A Multiagent System for the Reliable Execution of Automatically Composed Ad-hoc Processes

Walter Binder[1], Ion Constantinescu[1], Boi Faltings[1], Klaus Haller[2], and Can Türker[2]

[1] Artificial Intelligence Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
{walter.binder,ion.constantinescu,boi.faltings}@epfl.ch

[2] Database Research Group
Swiss Federal Institute of Technology Zurich (ETHZ)
{haller,tuerker}@inf.ethz.ch

Abstract. This paper 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 serious execution failures.[3]

Keywords: Ad-hoc Processes, Failure Handling, Process Execution, Service Composition, Process Expansion

1 Introduction

Users benefit from ad-hoc processes and distributed implementations of their execution environment, since for the first time information systems support users not only in standardized situations like workflow systems do. Instead, ad-hoc processes can provide ubiquitous support in a 24/365 way for everybody.

An example for an ad-hoc process is planning an evening. The user states its 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 Figure 1, 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 selected 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 situations are unique and usually not appear in the same way more than once.

[3] The work presented in this paper was supported by the Swiss National Science Foundation as part of the project MAGIC (FNRS-68155).

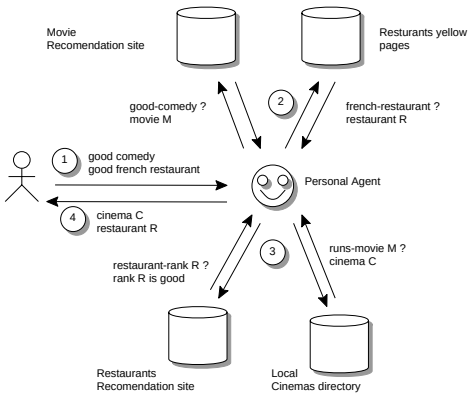

Fig. 1. Exemplary Scenario: Planning an Evening

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 who cannot be assumed to be located at only one place. 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. Also, centralized approaches are not suitable for ad-hoc process execution. Besides, 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 requires that the ad-hoc process is able to dynamically adapt, for instance, with the help of a decision support system such that nevertheless there is a valid plan constructed for the evening.

This paper presents an infrastructure which provides a service composition engine that allows to create and dynamically adapt ad-hoc processes. The underlying process engine ensures the reliable execution of processes by using transactional agents.

Section 2 discusses the overall architecture. We assume that individual services are advertised in service directories. The service composition requires three principal components: Directories that hold the service descriptions, a service composition engine that computes execution plans to fulfill user requirements (such as the evening planning task mentioned before), as well as an engine that executes the composed service in a distributed way using ad-hoc processes. In this paper we focus on the service composition component and on the execution engine. Also, we discuss their inter-dependencies. Scalable directories that facilitate service composition have been explored in previous work [3, 2]. Section 3 outlines how service capabilities and user request are represented. In Section 4, we discuss the service composition component of our architecture. Section 5 concentrates on the execution of the 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 [7]. The isolation property guarantees that the system looks stable for each transactional process executing in the system. This goal is achieved by a distributed implementation of a serialization-graph

testing protocol. Note that locking-based approaches like the S2PL/2PC are not suitable for ad-hoc processes which usually run for a long time. Thus, the service composition engine does not have to deal with problems like that there are seats available in the beginning but not later, because our infrastructure detects and solves them transparently to the user. Finally, Section 6 concludes this paper.

2 Sketch of the System Architecture

This section very briefly presents the overall architecture of our work and illustrates the interplay of the different components. Figure 2 contains peers on the left side, 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 users with their requirements. In between is our planning and execution infrastructure, which fulfills the user requirements by invoking services within the context of ad-hoc processes.

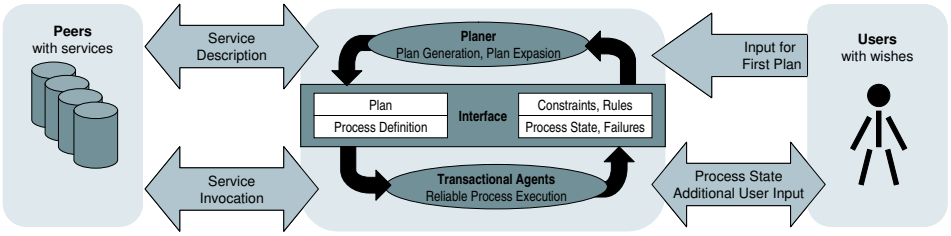

Fig. 2. Architecture

The users have to 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 provided and described 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 can access the process state and the failure description, which can also be considered as a constraint for the planner.

