ad-hoc processes require the same service types, we can speed up the service discovery by letting the peers also locally cache services provided by other peers.

When an ad-hoc process initiates the execution of a service invocation step, its coordination layer contacts the local coordination layer of the peer on which the process currently resides on. The coordination layer returns a reference to the peer providing the determined service. Each process knows the types of services it has to invoke because each process contains its service description. Furthermore, each process manages its execution state in its coordination layer (process coordination layer). Thus, processes can proceed autonomously. In case the process description contains parallel paths, these are executed by replicas of the process. For each path, a replica is created.

The coordination among the replicas is based on an additional component of the process coordination layer named *process board*. It manages variables (objects addressed by their names) of the process. Inter-process synchronization (e.g., for ensuring transactional guarantees which are handled transparently for the user) is performed by cooperation of the *process synchronization components* of the corresponding processes and peers.

5.2. Transactional guarantees for process executions

AMOR supports transactional properties, i.e., guaranteed termination [21] and isolation for process executions which can be understood as a generalization of the classical ACID property [1] known in the context of databases. In the following, we concentrate on the isolation property which prevents undesirable interference between different transactional agents. Thus, the planner does not have to care about this issue (and also complicated failure handling). It can simply assume that its plan will be executed as specified. Otherwise, the system will re-instantiate (by rolling back certain steps) a situation before the interference occurred. From this point, the system can plan over again. The following illustrates the benefit: A

transactional agent retrieves in a step s_r the information that there are exactly two free seats for the desired movie. However, before the agent definitely tries to book these seats, another transactional agent might have done this, too. Conventionally, such failure situations must be detected and dealt with by the planner. Thus, the planner has to check after the booking if it was successfully and has to specify how to deal with problems. In contrast, AMOR compensates the execution before the step s_r and then restarts again. AMOR is able to derive from the process description whether additional steps have to be cancelled cascadingly. This is done by AMOR transparently for the planner.

Technically, AMOR detects such failures by using a serialization graph [1]. The nodes of this graph correspond to the processes of the schedule while the directed edges refer to the conflicts occurred between these processes. The execution (schedule) is correct if the corresponding serialization graph is acyclic.

Traditionally, cycle checking is performed by a central coordinator which maintains a global serialization graph. AMOR's uniqueness is that it does not rely on a global coordinator for checking the acyclicity of the global serialization graph. Bridging this gap between the available, local view of transactional agents and the global knowledge needed to enforce the correctness criterion is therefore an important contribution of our novel protocol. The challenge is to enforce global correctness, although the transactional agents are acting autonomously and thus do not necessarily have up-to-date global knowledge.

In the following, we provide a brief overview of the algorithm which is described in detail in [15]. We equip each transactional agent with a local serialization graph which is kept up-to-date by communication between the transactional agents. However, for reasoning whether or not a transactional agent is allowed to commit, it must rely only on information which is guaranteed to be available at commit time. The incoming edges in the serialization graph are up-to-date, because they are caused by service invocations of the particular transactional agent on a peer and are therefore detected by the peer and returned to the invoking transactional agent. If there is no incoming edge (from an uncommitted process), the corresponding process is allowed at commit time. This property can be checked and enforced autonomously by each process. As performance results presented in [15] show, the communication overhead for graph exchange can be neglected in comparison to the duration of long-running processes.

The *commit processing* of a transactional agent T_c ensures that the peers clean up their logs used for conflict detection and inform the transactional agents which are waiting for the commit of T_c . The knowledge about these commit dependencies is delivered by the peers as a reply to the commit message of T_c . Then, T_c informs the dependent transactional agents about its commit.

Besides, AMOR must ensure that transactional agents, which are involved in a cycle, are able to detect this. Since none of the agents involved in a cyclic waiting situation is allowed to commit, this situation would last forever when not detected explicitly. Note that just knowing from which a transactional agent depends on is not sufficient to detect cycles. The transactional agents have to exchange their knowledge (conflict information).³ A transactional agent must inform all other transactional agents it depends on whenever any change happens in its local serialization graph, e.g., due to some service invocation, compensation, or its commit. An agent receiving such a message updates its local graph using additional information of the received one. In case its local graph has changed, a transactional agent transitively propagates its graph further to the agents it depends on. We refer to [15] for more details on the protocol and the recovery strategy when a cycle has been detected.

³ This cooperation and information exchange does not include semantic knowledge. It is like an information exchange needed to form a self-organizing peer-to-peer infrastructure, just on a higher level.

