Handling Conflicts in Autonomous Coordination of Distributed Collaborative Activities

Jörn Franke, François Charoy, Cédric Ulmer

Handling Conflicts in Autonomous Coordination of Distributed Collaborative Activities

Jörn Franke, François Charoy and Cédric Ulmer

Abstract— **Coordination between different organizations in dynamic situations, such as a disaster response, is challenging. Organizations are autonomous and coordinate the situation from their point of view. There is no central authority to coordinate all operations. To coordinate their actions, organizations need to exchange information on what they are doing. However, they cannot share everything with everybody due to privacy, regulatory or strategic reasons. Currently, only e-mail, telephone or fax are used to exchange information. This makes detecting and handling of conflicting views on the situation very difficult. We propose an approach for inter-organizational process management for these kinds of dynamic scenarios. It allows different organizations to share selected activities by replicating them in the different workspaces of the organizations. State changes of shared activities are propagated optimistically. We explain detecting and handling of two different types of conflicts that can occur in this setting. We provide an implementation and interviews to validate the concepts.**

I. Introduction

According to Gartner and McKinsey the management of activities in flexible distributed processes becomes more and more important for many organizations [1]. Even more important, we can say that not only the processes require flexibility, but also the coordination of activities by several autonomous organizations with shifting goals. In this paper, we consider processes that take place in the "real world" involving humans belonging to different organizations. We use the disaster response management domain as a critical example where coordination is important and where goals can change during an event. These processes cannot be fully described in a structured manner due to their dynamic nature. From an inter-organizational perspective we also argue that it is not possible to create a global shared process that is coordinated by one entity. Each organization coordinates its own activities based on its experience and governance rules. However, in order to coordinate with each other, these organizations need to share information about what they are doing and how they rely on activities of other organizations. Today, it is mostly done using communication tools that are not the best tools for coordination (e.g. email, phone or fax). It is difficult to establish an adequate situation overview and to model coordination using these tools. We explain in this paper how an activity management system can be designed to address this problem. We assume that there is no central coordination, but a network of organizations that need to synchronize their actions. The network structure is not known in advance. Organizations need to exchange information about what they plan to do, are doing or have done. They may also ask others to do something. This will require to share information about activities and to allow concurrent changes of this information. This may cause conflicts and we will show how they can be detected and handled. Our work is based on an approach presented in [2]. This paper is an extension taking into account the inter-organizational dimension, where contrary to a centralized solution, not everything can be shared amongst the different organizations (e.g. the police cannot share information related to crime investigations with the fire fighters). This has, for example, regulatory, privacy or strategic reasons. The main contribution of this paper is how conflicts caused by this can be detected and handled in this distributed setting.

In the next section, we describe a realistic use case in the field of disaster response management where distributed coordination is needed. It has been developed together with end users [3]. We explain how the coordination, i.e. activities and their dependencies, is modeled (based on the framework proposed in [2]) in section three. We describe then how this model is leveraged on the inter-organizational level to coordinate activities of different organizations in section four. Particularly, the focus is resolving conflicts when sharing activities and integrating them in the processes of different organizations. We describe the architecture in section five and the implementation in section six. In section seven, we discuss end user feedback. Finally, we describe related work and give an outlook on future research.

II. Use Case

To illustrate our work, we present here a use case that demonstrates the need for flexibility and coordination among autonomous organizations. It has been derived from a realistic disaster response use case developed in the SoKNOS project [3] together with end users, such as fire fighter and police commanders. The organizations need to work together, but none of them is hierarchical superior to the other. They form an organizational response network. In the simplified version of the use case three organizations respond to a flood: the police, the fire fighters and the military. The military is responsible for protecting a chemistry plant from the flood. They fill sandbags, transport sandbags and build a dam to fulfill this objective. The fire fighters are building a dam to protect a residential area from the flood. They rely on the military to provide sandbags to them. The police has to evacuate the residential area in case of a flood. They have to determine people, warn the people, order shelter