3 Service Descriptions and Requests

We represent service advertisements and service requests through parameters and states of the world [19, 5]. Parameters can be either *input* or *output*, and states of the world can be either *preconditions* (required states) or *effects* (generated by the execution of

the service). We presume that terms in the service descriptions are defined using a class/ontological language like OWL [18]. Primitive data-types can be defined using a language like XSD [20].[4] As specified by the latest version of OWL-S [5], in our formalism each parameter has two elements:

- A *role* describing the actual semantics of the parameter (e.g., in a travel domain the role of a parameter could be $departure$ or $arrival$).
- A *type* defining the actual datatype of the parameter (e.g., the datatype for both $departure$ and $arrival$ could be $location$).

We define states of the world through preconditions and effects. We extend the normal semantics of concepts that can be included in preconditions or effects such that services can achieve behaviour equivalent to standard STRIPS planning operators which in turn allows us to model compensable transactions.

A major difference between the approach used by STRIPS and ours is that traditional planners are based on a *closed world assumption* which is not applicable in our case since by definition web services are entities acting without a central authority. Consequently, we cannot use exactly the same formalism and semantics as for STRIPS. For example, a proposition might not be satisfiable for an agent due to its lack of global knowledge. Hence, in our approach we only use explicit states of the world where negation has also to be explicit. Formally, we do not use delete lists or terms with negation but we rely on the definition of *inverse* relations between terms (antonyms).

We use *inverse* preconditions or effects for modelling semantically compensable services. Given two services S_1 and S_2 with effects e_1 and e_2, where $e_2 \equiv \neg e_1$, we can infer that S_2 can be used to semantically compensante S_1, since applying S_2 will actually undo the effects of S_1. E.g., in a banking system the states $credit$ and $debit$ could be defined such that $credit(x) \equiv \neg debit(x)$. Thus, a service that has as effect $debit(1000)$ can be semantically compensated by calling another service with the effect $credit(1000)$.

In service advertisements input and output parameters, as well as preconditions and effects, have the following semantics:

- In order for the service to be invokable, a value must be known for each of the service input parameters and it has to be consistent with the respective semantic role and syntactic type of the parameter. The parameter provided as input has to be semantically more specific than what the service is able to accept. Regarding the parameter type, in the case of primitive data types the invocation value must be in the range of allowed values, or in the case of classes the invocation value must be subsumed by the parameter type.
- Upon successful invocation the service will provide a value for each of the output parameters and each of these values will be consistent with the respective parameter role and datatype.
- The preconditions define in which state the world has to be before the service can be invoked. All preconditions must be entailed by the conditions specified by the current state of the world.

[4] At the implementation level both primitive datatypes and classes are represented as sets of numeric intervals [3].

– After invocation the state of the world will be modified such that all effects listed in the service advertisement will be added to the new world state. Terms in the original state conflicting with terms in the new state (e.g., terms that are in an *inverse* relation) will be removed from the new state.

Service requests are represented in a similar manner but have different semantics:

– The service request inputs represent available parameters (e.g., provided by the user or by another service). Each of these input parameters has attached a semantic role description and either some description of its datatype or a concrete value.
– The service request outputs represent parameters that a compatible (composed) service must provide. The parameter role defines the actual semantics of the required information and the parameter type defines what ranges of values can be handled by the requestor. The compatible (composed) service must be able to provide a value for each of the parameters in the output of the service request, semantically more specific than the requested role, and having values in the range defined by the requested parameter type.
– Preconditions in a request represent the state of the world available for any matching service advertisement. They are equivalent to initial conditions in a classic planning environment. This state has to entail the state required in the precondition of any compatible service.
– Effects represent the change of the world desired by the requestor of the service or the goals that the service request needs to be fulfilled. In order for any of the goals or effects of the service request to be considered fulfilled, the state of the world after the invocation of a given service will have to contain an effect entailing the respective goal.

4 Service Composition with Partial Type Matches

The composition algorithms presented here take as input a request, consisting of a set of provided input parameters, a set of required output parameters, initial preconditions, and desired effects. The composition algorithm interacts with a service directory in order to retrieve relevant service descriptions. If a given service composition problem can be solved, the algorithm returns the workflow of a composed service.

