RELEVANCE-BASED CONTEXT SHARING THROUGH INTERACTION PATTERNS

(Invited paper)

Robert Gombotz, Daniel Schall, Christoph Dorn, Schahram Dustdar
Distributed Systems Group, Institute of Information Systems
Vienna University of Technology
Argentinierstrasse 8, 1040 Wien, Austria
{gombotz, schall, dorn, dustdar}@infosys.tuwien.ac.at

***Abstract*-In collaborative working environments (CWE), human interaction patterns represent reoccurring situations describing the sequence and type of interactions between individuals. We believe that such patterns provide information that may be used to improve human collaboration. In this paper we introduce interaction patterns to an existing context sharing platform used by distributed teams. We use these patterns to formulate rules that help determining the relevance of context information between users and that raise team awareness between interacting entities. These rules are integrated in an existing platform for context sharing between mobile users which allows us to demonstrate the practical applicability of our approach.**

I. INTRODUCTION

Optimized execution of business processes is a necessity for every organization in today's competitive market. Enterprises need to define and engineer, automate and implement processes in order to optimize intra-organizational communication. The management of such processes includes monitoring of execution parameters such as throughput, average execution time, etc. and also deviation detection including bottlenecks, deadlocks, etc. based on process models. Ad-hoc processes pose additional challenges as they emerge and change in a highly dynamic fashion. A detailed process description or schema may not be available at design time as interactions and communication among humans develop at run-time.

Our paper deals with context-aware interaction in such ad-hoc collaboration environments. Predefined process models and assignment of roles (i.e., team roles in collaborating teams) in a static way are not suitable in highly dynamic ad-hoc environments. We describe an initial set of interaction patterns, which can be found in the Software Engineering domain and show how these patterns can be used to make collaboration and communication context-aware, particularly focusing on teams.

A. Context in Collaborative Environments

Collaboration in distributed teams of knowledge workers becomes increasingly important as we rely on others to contribute their expertise to make decisions. Awareness in teams is necessary to achieve goals or process tasks and subtasks, which may be scattered in form of activities among team members, in a timely fashion. Consider a workflow such as "organize conference". This workflow comprises *actors*, e.g., conference chair, organizer, etc. and a set of activities including "schedule meeting" and "check availability", which are executed in form of tasks by the system. *Awareness* among team members is achieved through exchange of context information. However, the level of information needed depends on the type of interaction between team members. For example, a team leader or coordinator wants to be updated regarding the status of a specific activity, team members want to share their availability/location status.

In our previous work [4], we have introduced context hierarchies as means for structuring and processing context information. Information is organized according to levels of detail where coarse-grained information is stored at the top and fine-grained information saved at the bottom of each hierarchy. This mechanism reduces context data that needs to be exchanged as only relevant information is transferred across the network. In our Context Sharing and Subscription Framework [6] we have implemented a number of hierarchies including *Location*, *Activity*, *Reachability*, *Availability*, and *Device Status*.

For example, a *location hierarchy* essentially emulates a tree structure with a set of linked nodes. Child nodes represent fine grained location information and links between nodes denote spatial relationships and proximity (e.g., a set of rooms at the same floor level). A child's parent is a superset of a particular spatial region. Quite intuitively, a floor level in a building comprises a set of rooms (child nodes); the floor's parent level is the building and so on. In addition, each level in a location hierarchy holds confidence intervals of obtained location information (e.g., how likely is the user located within a spatial region). The current framework allows subscriptions to be made at hierarchy levels. When multiple context hierarchies are used in combination, users can issue queries, e.g., select users having given communication capabilities and located within given region", or make subscriptions such as tell me when given user is available.