J. Franke and C. Ulmer are with SAP Research, France joern.franke@sap.com

F. Charoy is with LORIA-INRIA-CNRS, Université de Lorraine, BP 239-54506 Vandoeuvre-lès-Nancy Cedex, France charoy@loria.fr

or transport people. Some of these actions depend on the outcomes or on the status of others. Police activities depend on the success of the dam construction. The fire fighters rely on the delivery of sandbags by the military. It is beneficial for each organization to have knowledge about what is actually happening and what may concern the other. But of course, each organization needs also to keep some of their actions private (i.e. everything cannot be shared) for privacy reasons or due to internal policies. The organizational network can be extended at any time. For example, different regions or states may provide additional command centers for supporting the coordination among different disaster sites. In this case, actions may also have an impact on each other (building dams at two different places for instance). During our research within the SoKNOS project and interactions with end users (e.g. workshops) we found out that current means, such as email, telephone or fax, cause some problems for coordination. For example, it is almost impossible to get an accurate overview of the status of all ongoing actions. Some organizations may think that something is happening while it has failed (an order to close an airport has been given and is assumed to be completed while it is not). It can be very difficult to detect these conflicting views using the traditional means. This leads to confusion about the current situation.

We argue that a process based approach to manage this coordination can address these challenges. In such an approach, we consider that the organizations are autonomous and coordinate the situation from their point of view. There is no central entity defining how to do the coordination in detail for all organizations.

III. A Framework for Coordination of Activities

We describe in this section how activities and temporal dependencies are modeled by users to describe explicitly coordination (based on the framework explained in [2]). The cited work describes also how this model is verified and executed and we will reuse this functionality in the following sections.

Definition 1 An *activity type* $at_d = (S, st, se, f, G)$ represents the management lifecycle of an activity with S is a finite set of activity states, $st \in S$ describes the start state of an activity type, $se \in S$ describes the end state of an activity type (i.e. a state where no further transition is possible), $st \neq se$ a start state is not an end state and $f : S \rightarrow S$ is a transition function defining the possible transitions from one state to another for one activity type. The specification of the activity type can be extended by governance rules $G = \{g_1, .., g_n\}$. They describe who can transit from one state to another, e.g. $g_x \subseteq f$ is the transition function of the role "x". Fig. 1 illustrates an example for such an activity type. The white circle describes the start state and the black circle describes an end state. Other states are "Plan", "Execute", "Idle", "Fail", "Cancel" and "Finish". We do not allow strongly connected components (i.e. cycles) in the activity type, because this causes confusion (cf. [2]). For example, it would be possible to go from state "Execute" to "Fail" and vice versa. This is difficult to display and understood by the user. Particularly, when the status is shared with other users.

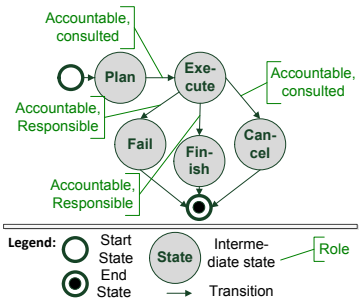

Fig. 1. Example for an activity type with governance roles

Definition 2 An *activity* is defined as $a_i = (uid, name, cat, cs, P)$ where uid is a unique identifier of the activity, $name$ describes the activity, cat is the activity type of the activity, cs is the current state of the activity. On creation it must be the start state st of its activity type. P is the set of participants assigned by the creator of an activity to a governance role in $cat.G$. An activity can change its state in parallel to other activities without affecting them. However, a dependency can be established between activities, if it is perceived by the user as important. Any further data can be attached to the activity.