Frequently, provided outputs and required inputs don't have complete compatibility because the type of a required service input may not cover the full range of possible request inputs. However, there may be multiple services that together cover the full range. To address this, we have designed a service composition algorithm based on forward chaining, which we call *forward chaining with partial type matches*. In this algorithm we do not require the range of a service input parameter to cover the full range of an available request input parameter. For a service to be considered, it is sufficient that for each required service input there is a parameter with matching roles and overlapping type provided by the request. We still require that all preconditions of s are fulfilled by r. This kind of matching between the types of the inputs of request r and of service s corresponds to the *overlap* or *intersection* match [15, 3].

Service composition with partial type matches combines several partially matching services into a composite service using a *switch* that maps each possible combination

of parameter values from the space of available request inputs to one or more partially matching services. To be able to fulfill the same functionality as a completely matching service, there has to be at least one service for each range combination of input parameters that accepts these parameter values. The switch corresponds to a non-deterministic planning operator: The choice point that it introduces allows for a number of possible service invocation paths to be followed without commitment to a particular one during service composition. The choice will be made at runtime during the executing of a service composition plan based on the input parameter values of the switch. Each of the branches in a switch provides a (possibly different) set of available parameters.

Experimental results carried out in various domains show that using partial matches decreases the failure rate by up to 7 times compared with an integration algorithm that supports only complete matches [4].

5 Process Execution

5.1 Processes and Execution Infrastructure

After the user has specified its requirements and the planner has generated an initial plan, this plan is compiled into a transactional agent. Transactional agents are used in the AMOR prototype [8, 9, 11, 13, 12, 17] to execute *ad-hoc processes*. Ad-hoc processes satisfy a user need which might not emerge again. The definition of an ad-hoc process is quite similar to that of workflows. Ad-hoc processes are composed of a set of steps. These steps can be of different types:

- *Activities* are either service invocation or user interaction steps. These steps contact and/or influence the outside world. *Service invocation steps* invoke services on one or more peers, which may retrieve or manipulate data. An example is the reservation of a seat for a movie. *User interaction steps* present data to the users or allow them to give information to the process. For instance, the user can state that he does not want to see action movies.
- *Control flow steps* allow to define parallel as well as alternative execution paths. Parallel execution paths are specified by using *fork* and *join* steps.
- *Expansion steps* define situations in the process in which the execution state and failure information is delivered to the planner which in return provides the ad-hoc-process with steps by which the process is extended with. Placing user interaction steps strategically before expansion steps, the planner can use them to get information from the user.

The AMOR prototype executes ad-hoc processes in a peer-to-peer fashion. An ad-hoc process instance moves from peer to peer to invoke the specified services on the corresponding peers. Ad-hoc processes are internally implemented as mobile agents. Without relying on a centralized coordination engine, AMOR ensures globally correct execution. To realize this, each peer is equipped with a local coordination layer, as depicted in Figure 3. This 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 repository manages connections to other peers.

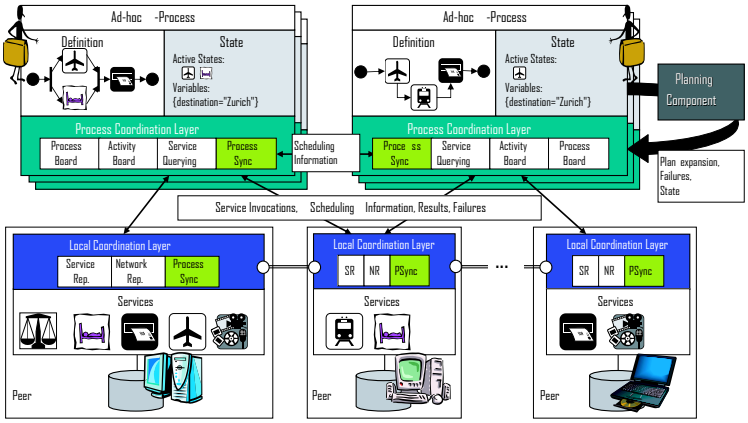

Fig. 3. The AMOR Process Execution Infrastructure

Thereby, peers are able to establish ad-hoc communities. These two repositories allow ad-hoc processes to discover services locally as well as remotely (see [9] for details). To speed-up the service discovery in case of similar service queries, information regarding services on other peers could also be cached locally.