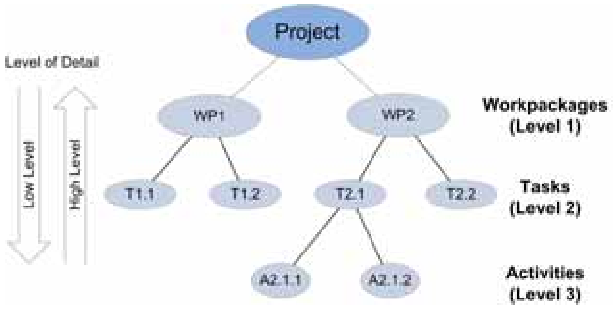

Figure 1. Project-Activity Hierarchy

Fig. 1 illustrates a context hierarchy as a tree structure. Moving up in the tree structure decreases the level of detail of context information. A lower level in the tree means higher level of abstraction. In the following, *higher level of context information* refers to levels in the context hierarchy, i.e., a higher level of abstraction, or, by definition, a lower level of detail.

II. INTERACTION PATTERNS IN CWE

The term *interaction pattern* refers to a common, reoccurring interaction scenario between actors. The term *relation* refers to a tie or link between 2 actors within a pattern. In our context sharing platform, we take three initial interaction patterns that are well known in the domain of Software Engineering (SE) and apply them to the domain of human collaboration.

A. Proxy Pattern

Originally, the Proxy pattern was introduced by Gamma et al. as a structural pattern in software design. The intention for using a proxy is "to provide a surrogate or placeholder for another object to control access to it [2]."

Besides forwarding the clients' requests and sending back the response, a proxy can do pre- or post-processing depending on its type (e.g., protection proxy, cache proxy). A real-life example of a proxy in human collaboration is a secretary. He or she receives emails, phone calls, messages, etc. which are actually intended for a different entity, i.e., the boss. The secretary pre-processes these client requests by, for example, filtering out unwanted requests (protection proxy) or even answering simple requests without having to involve the boss (cache proxy) [1]. A proxy pattern is depicted in Fig. 2.

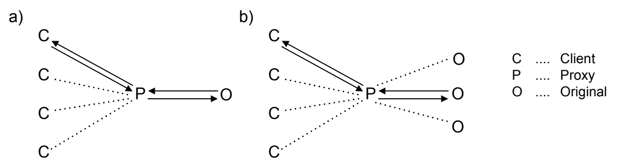

Figure 2. Proxy Pattern

In most proxy types there is a 1:1 relationship between proxy and original (Fig. 2 a), i.e., a proxy is a placeholder for exactly one component. However, there are two exceptions, remote proxies and firewall proxies, where a proxy is responsible for multiple originals (Fig. 2 b, details can be found in [3]). The relations in this pattern are:

1. P handles requests from **C** as a proxy[1]
2. P is a proxy for **O**

The notation for these relations that is used throughout the remainder of this paper is

```
1. P.handles_requests_as_proxy(C)
2. P.is_proxy_of(O)
```

B. Broker Pattern

The Broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote (service) invocations. "A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions [3]." According to [1], "a broker's foremost goal is to achieve location transparency of servers/services. […] The broker is responsible to locate a server/service that can handle a given request. Then the broker forwards the request to the appropriate component, receives its response and delivers the response to the client." In contrast to a proxy, a broker does not perform any pre- or post-processing. Fig. 3 depicts a broker pattern. Broker B sends messages received from a set of clients **C** to a pool of servers **S**.

The relations in this pattern can be expressed as follows:

1. B brokers requests from **C**
2. B brokers requests to **S** or **S** receives brokered requests from B

The notation used in the following is:

```
1. B.brokers_requests_from(C)
2. B.brokers_requests_to(S)
```

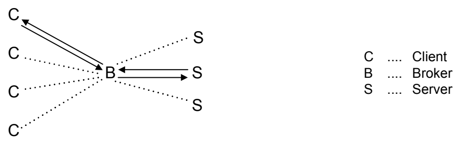

Figure 3. Broker Pattern

C. Master/Slave Pattern

The SE domain defines a Master/Slave (M/S) pattern as follows: "The Master-Slave design pattern supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results these slaves return [3]." The M/S pattern is illustrated in Fig. 4.