Definition 3 A *temporal dependency* is defined as $d_i = (a_s, s_s, a_d, s_d, type)$ with a_s is the source activity, s_s is the state of the source activity, a_d is the destination activity, s_d is the state of the destination activity and $type$ is the type of temporal dependency. We use Allen's proposed time interval relationships for describing different types of temporal dependencies [4] (see Fig. 2 illustrating seven of them and omitting six inverse dependencies). The dependency changes its state ("Violate" or "Neutral") depending on the order of the state changes of the associated activities (cf. [2]). An example for a temporal dependency is the dependency "overlaps" between the states "Execute" of activity "A" and "B". This means that activity "A" has to enter state "Execute" and then later activity "B" can enter state "Execute". Activity "A" has to leave state "Execute" before activity "B" does. If this does not happen in this order then the dependency is violated. This can happen, because the situation requires this or people of other organizations are not aware of this dependency. In this case the user is made aware of this and can take appropriate actions, such as communicating with the stakeholders of the activities or by creating new ones.

We will provide in the subsequent sections examples demonstrating how this model is used on the inter-organizational level to coordinate activities. We explain the nature of conflicts that may occur in this distributed setting, how they can be detected and managed.

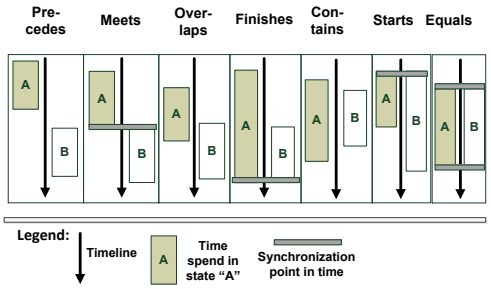

Fig. 2. Types of supported temporal dependencies

IV. Coordination and Conflict on the Inter-Organizational Level

As we explained in the use case section, organizations need to share information about their actions to coordinate. In this section, we will describe how they can coordinate by sharing activities. We consider that each organization maintains an activity workspace (AW) containing all its own activities and dependencies. A person from one organization can decide to share an activity with a person of another organization. The selection of organizations is based on an existing social network (e.g. the fire fighter commander knows the police commander) and is out of the scope of this paper. The person of the other organization can then decide to insert this activity in its AW. This preserves autonomy of both organizations. The shared activity is then replicated in the AWs of both organizations and can be managed like any other activity of the workspace. New dependencies can be created from and to this activity. The status of this activity can also be changed in both AWs and changes are propagated optimistically. Thus, both organizations will have a shared view on the current operations. Optimistic replication means that concurrent state changes can occur that may conflict or have different outcomes regarding dependencies violation. In the remainder of this section, we detail the general principles of the approach and our proposal to deal with conflicts.

A. Sharing of Activities

In our approach, participants model activities and dependencies on an activity workspace (AW). They can share some activities with other participants of another AW. They can establish dependencies between shared activities and their own activities. In Fig. 3 we describe an example for sharing of activities. In the first step (T1), the fire fighter commander shares the activity "Build Dam" with the police commander. In a second step (T2), the police commander has integrated the shared activity "Build Dam" in his AW and created a dependency from the shared activity to his activity "Warn People".

B. Updating States of Shared Activities

We describe in this section how state changes of shared activities are propagated to all AWs, where the activity is replicated.

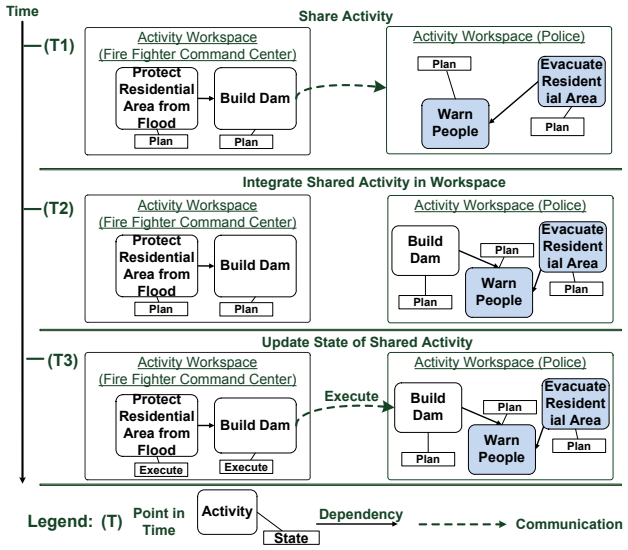

Fig. 3. Example for sharing of activities, integration of shared activities in an activity workspace and updating the state of shared activities