Whenever an ad-hoc process intends to execute a service invocation step, its coordination layer contacts the local coordination layer of the peer on which the process currently resides on. The latter returns a reference to an appropriate peer on which the service can be executed. So, each ad-hoc process must have a process description to know its application semantics. Also, each process manages its execution state such that the coordination layer of the process can control the current process execution context without requiring a centralized engine. For exchanging information within the process, the process coordination layer includes additional components, e.g., a *process board*. The latter manages variables, i.e., objects addressed by their names, of the process. Information flows between different (sequential or parallel) steps of a process via this process board. Inter-process synchronization 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 [16] and isolation [1] for process executions. In the following, we concentrate on the isolation property which prevents undesirable interference between different transactional agents. The planner need therefore not care about this issue (and also complicated failure handling). It can simply assume that its plan will be executed as it is specified. Otherwise, the system will re-instantiate a situation from which it can plan over again. The following situation 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 situation must be detected and dealt with by the planner. AMOR compen-

sates the execution before the step s_r and then restarts it again. This time, the seats are not available and another option of the plan is chosen according to the process definition. Of course, this is transparent to the planner which need not to be contacted.

Technically, AMOR detects such failures by using a serialization graph [1]. The nodes of this graph correspond to processes of the schedule while the directed edges refers 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. In our scenario, however, there is no such global coordinator. An important aspect of our novel protocol is to bridge the gap between the available, local view of transactional agents and the global knowledge needed to enforce the correctness criterion. 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.

Therefore, 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.

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

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 can 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). A transactional agent must inform all other transactional agents it depends on whenever any changes 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. For more details about the protocol and the recovery strategy when a cycle has been detected we refer to [10, 13].

5.3 Plan Expansion

In the following, we will illustrate the interplay between the planner and the execution component and thereby the power of the concept of expansion steps in a small example in Figure 4.

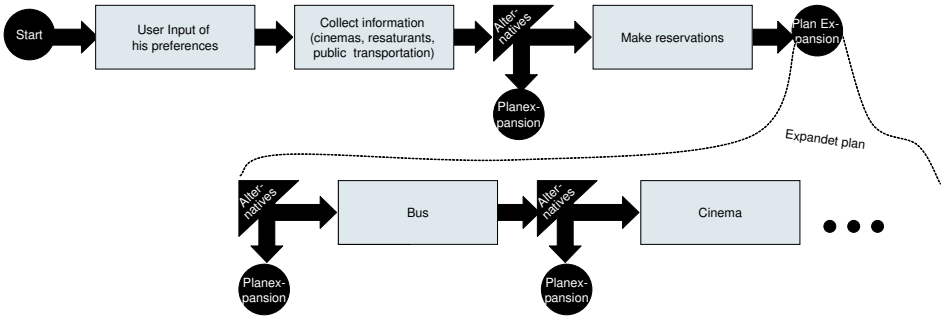

Fig. 4. Example for a Process Expansion

The initial process consists of three activities, beginning with the user input of its preferences how to spend the evening, followed by collecting the necessary information (e.g., about cinemas, restaurants, and public transportation), before the process actually performs the reservations. As we can see, the process also contains two expansion steps. If the process reaches an expansion step, it invokes the planner. The two expansion steps illustrates two possibilities to use them by late or post modelling [6].

Firstly, a plan expansion can be the last step of a regular path because it is not known how to proceed (late modelling). An example for late modelling is the expansion step at the end of the process. Here, the planner cannot know at the time of the process definition exactly what is happening afterwards (e.g., whether a bus or a cab is taken or the user wants to go to the cinema or to the restaurant). Thus, the process is executed until that point, then the planner has enough knowledge to construct the next steps.

Secondly, expansion steps allow to cope with unexpected failures (post modelling) or failures which the planner seems not likely enough to plan for such events. The expansion step between the second and third activity is an example for such a situation: In case the reservation fails, which the planner assumes not to happen likely, the process would step back from the step "reservation" and take the alternative ending at an expansion step. Now, the planner could use the knowledge collected in all steps executed before to generate a new plan, e.g., by booking a theater ticket instead of reserving seats in cinema.

5.4 Enhanced User Support

To support ad-hoc-processes for users in a 24/365 style requires to 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*" [14]. A user can log out and log in from a different PC and can continue the same session on another device. Due to the peer-to-peer process execution, it is unclear for the user where the process currently executes when he logs in again. This requires that either the processes discover the user or vice versa. The AMOR prototype follows the first approach.

If the process reaches an *user interaction step*, it sends a query for a service 'Interaction with the user X'. If such a user 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. Also they must be transferred to another peer if the user logs in there.