[1] Bold font depicts a vector

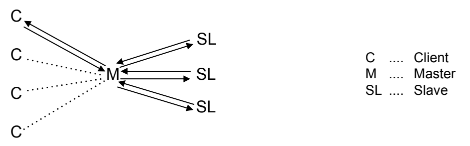

Figure 4. Master/Slave Pattern

Fig. 4 shows that M acts as a master and uses the set **SL** as slaves. The relations are:

1. M creates subtasks from **C**'s requests
2. **C** is a set of slaves to M

In the following these relations will be expressed as:

```
1. M.creates_subtasks_from_requests(C)
2. C.is_slave_of(M)
```

III. RELEVANCE OF INTERACTION PATTERNS IN CONTEXT SHARING

In this section we introduce interaction patterns to context-aware collaborative environments and describe how we intend to use patterns in order to enhance the existing context sharing approach.

A. Interaction Patterns in Context-aware Collaborative Environments

In [1] an algorithm is presented that extracts interaction patterns from log files provided by collaborative systems. In our system we assume that there exists such a pattern discovery feature. It is important to note that context information should also be used during pattern discovery. Typically, an interaction pattern between a number of entities will not be valid for *all* their interactions, but will more likely occur only when these entities are in a certain context.

Consider context type "Activity" and suppose its value is known for all users in a collaborative system at any given time. In other words, all interactions are connected to some context information regarding "Activity". Such context information may also be represented as a hierarchy as depicted in Fig. 1. Users may be involved in different projects, or work packages, and it is very well possible that individuals take up different roles in different activities. The conclusion of this is that the *same* users may be actors in *different* interaction patterns *depending* on the "current activity"-context, or possibly even depending on other types of context. User A may be a proxy of user B in work package X (`A.is_proxy_of(B)`) while being C's slave in work package Y (`A.is_slave_of(C)`). We emphasize this attribute of patterns, i.e., the sensitivity of patterns to context, and see it as implicit throughout the remainder of this paper. Therefore, unless otherwise noted, the term (interaction) pattern should in the following always be seen to be in connection with a given context. For completeness, we point out that for a pattern to be able to occur, the actors do not have to be in the *same* context. For example, consider B to be (statically, for now) in context "project 1". He may create subtasks from user A's requests when A is in context "project 2" (M/S pattern) and execute A's requests himself when A is in context "project 1" (not reflected by any of our patterns). The M/S pattern between A and B is therefore valid in the constellation `context(A)=="project 2"` and `context(B)=="project 1"`.

B. Using Patterns in Context Sharing

The original context sharing framework allows entities to subscribe for notifications of other entities' context changes. When doing this the client is required to specify the level of detail of context information he wishes to receive. Let us look at an example and how context subscriptions and notifications would take place without considering a particular pattern/role.

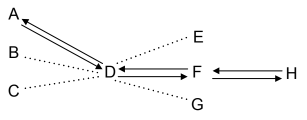

Figure 5. Interaction Pattern among Actors

Suppose we have a configuration as depicted in Fig. 5. Entity D handles multiple requests of the same type received from entities A, B, and C. D checks whether a particular request is valid. If valid, the request is forwarded to E, F, or G, if an invalid request was received, D simply rejects the request. D collects responses from E, F, and G, and replies to the requesting client. Such a scenario can be found in technical support or processing of insurance claims. Suppose the customer support (i.e., D) receives requests concerning similar issues or claims and distributes those requests to agents according to their workload and availability.

In our example D, E, F, and G are collaborating entities and share context among each other. As mentioned above, our context sharing framework allows D to subscribe to context information such as availability and status changes of E, F, and G. D needs to add all agents to its buddy list and subscribe to context information and changes. Context information such as *InOffice/Offline*, *Busy*/*Idle,* and *Communication Capabilities* would probably be sufficient for this case. Note, subscriptions, type of context information to what granularity need to be configured manually (to some extend), thus configuration tasks may be a burden for the user as complexity due to number of users, devices, etc. increases.

Besides configuration issues, let us suppose now that F delegates subtasks to agent H (e.g., second level support). Since D is not aware of this communication, context information would only be exchanged between D and F; and also between F and H, though H's context is relevant to D as well (H is a "*hidden*" actor from D's viewpoint).