We propose to do this optimistically, i.e. we propagate the state change and detect as well as handle conflicts afterwards. This allows coordinating in an instant like with traditional means. A pessimistic approach would mean to lock the activity for a period of time in which no state changes can be entered by the user. This would limit unnecessarily the possible interaction with the system (cf. also [5]). A pessimistic approach can lead to inaction, because people have to wait until they can provide input or receive input to do action. This is contrary to what happens in disaster response management. Thus, a pessimistic approach is not possible in our use case. In Fig. 3, we illustrate in step three (T3) that the fire fighter commander changed the activity "Build Dam" to state "Execute". The AW of the fire fighter commander sends the update to the AW of the police commander, because the activity "Build Dam" has been shared with the police commander before.

We presented in [6] a protocol for optimistically propagating state changes. The underlying assumption is that messages arrive eventually. The outcome of the protocol is that state changes are applied in any AW where the shared activity is replicated. Applying a state change in a model means detecting if dependencies are violated by it (cf. for a detailed explanation [2]). Since the protocol only provides optimistic replication, we need to detect and handle conflicts afterwards. We describe two important types of conflicts that we have identified in the next subsections.

In the next two subsections we describe detecting and handling of two different types of conflicts that can occur after optimistic propagation by this protocol.

C. Detecting and Handling Conflicts of one Shared Activity

The first type of conflict occurs when the state of one shared activity is changed into two different states concur-

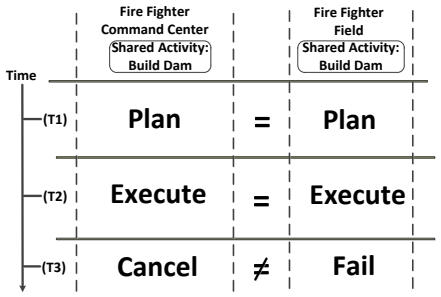

Fig. 4. Example for detecting conflicts caused by state changes of the shared activity "Build Dam"

input : Execution history σ of one activity with conflicting state changes

conflictingstatechangeslist ← `GetConflictingStateChanges(`σ`);`
chosenStateChange ← conflictingstatechangeslist[0];
for $i \leftarrow 0$ **to** *conflictingstatechangeslist.size - 1* **do**
  if *chosenStateChange.role* < *conflictingstatechanges[i].role* **then**
    chosenStateChange ← conflictingstatechangeslist[i];
  end
end
`SetCurrentState(`chosenStateChange`)`

Algorithm 1: Handle conflicting state changes of one activity

rently. For example, when considering the activity type in Fig. 1, a conflict can occur if one person sets an activity based on this activity type to state "Cancel" and the other one to state "Fail" concurrently. This type of conflict is illustrated in Fig. 4. The activity "Build Dam" is changed by the commander in the command center to state "Cancel" and by the commander in the field to state "Fail" in the third step (T3). If the protocol above is used, the conflict can be detected based on the activity type and the history of state changes of the activity. In the activity type, it is impossible to transit from state "Execute" to "Fail" and at the same time from "Execute" to "Cancel". Thus, there is a conflict.

Definition 4 *Conflicting state change history*: Let $\sigma_y = (s_1 \rightarrow s_2, .., s_{n-1} \rightarrow s_n)$ be the execution history with the state changes $s_1 \rightarrow s_2, .., s_{n-1} \rightarrow s_n$ of activity y based on the activity type at. A conflict occurs when: $\exists((s_i \rightarrow s_j) \wedge (s_i \rightarrow s_k) \wedge (s_j \neq s_k)), i = 1..n-1$. This definition means that there is conflict in the history of state changes if there are two or more state changes originating from the same state (s_i) of the same activity.