Nomadicity of users also implies that his information needs might depend on its current location. Weiser [21] 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*". This property is often known as location-awareness or context-awareness. Realizing this vision requires to determine the user location. In simple scenarios, it might be sufficient to determine only the physical coordinates, but sophisticated applications require logical places like "The nearest train station is the main railway station", which have to be derived from the physical location. Obviously, this is a hard problem which cannot be solved in a generic way.

Thus, the AMOR prototype focused on a sample of a location-aware timetable information service. The current logical location of a user using this service is received from the peer information the user is logged in. This information can either be static in case of stationary peers like desktop PCs or dynamic depending on the current physical location determined. We use the latter one to implement a location-aware timetable information system. For that, 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), on which our sample process of an evening planner gets the location as one input and checks for suitable connections to reach, for instance, a cinema or a restaurant.

6 Conclusion

This paper 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 the ad-hoc processes in a completely distributed way with transactional guarantees. In case the execution engine itself 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 paper 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 every day live 24 hours, 365 days.

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
2. Walter Binder, Ion Constantinescu, and Boi Faltings. A directory for web service integration supporting custom query pruning and ranking. In *European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, September 2004.
3. Ion Constantinescu, Walter Binder, and Boi Faltings. An extensible directory enabling efficient semantic web service integration. In *3rd International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, November 2004.
4. Ion Constantinescu, Boi Faltings, and Walter Binder. Type-based composition of information services in large scale environments. In *The 2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)*, Beijing, China, September 2004.

5. DAML-S. DAML Services, http://www.daml.org/services.
6. W. Deiters, Th. Goesmann, K. Just-Hahn, Th. Loeffeler, and R. Rollers. Support for exception handling through workflow management systems. In *Workshop 'Towards Adaptive Workflow Systems', Conference on Computer-Supported Cooperative Work, Seattle, WA*, 1998.
7. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
8. K. Haller, M. Ackermann, C. Munari, and C. Türker. Enhanced User Support for Mobile Adhoc Processes. In H. Höpfner and G. Saake, editor, *Beitragsband zum Workshop "Grundlagen und Anwendungen mobiler Informationstechnologie", Heidelberg, 23.-24.03.2004*. Fakultät für Informatik, Universität Magdeburg, 2004.
9. K. Haller and H. Schuldt. Using Predicates for Specifying Targets of Migration and Messages in a Peer-to-Peer Mobile Agent Environment. In *5th Int'l Conf. on Mobile Agents (MA), Atlanta, GA*, 2001.
10. K. Haller and H. Schuldt. Towards a Decentralized Implementation of Transaction Management. In *Workshop Grundlagen von Datenbanken, Tangerm"unde, Germany*, 2003.
11. K. Haller, H. Schuldt, and H.-J. Schek. Transactional Peer-to-Peer Information Processing: The AMOR Approach. In *4th Int. Conf. on Mobile Data Management, Melbourne, Australia*, 2003.
12. K. Haller, H. Schuldt, and C. Türker. Flexible Fehlerbehandlung für mobile Ad-hoc-Prozesse. In *Persistence, Scalability, Transactions — Database Mechanisms for Mobile Applications, Proc. Workshop by the GI-Arbeitskreis "Mobile Datenbanken und Informationssysteme*, Lecture Notes in Informatics P-43, pages 55–68. Gesellschaft für Informatik, 2003.
13. K. Haller, H. Schuldt. Consistent Process Execution in Peer-to-Peer Information Systems. In *Proc. 15th Int. Conf. on Advanced Information Systems Engineering*, 2003.
14. L. Kleinrock. Nomadic computing (keynote address). In *International Conference on Mobile Computing and Networking, Berkeley, CA*, 1995.
15. Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, 2003.
16. H. Schuldt, G. Alonso, and H.-J. Schek. Concurrency Control and Recovery in Transactional Process Management. In *Proc. of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'99, May 31–June 2, 1999, Philadelphia, Pennsylvania*, pages 316–326. ACM Press, 1999.
17. C. Türker, K. Haller, H. Schuldt, and H.-J. Schek. Mobilitätsaspekte in Informationsräumen. *Datenbank-Spektrum*, 5:16–23, 2003.
18. W3C. OWL web ontology language 1.0 reference, http://www.w3.org/tr/owl-ref/.
19. W3C. Web services description language (wsdl) version 1.2, http://www.w3.org/tr/wsdl12.
20. W3C. XML Schema Part 2: Datatypes, http://www.w3.org/tr/xmlschema-2/.
21. M. Weiser. The computer for the 21st century. *Scientific American 265(3): 66-75*, 1991.