We now intend to enhance the original system by using knowledge about interaction patterns in order to support the user in two ways. Firstly, we want to provide the opportunity to request subscriptions without having to specify the desired level of detail. And secondly, we want to provide automatic subscription to context information of other entities of interest, which the subscriber is possibly unaware of. We call the first feature *"determining relevant level of detail (of context information)"* and the second *"automatic subscription to third parties"*. Both of these features are implemented using rules which incorporate knowledge extracted from interaction patterns. A third feature called *"automatic updates of subscriptions"* builds upon automatic subscriptions to third parties.

Considering the example in Fig. 5, according to our definitions in previous sections, we can classify the interaction scenario between D, E, F, and G as a *proxy pattern*. D receives requests of the same type, does pre-processing and forwards requests to E, F, or G. Note that, in contrast, a broker does not perform any pre- or post-processing. The type of interaction between F and H depends on the communication, whether F simply forwards the request to H (broker) or splitting a task into subtasks (M/S pattern). It is important to note that causal knowledge is required to classify complex patterns correctly [1]. As an example for causal dependency: D sends a request to F, F interacts with H, hence F's request to H must be a relevant communication within the given request "context" initiated by D. In practice, this link or causal dependency can only be obtained if we have *a priori* knowledge about the process or particular activity. At the same time it is important to filter links or interactions that are not causally related in order to classify patterns correctly and to reduce computational complexity.

TABLE I. EVENTS AND PATTERNS

Event	Action	Example
Node/Entity joining and pattern detected	Apply context subscription based on similarity	D in Fig. 5 has context subscriptions to E and F. D interacts with F in proxy fashion, thus context subscription to F applied.
Role of actor detected or changed	Adjust context being shared based on rules	D is F's proxy. F is H's master. D receives additional context information such as F's workload.
Causal dependency in interaction detected	Context subscription based on rules	D forwards a request to F, F delegates to H. Context is also shared between D, H.

In Table 1 we illustrate some events and corresponding actions that have impact on the context being shared and exchanged between entities. In the following section we will outline some initial rules that are required to realize a pattern based adaptive context sharing.

IV. REALIZING PATTERN-BASED CONTEXT SHARING

In this section we present concrete examples of how pattern knowledge may be used to formulate rules for context sharing.

A. Pattern-aware Context Processing

In our following discussion we will focus on how to use patterns in the context sharing platform. The existing system already contains a repository for two types of rules. These are privacy rules and dominance rules. Privacy rules restrict access to context information at certain levels in order to preserve the user's privacy. Dominance rules describe the concept of ordering context according to importance. In case of certain context conditions (e.g. Availability == Offline) context of another type (e.g. Location) is no longer relevant for sharing. We now add a third type of rules, which we call relevance rules. Relevance rules should be used to determine the relevance of context information to a given subscriber. Fig. 6 depicts a block diagram showing additional components and also highlights an algorithm that processes a client's context request, which may be either a subscription or query, based on relevance rules.

Fig. 6 shows two entities, context requestor and provider (note, naturally these roles do not need to be mutually exclusive). Depending on the interaction pattern, these entities may have certain roles and relationship in which the two entities stand. This relationship is determined by the previously introduced interaction *patterns*. Without going into detail, the system needs to perform several steps to extract patterns/roles. First, features need to be extracted from log files (relevant work can be found in [5]). Then, patterns are detected and classified. An algorithm aims to detect whether a pattern exists and classifies a given pattern based on *a priori* knowledge (i.e., given classes such as proxy, broker, M/S).

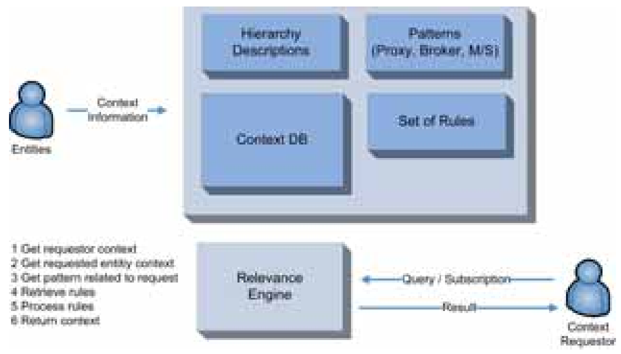

Figure 6. Pattern-aware Context Architecture

We will focus in the following discussion on the relevance engine, which determines context information to be shared based on roles, relations, and rules. The algorithm in Fig. 6 does following steps:

- Get context of *requestor* and *provider* (step 1 and 2). We argue that the context of the requesting entity is important as well because the relevance of the delivered context information depends on the requestor's particular situation (e.g., being mobile, busy, etc.).