As mentioned, we assume that all AWs sharing the activity have eventually the same elements in their execution history. We cannot go twice through the same state without causing conflicts in the history with the activity type. This is only possible when the activity type has cycles, which we excluded by definition.

However, if this conflict is handled manually by the user then it cannot be guaranteed that it will be resolved eventually. Thus, we propose an automated approach. Our approach is inspired from [7], but we adapted it to our requirements, where we do not have a central authority. It is based on the governance roles defined by the creator of an activity (see previous section) to resolve automatically the conflict. For example, the commander in the command center has shared the activity "Build Dam" with the commander in the field and both perform conflicting state changes. The commander in the command center changes the state to "Cancel" and the commander in the field to "Fail". Since the commander in the command center has the accountability for the activity, he is higher in the role hierarchy than the commander in the field who is responsible. This means the final state in both AWs is "Cancel" for the shared activity "Build Dam". We now present an algorithm to handle conflicting state changes of the same activity automatically. We assume that the governance roles of the activity type form a hierarchy (e.g. accountable > responsible > informed). If the roles are on the same level of the hierarchy then the conflict can be highlighted to the users and they can resolve the conflict manually. Every time a conflicting state change of the same activity is detected, each AW performs algorithm 1. This algorithm ensures eventually that there is one agreed activity state, when there was one conflict.

Using algorithm 1, the AWs select the state set by the highest governance role. This means each AW reaches eventually the same state for the activity, because they receive eventually a history of all state changes. In more complex activity types there is the possibility of several conflicts. For example, let's assume the activity type of the activity "Build Dam" is extended by adding two further states "Complete Failure" and "Partial Failure" after the state "Fail". This means there can be a conflict, when the activity is changed from "Execute" to "Finish" by the fire fighter commander in the field and from "Execute" to "Fail" by the fire fighter commander in the command center. Then, the military commander changes it from "Fail" into "Partial Failure" and the fire fighter commander in the command center changes into "Complete Failure". There are now two conflicts. The algorithm can be extended to resolve several conflicts by applying it to all conflicts and by removing state changes causing the conflicts from the history. We omit the details here due to space restrictions. It should be noted that the algorithm is not about handling "wrong" states. Although the states are conflicting, each party (fire fighter commander in the command center, fire fighter commander in the field or military commander) may have justified reasons for its own view. The main goal of the algorithm is to converge to a common view based on strategic direction and defined governance roles. This does not require additional communication, because we assume that eventually the history of state changes of an activity contains the same items in all

AWs.

D. Detecting and Handling Conflicts of Shared Activities with Dependencies

The second type of conflict can occur when two shared activities, connected via one dependency, change their state and this may lead to the case that the same dependency in different AWs is in different states (e.g. in one "Neutral" and the other "Violated"). This conflict is different from the previous one and the situation causing it is illustrated in the upper part of Fig.5. The military commander has shared the activity "Transport Sandbags" with the fire fighter commander in the command center. The fire fighter commander has created a dependency "overlaps" to his own activity "Build Dam". The own activity "Build Dam" has been shared with the fire fighter commander in the field. In the bottom part of the figure, we illustrate the problem as a sequence diagram. We assume that the military commander changes the activity "Transport Sandbags" to the state "Execute" and the fire fighter commander in the field changes the activity "Build Dam" to state "Execute". The fire fighter commander in the command center cannot determine the order of state changes properly, because there might be delay when transmitting the state changes. This means the state change of the military commander is received after the state change of the fire fighter commander in the field, although the military commander changed it before the fire fighter commander. This can also lead to a different temporal order of state changes in different AWs, because each AW can receive state changes in different orders. This means they have a conflicting view on the situation. This implies that we need to ensure global causality eventually, so that all participants have the same view on the situation.