5.3. Plan expansion

Plan expansion is the glue to put planning and process execution together. We illustrate the interplay and thereby the power of the concept of expansion steps in a small example in Fig. 10. This figure contains an initial process consisting of three activities. The process starts with the user input of his preferences for spending the evening. A step for collecting the necessary information (e.g., about cinemas, restaurants, and public transportation) follows, before the process actually makes reservations in the third step. Additionally, the process contains two expansion steps. If the process reaches an expansion step, it invokes the planner which adds new steps to the process.

The two expansion steps illustrate two possibilities to benefit from expansions steps by late or post modeling [9]:

- *Late modeling* means that the planner knows at planning time that it is not able to define the whole plan in advance because additional information is required respectively there are too many options at the moment. Consequently, the planner places an expansion step after the steps which collect the required information.
- *Post modeling* is the conceptual term used for unexpected situations caused by implicitly made assumptions which turned out to be wrong. One example could be a blackout in the city such that cinemas cannot show any movie and busses run late because there are no traffic lights working. Expansion steps define strategic fallback points from which the planner assumes to be able to re-plan in a sensible way. In this way, expansion steps allow a kind of postponed contingency planning.

Placing an expansion step requires to know the semantics of the process and thus has to be done by the planner. Adding an alternative path with an expansion step (e.g., after “collect information”) after each step is the simplest approach. However, placing many not helpful plan expansion steps implies that in case of a failure the planner is called many times without having a change. For example, in Fig. 10, placing an alternative path with an expansion step after the “user input” step would not be helpful. If the step “collecting information” fails because the databases for cinemas, restaurants, etc. are not available, re-planning would not help (but waste time and resources).

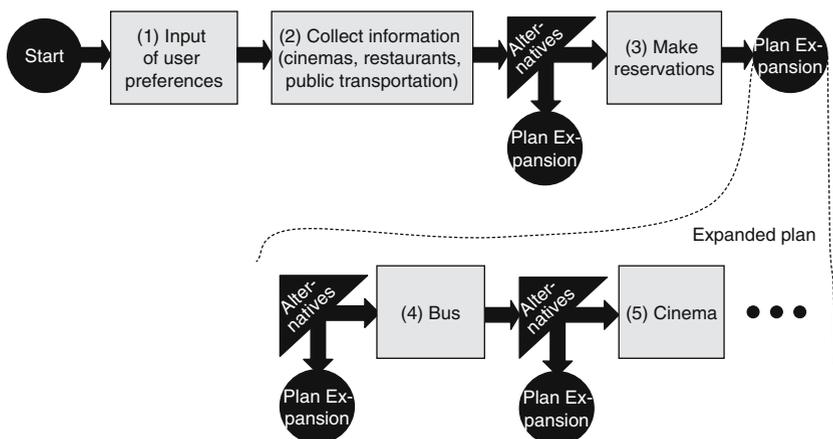


Fig. 10 Example of a process expansion

5.4. Enhanced user support

Research in the area of peer-to-peer computing has become quite fashionable. Whereas one of the first adopters concentrated in finding objects characterized by a single string, we concentrated from the beginning on finding objects (or more precisely services) characterized by a set of attributes [13]. However, we do not concentrate so much on the implementation techniques but in generating added value for users by providing new possibilities. Firstly, we support users by ad-hoc processes with support for ubiquitous applications. We accept users as “. . . nomads, moving between office, home, airplane [. . .] In doing so, we often find ourselves decoupled from our ‘home base’ computing and communication environment” [16]. Furthermore, we accept that a user may log out from one device and log in on a different PC and that he wishes to continue his old session on the new device. Technically, when a process reaches a *user interaction step*, it normally cannot know which device the user currently employs, who is responsible for the next user interaction step. So in our technical solution the process has to figure this out, i.e., it has to discover the particular user in the network.

Therefore, the process sends a query for a service “*Interaction with the user X*”, because each user is modeled as a service “*Interaction with . . .*” on the device he uses at the moment. These services are stored in the user repository of the local coordination layer, which acts as a user manager. When such a user interaction service and thereby the user’s actual location is identified, the ad-hoc process migrates to the corresponding peer and opens a window for the user interaction. Certainly, such windows must be closed if the user logs out. Moreover, they must be transferred to another peer if the user logs in there.