- Determine whether entities stand in a specific relationship (step 3). This information is detected by the above mentioned modules, i.e., logging and pattern extraction.
- Retrieve and process relevance rules (step 4 and 5).
- Store subscription or return processed context information (step 6).

A rule is composed of two parts, a condition and an action, *if-then* statements. When the condition is met, the action is executed. The *if* portion contains conditions (such as `activity equals "TRAVELLING"`), and the *then* portion contains actions (such as `set_preferred_voice_device = Mobile_Phone`). The advantage is that rules are easy to understand and can be modified by the user. Another type of rules are *model based rules*. E.g., `If context_source_confidence < 10% Then discard result`. Our system comprises general and specific rules:

- Level rules (**L**), which determine the level of detail
- Generating rules (**G**), used for generating new subscriptions

B. *Determining Level of Detail of Context*

The first application of pattern-based context sharing is determining the relevant level of detail of context information between two entities. The idea is to allow a subscriber A to send a subscription request to (changes in) B's context without having to specify the desired level of detail. This level of detail is then determined by a level rule in the system's repository.

We argue that such relevance may in some cases depend on the interaction pattern or, to be more precise, on the relation, in which these entities stand. As stated in Section 2, the relation of two entities is directly derived from interaction patterns.

We define relevance, or relevant level of detail of context information, as a function: `Relevance = F(context(A),relation(A,B),context(B))`, where A and B are the entities, or users, involved in a context subscription request.

The steps involved when determining the relevant level of detail are the following. This process is triggered by the event of an entity sending a subscription request to the system. In the following we assume that a request by entity A was received to be informed of context changes regarding entity B's location context.

1. Compute relation(A,B).
2. Scan repository for level rule X that applies to relation(A,B), context(A), context(B).
3. Assuming such a rule was found, issue subscription at the level of detail specified by X.

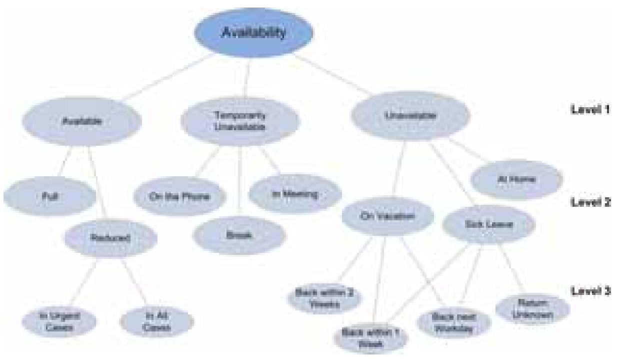

Figure 7. Availability Hierarchy

The following example should make the usefulness of level rules clear.

1. User A subscribes to be informed of changes regarding user B's availability context.
2. The system discovers that these entities stand in the relation "B brokers A's messages".
3. The system scans its level rules and discovers the rule
```
IF X.brokers_requests_from(Y)
AND TYPE_OF_CONTEXT=="Availability"
[AND context(X).equals("xyz")
 AND context(Y).equals("xyz")]
THEN LEVEL_OF_DETAIL="level 1"
```

In natural language this rule translates to "If entity X brokers Y's messages to others and the type of context named in the subscription is "availability" the subscription should be at level 1" (see Fig. 7). Such a rule is justified by the argument, that a broker does not execute tasks himself but only directs them to an appropriate server. Therefore, the sender of tasks only needs to know whether the broker is available or not, but not, for example, whether availability is reduced. Brokering work might be considered "easy and quick", and detailed context information such as "in urgent cases" can be considered irrelevant to X.