Definition 5 *Eventual global causality*: $a_x : s_i < a_y : s_{i+1} \rightarrow C_j(a_x : s_i) < C_j(a_y : s_{i+1}) \, \forall AW j = 1, ..n$ sharing activity x and/or y. This definition means that when state change $a_x : s_i$ happens before state change $a_y : s_{i+1}$ then this needs to be equally observed in all AWs where the shared activities are replicated.

Lamport [7] introduced the notion of virtual time in distributed systems. It is similar to the idea in Definition 5. Virtual time progresses in terms of events, i.e. time stands still when there is no event. An event in our approach is a state change. This notion allows defining of global "happen before" relationships between state changes in each AW. Using this notion, we can also detect in the example which state change happened before the other one (cf. also our activity framework for the non-distributed setting [2]). Ensuring Definition 5 means we need to find a function C for each AW, so that it is able to order the events in the same order like the other AWs. Vector clocks [8] address this by using a vector containing the clock (event counter) of each AW $n : V = (c_1, .., c_n)$. Every time an AW i propagates a state change s to the other AWs, it increases only its counter of the vector clock (i.e. the i-th item of the vector): $V[i] = V[i] + 1$. It attaches its vector clock V to the state change. An AW receiving a state change can now put them in

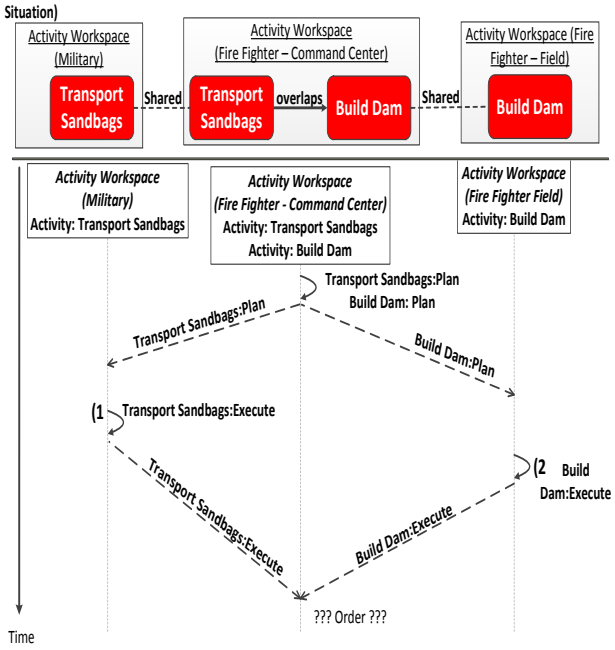

Fig. 5. Example for a situation that can cause a conflicting view on causality

an order by comparing the vectors of different state changes using the following clock function C: s_i (with clock vector V_x) is partially ordered before s_j (with clock vector V_y), if : $V_x[k] \leq V_y[k] \forall k$ (otherwise they are simultaneous). It is always possible to create this partial order (cf. [8] for proofs of these concepts). The ordered state changes can be inputted into the state machine representing the dependency (cf. [2]) to detect if a dependency is violated or not and since it is the same order it will always be the same result in all AWs.

Although the vector clock approach seems to be suitable for our purposes, it has one drawback, because not everything is shared with everybody. This may lead to a situation where it is not possible to establish causality, because some of the AWs do not know about each other. For example, let us assume that in the situation illustrated in Fig. 5 the military commander in his AW changes the activity "Transport Sandbags" into state "Execute" and propagates the state change (with Vector clock $V_{Military} = ((1, "Military"), (0, "FireFighterCommandCenter"))$) to the AW of the fire fighter commander in the command center (illustrated in the upper part of the figure). The fire fighter commander in the field changes the activity "Build Dam" into state "Execute" and propagates the state change to the AW of the fire fighter commander in the command center (with vector clock $V_{FireFighterField} = ((1, "FireFighterField"), (0, "FireFighterCommandCenter"))$. The fire fighter commander in the command center is never able to establish causality in this case, because the AW of the military and the AW of fire fighter in the field do not know their vector clocks.