As a second enhanced user support feature, we provide a framework for location-aware respectively context-aware services. They are necessary, because the nomadicity of users also implies that their information needs might depend on their current location. Weiser [26] stated “*If a computer merely knows what room it is in, it can adapt its behavior in significant ways without requiring even a hint of artificial intelligence*”. Realizing this vision requires to determine the user location. We realize this functionality in two steps. Firstly, each local coordination layer provides a location module providing for the *physical* coordination. This module relies on GPS information or in case of desktop PCs it might be implemented as a simple data field with a fixed value. The physical coordination might be sufficient for simple scenarios, but sophisticated applications require logical places like “The nearest train station is the main railway station”. The transformation from the physical to the logical location depends on the user (e.g., “home” is usually at a different location for different persons) and the application (for a railway schedule application, “home” is the nearest station and not the GPS coordinates of the user’s flat). Thus, the mapping from physical to logical coordination is done by the process. Whereas a generic mapping solution is not possible, putting this functionality in a module allows to reuse such a mapping module in different processes and applications.

In our prototype (for details see [12]), we implemented a timetable information service as a simple location-aware service. The current physical location of a user accessing this service is received from the peer information the user is logged in. This information is static for stationary peers like desktop PCs or dynamically determined for mobile devices like a notebook. Therefore, we equipped a notebook with an Haicom HI-202E USB GPS mouse (used together with the Chaeron GPS Library and IBM’s Java Communication Library). For the mapping from the physical to the logical place, we use a table which contains in the first column the logical name of the place and in the following columns points representing a rectangle. The logical name of the place corresponds to a stop of a bus or a train station and

therefore can be used as an input to timetable information service. Certainly, for our evening planner we also have to provide the nearest stations respectively bus stops for all cinemas and restaurants.

6. Related work

Below we review related work in the area of service composition and service execution.

6.1. Service composition

Prevailing approaches to service composition are based on planning techniques that rely either on theorem proving (e.g., Golog [19]) or on hierarchical task planning (e.g., SHOP-2 [27]). These approaches assume that the relevant service descriptions are initially loaded into the reasoning engine and that no discovery is performed during composition. In contrast, our service composition algorithms interact with an external directory in order to dynamically retrieve relevant service descriptions.

While our approach aims at functional-level service composition by selecting and combining services to match given user requirements, process-level service composition [25] is able to handle complex service interactions. These two approaches are complementary: Integrating process-level service composition (after an initial functional-level service composition step) would allow to deal with complex service protocols.

6.2. Service execution

The overall goal of our execution engine is to provide a decentralized implementation for concurrency control and recovery. This is closely related to distributed deadlock detection [17], which is however optimized for short-living transactions. Following an optimistic approach, our system also addresses long-running transactions. As stated before, our serialization graph-based approach does not block processes using locks. Processes are executed in an optimistic manner. If an isolation failure is detected, the process execution is rolled back by compensation. Our analytical as well as experimental results [14,15] show that this strategy even becomes better in comparison to locking-based approaches when the duration of processes increases.

To support atomic applications in large-scale environments, the Transaction Internet Protocol (TIP) has been proposed [10]. Recent extensions, such as Web Service Transactions (WS-Transactions) [3], also support ‘Business Activities’, which are based on an optimistic service execution model with compensation. However, both TIP and WS-Transactions do not consider isolation.

In [5] the authors discuss how multiple (non-mobile) agents can cooperate in order to guarantee atomic (but not necessarily isolated) workflow execution. The idea of mobile agents executing workflows — first published in [4] — is similar to our approach to build transactional agent applications on top of services made available by different resource agents. Other approaches even address the integration of mobile agent technology and transaction management. First results were achieved by enforcing atomicity and fault tolerance based on replication techniques [23]. More recent approaches, such as [22], also provide support for concurrency control. However, all these efforts focus on short-living transactions. Hence, they do not consider isolation and consequently lack support for the reliable execution of ad-hoc processes in large-scale environments.

7. Conclusion

This article presented a novel service composition approach to complex value-added services. Our approach bundles services provided by peers in a peer-to-peer network to ad-hoc processes, which fulfill user needs. Our execution platform runs ad-hoc processes in a completely distributed way with transactional guarantees. If the execution engine cannot resolve a failure situation, it interacts with the service composition engine to find an alternative solution. Here, dynamic process expansion (redefinition) takes place. An important contribution of this article is that it combines service composition techniques (from the A.I. community) with mechanisms for reliable ad-hoc process execution (from the database community) to realize a system, which can support users in their daily live and work everywhere and anytime.