4. The system issues a subscription for B's availability context at level 1.

C. *Automatic Subscription to Third Parties*

Another way of integrating patterns in context sharing is to provide automatic subscription to third parties' context information based on relations between entities. The interaction patterns we have presented contain at least 3 involved entities, for example, client (C), proxy (P), and original (O) in the proxy pattern. In such a proxy pattern it is very well possible that C is unaware of the fact that he/she communicates with a proxy only rather than the original server object. However, we argue that the server's context may be just as important to the client as the proxy's, if not more so. As an example of such a situation we give information regarding availability. The system can provide such additional information to the client through *automatic generation of subscription to third parties*. In the following we describe the steps performed in such a situation.

1. Entity C subscribes to context information of entity P.
2. The system scans its relations repository and discovers that P acts as a proxy for O whenever P receives requests from entity C.
3. The system scans its repository of subscription generating rules and discovers a rule that states
   ```
   IF P.handles_requests_as_proxy(C)
   AND P.is_proxy_of(O)
   THEN issue_subscription(C,O)
   ```
4. The system saves C's subscription to P's context and issues a second subscription to O's context.

The following example should further clarify this application.

1. User A requests to be informed of changes regarding user B's availability context.
2. Upon receiving this request, the system scans its relations repository and discovers the relations `B.creates_subtasks_from_requests(A)` and `C.is_slave_of(B)`
3. The system scans its repository of subscription generation rules and discovers a rule
   ```
   IF X.subscribed_to(Y)
   AND Z.is_slave_of(Y)
   THEN issue_subscription(X,Z)
   ```
4. The system issues subscriptions for A not only to user B's context but also to user C's context.

The above example rule is based on a master/slave pattern. We argue that a client subscribing to a master's availability context is likely to be interested in the slave's availability information as well. In the above rule it is assumed that the automatically generated subscription goes to the same type of context, i.e., availability, and to the same level of detail. It may also be justified to issue automatic subscriptions to the server's context in a proxy pattern at an even higher level of detail than the subscriptions to the proxy object, assuming that the server is more important (to the completion of a task) than the proxy.

We point out that the described process of automatic generation of subscriptions may be triggered by two different events. The first option is the one described in the previous example, i.e., a user's subscription request. The other event that may trigger automatic generation of subscriptions is the detection of a pattern that was previously unknown. It is, of course, the nature of patterns that they are only discovered after a certain period of time and when a sufficient amount of log data is available [1]. Therefore, during runtime of the system patterns may and will be detected. Such a pattern detection event should, of course, trigger the same mechanism as if the user just subscribed to an entity involved in the pattern. Therefore, whenever a pattern is detected the system performs the previously described steps beginning at step 2.

D. *Automatic Updates of Subscriptions*

This scenario builds upon the features described in the previous subsection and addresses the possible dynamics of interaction patterns. We consider a master slave/pattern where B creates subtasks from A's requests and hands them over for execution to slaves C1 and C2. We assume that at some point in the past the system automatically generated a subscription of user A to certain context information of C1 (and C2), analogously to the example provided in the previous subsection.

The key event in this scenario is that at some point in time the system discovers that A's requests are no longer delegated to C1 and C2, but instead to C3 and C4. In other words, B uses new slaves for the processing of A's requests. The system addresses such cases by an *automatic update of subscriptions*. When a pattern change is detected, the following steps are performed:

1. The newly discovered instance of the pattern is compared to the previous pattern and the change in actors is determined.
2. The system scans its log of generated subscriptions for subscriptions that were issued based on the outdated pattern, i.e. B sub-tasking to C1 and C2.
3. The system collects all subscriptions that were generated based upon the outdated pattern and determines their *quality*, i.e., it checks if the generated subscriptions were kept up by the user or if they were cancelled.
4. In case of a successful quality check, the automatic subscription rule is considered to be fitting for that users needs. Therefore, new subscriptions regarding the newly discovered, exchanged actors are issued. The outdated subscriptions are dropped.

In this scenario, some new components of the system are introduced. In order to implement the just described algorithm, the system needs to keep a careful record of all generated subscriptions. A record of a generated subscription needs to contain

1. The subscription itself
2. The rules, based upon which the subscription was generated
3. An indicator of the subscriptions quality, i.e., a record whether the subscribing entity kept up the subscription.