This is undesired, because it would make the definition of temporal dependencies in this special case useless.

We solve this problem by introducing the following rule in our protocol:

When a vector clock with a state change is received then the AW i increases its own clock c_i and sends the updated clock vector to all AWs it has activities shared with.

The effect of this rule can be illustrated via the previous example. Let us assume that the AW of the fire fighter commander in the command center receives the state change from the military. It then updates its vector clock and sent it ($V_{FireFighterCommandCenter} = ((0, \text{“}FireFighterField\text{”}), (1, \text{“}FireFighterCommandCenter\text{”}))$) to the AW of the fire fighter commander in the field as well as the AW of the military. When now the fire fighter commander in the field changes the activity “Build Dam” to “Execute” it propagates the state change together with the updated vector clock ($V_{FireFighterField} = ((1, \text{“}FireFighterField\text{”}), (1, \text{“}FireFighterCommandCenter\text{”}))$) to the AW of the fire fighter commander in the command center. The AW of the fire fighter commander in the command center is now able to establish causality according to Definition 5.

Approaches in distributed systems (e.g. [9]) expect that everything is shared among everybody and thus the problem does not occur there. This also means that our approach has an advantage in our scenario over the other approaches.

V. Architecture

We implemented our concepts mentioned before to do student experiments with the prototype. They are implemented as an extension to Google Wave that enables instant collaboration with optimistic replication of shared documents (“Waves”). Shared documents can be distributed between different servers of different organizations (illustrated on Fig. 6 as different Wave servers). A “Wave” can have participants from different servers. The reason for choosing a collaboration platform over a simpler platform was to show how our approach works in the context of different tools needed for disaster response management (e.g. text exchange, maps, images or videos). Furthermore, it provides the infrastructure for implementing sharing of activities. Google Wave can be extended in two different ways: “Gadget” and “Robot”. A “Gadget” can be inserted into a “Wave” and is a graphical user interface to provide additional collaboration functionality (e.g. collaborating on images or collaborative modeling). It is rendered within the Google Wave Web Client in a web browser. A “Robot” can be added as an automated participant to a “Wave” and can react on events in “Waves” and modify them. It can also create further “Waves”. Google Wave is in process to be fully open-sourced platform [10].

We illustrate the architecture of our extension in Fig. 6 in context of the Wave Federation Architecture [10]. Activities and dependencies can be modeled in a special “Wave” called “AW-Wave” containing a “Gadget” providing the necessary functionality. The “Gadget” is called “AW-Gadget” and stores its data (e.g. the model) in the “AW-Wave”. The “AW-Wave” can be compared to an AW. A robot is a distributed application on the Google App Engine or any other server. It is responsible for propagating the state changes of activities to different “AW-Waves”. Activities themselves are linked to special “Waves” called “Activity-Waves”. People can collaborate in this activity, e.g. they can insert pictures, write text or work collaboratively on a map of the situation. An activity can be shared by inviting a participant to an “Activity-Wave”. This means Google Wave provides already a mean for sharing and replicating activities as required by our approach. The robot makes the shared activity available in the “AW-Waves” of the participant who has been invited to the “Activity-Wave”. The activity is then shown in the “AW-Gadgets” of the “AW-Waves” of the participant and the participant can replicate it into his/her models. They can also create dependencies to their own activities. State changes can be initiated via the “AW-Waves”, where the activity is replicated. The “AW-Gadget” stores for this a state change as well as the vector clock in the “AW-Wave”. The robot copies then the state change to all “AW-Waves” where the activity is replicated. According to the rule mentioned in the previous section, the robot copies the vector clocks to all “AW-Waves” with which activities have been shared. The “AW-Gadgets” can create the global order of state changes based on the vector clocks. They also highlight violation of dependencies to the user and they display to the user when there have been conflicting state changes according to Definition 4.