References

- Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency control and recovery in database systems*. Addison-Wesley.
- Blum, A. L., & Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2), 281–300.
- Cabrera, F. et al. (2001). Web services transaction. BEA Systems, IBM, Microsoft.
- Cai, T., Gloor, P., & Nog, S. (1996). Dartflow: A Workflow Management System on the Web using Transportable Agents. Technical Report TR96-283, Dartmouth College, Hanover, NH.
- Chen, Q., & Dayal, U. (2000). Multi-agent cooperative transactions for e-commerce. In *7th Int. Conference on Cooperative Information Systems, Eilat, Israel*.
- Constantinescu, I., Binder, W., & Faltings, B. (2005). Flexible and efficient matchmaking and ranking in service directories. In *2005 IEEE International Conference on Web Services (ICWS-2005)*, Florida, July 2005.
- Constantinescu, I., & Faltings, B. (2003). Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, (pp. 75–81).
- Constantinescu, I., Faltings, B., & Binder, W. (2004). Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, (pp. 506–513), San Diego, CA, USA, July 2004.
- Deiters, W., Goesmann, T., Just-Hahn, K., Loeffeler, T., & Rollers, R. (1998). Support for exception handling through workflow management systems. In *Workshop 'Towards Adaptive Workflow Systems', Conference on Computer-Supported Cooperative Work, Seattle, WA*.
- Evans, J. L. L., & Klein, J. (1998). Transaction Internet Protocol Version 3.0. <http://www.ietf.org/rfc/rfc2371.txt>. IETF RFC 2371.
- Gray, J., & Reuter, A. (1993). *Transaction processing: Concepts and techniques*. Morgan Kaufmann.
- Haller, K., Ackermann, M., Muinari, C., & Türker, C. (2004). Enhanced User Support for Mobile Ad-hoc Processes. In *Proc. of the German Informatics Workshop on Foundations and Applications of Mobile Information Technology, Heidelberg, Germany*, March 2004, (pp. 53–62).
- Haller, K., & Schuldt, H. (2001). Using Predicates for Specifying Targets of Migration and Messages in a Peer-to-Peer Mobile Agent Environment. In *5th International Conference on Mobile Agents (MA)*, Atlanta, GA.
- Haller, K., Schuldt, H., & Schek, H.-J. (2003). Transactional Peer-to-Peer Information Processing: The AMOR Approach. In *4th Int. Conf. on Mobile Data Management, Melbourne, Australia*.
- Haller, K., Schuldt, H., & Türker, C. (2005). Decentralized Coordination of Transactional Processes in Peer-to-Peer Environments. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005)*, Bremen, Germany, Nov. 2005.
- Kleinrock, L. (1995). Nomadic computing (keynote address). In *International Conference on Mobile Computing and Networking, Berkeley, CA*.
- Krivokapic, N., Kemper, A., & Gudes, E. (1999). Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis. *VLDB Journal*, 8(2), 79–100.
- Li, L., & Horrocks, I. (2003). A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*.
- McIlraith, S. A., & Son, T. C. (2002). Adapting Golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. McGuinness, & M.-A. Williams (Eds), *Proceedings of the 8th International*

- Conference on Principles and Knowledge Representation and Reasoning (KR-02)* (pp. 482–496), San Francisco, CA, Apr. 2002. Morgan Kaufmann Publishers.
20. Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. (2002). Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, 2002.
 21. Schuldt, H., Alonso, G., & Schek, H.-J. (1999). Concurrency Control and Recovery in Transactional Process Management. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'99)*, (pp. 316–326).
 22. Sher, R., Aridor, Y., & Etzion, O. (2001). Mobile Transactional Agents. In *21st International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, AZ, 2001.
 23. Silva, A., & Popescu-Zeletin, R. (1998). An Approach for Providing Mobile Agent Fault Tolerance. In *Second Int. Workshop on Mobile Agents (MA)*, Stuttgart, Germany.
 24. Thakkar, S., Knoblock, C. A., Ambite, J. L., & Shahabi, C. (2002). Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, (pp. 1–7). Edmonton, Alberta, Canada, July 2002.
 25. Traverso, P., & Pistore, M. (2004). Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, Vol. 3298 of *Lecture Notes in Computer Science*, (pp. 380–394). Springer.
 26. Weiser, M. (1991). The computer for the 21st century. *Scientific American* 265(3), 66–75.
 27. Wu, D., Parsia, B., Sirin, E., Hendler, J., & Nau, D. (2003). Automating DAML-S web services composition using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*.