Unlike in the first two scenarios, the automatic update of subscriptions is not performed based on rules. This is because the rules have already been applied in the original generation of subscriptions. When updating subscriptions due to pattern changes the question is only whether the same subscription should be reissued to the newly discovered actor's context or not. This question is answered by whether the user kept up the original subscription. In that way, the system could be considered to be "learning".

V. RELATED WORK

Dustdar and Hoffmann [1] describe three patterns (Proxy, Master/Slave, Broker) found in collaborative environments and illustrate algorithms to classify patterns in Caramba, an ad-hoc collaboration software.

In [4] we have introduced hierarchies to enhance context sharing in mobile environments. We implemented a hybrid push/pull mechanism that allows querying or subscribing to context data. The current approach automates the subscription part by providing more relevant context which at the same time improves on required bandwidth as well as lowers administrative overhead. This paper picks up from our latest work in [6] where we have focused on providing the basis for relevance based sharing.

Kilander et al. [7] introduce a framework for managing distributed context data. Applications receive context information through subscription at a context manager by means of production rules. The authors expect these rules to be supplied by the application programmer during the design phase. Different levels of detail are mentioned as an exemplary basis for distribution policies. Yet, the authors remain on a completely abstract level regarding context modeling, subscriptions and actual sharing.

Costa et al. [8] designed a platform for mobile context-aware applications including the WASP subscription language (WSL). Besides lacking support for information granularity, the applied subscriptions resemble ECA rules thus triggering actions rather than exchanging context. WSL rules are also targeted at quite static situations without the need to adapt to changing context.

Bose et al. [9] propose a system to distribute enterprise context information in a collaborative environment. Their XML-based context data format is specifically targeted to the use on mobile devices and describes former and current collaborative interactions. This context information can be transferred between devices and also used offline. Yet, the authors neither present techniques to control the amount of information shared nor mechanism to notify about changes.

VI. CONCLUSION AND FUTURE WORK

In this paper we have shown how the knowledge incorporated in interaction patterns may be used to enhance and automate the sharing of context information between collaborating users.

We have used patterns to formulate rules that determine relevance of context information between users that are known to stand in a specific relation to each other. In another application we have shown how to use patterns to raise team awareness by automatically providing context information about third parties that the receiver of the information was possibly unaware of but which are, nevertheless, relevant in his context.

So far, rules in our systems are either user defined or provided by a model built using expert and/or experience knowledge. In our future work we will investigate strategies to automatically generate rules. The mining of log files provided by the original system where users specify relevance of information themselves seems to be a promising approach.

REFERENCES

[1] S. Dustdar and T. Hoffmann, “Interaction Pattern Detection in process oriented information systems”, *Data and Knowledge Engineering*, Elsevier, in press.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System Of Patterns*, West Sussex, England: John Wiley & Sons Ltd., 1996.

[4] C. Dorn and S. Dustdar, “Sharing Hierarchical Context for Mobile Web Services”, *Technical Report*, TU-Vienna, April 2006.

[5] S. Dustdar, T. Hoffmann, and W.M.P. van der Aalst, “Mining of ad-hoc business processes with TeamLog”, *Data and Knowledge Engineering*, 55(2), pp. 129-158, Elsevier, 2005.

[6] C. Dorn, D. Schall, S. Dustdar, “Granular Context in Collaborative Mobile Environments”, *International Workshop on Context-Aware Mobile Systems CAMS’06*, Springer, in press.

[7] F. Kilander, W. Li, C.G. Jansson, T. Kanter, and G. Maguire, “Distributed Context Data Management”, *Workshop on Managing Context Information in Mobile and Pervasive Environments* (MCMP’05), May 9, 2005, Ayia Napa, Cyprus.

[8] P.D. Costa, L.F. Pires, M. van Sinderen, and J.P. Filho, “Towards a service platform for mobile context-aware applications”. *Ubiquitous Computing* - IWUC 2004. (2004) 48–61.

[9] S. Bose, A. Ennai, and S. Sohi “Portable Enterprise Collaboration Contexts” *Collaborative Computing: Networking, Applications and Worksharing*, Dec 19-21, 2005.