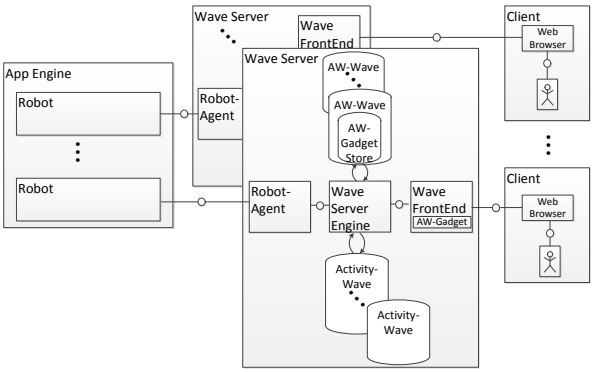

Fig. 6. Architecture of our extension

VI. Implementation

We illustrate a screenshot of our extension in Fig. 7. We support a graphical modeling notation, but as an alternative the activities and dependencies can be represented in a table. The later view seems to be more easy to use when many activities need to be created. The figure illustrates the activity workspace of the fire fighter commander in our use case. It uses a graphical modeling notation. The fire fighter commander sets the activity “Build Dam” conflicting to the field commander to state “Fail” and “Cancel” respectively.

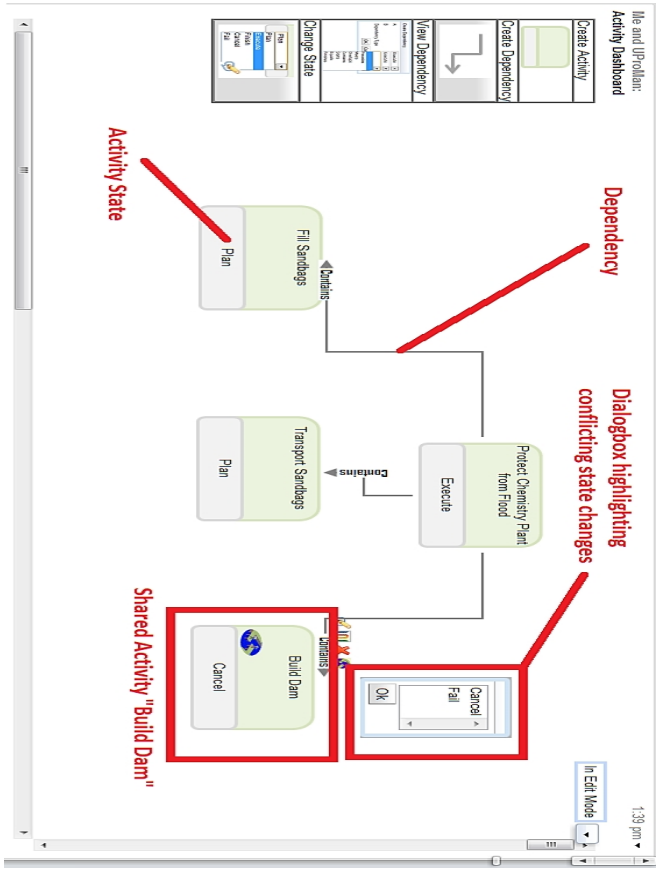

Fig. 7. Screenshot of our prototype

This is illustrated as a symbol on the activity and he has currently opened a dialog box showing the two conflicting states.

VII. Discussion

We started validation of our approach by presenting it to four experienced disaster managers. The general approach of activity management has been commented positively by the disaster managers (cf. [2], [11]). The problems of traditional means for coordination have been recognized by a fire fighter commander (Southern California):

"[..] the pile of messages in the inbox, [which contains] the reality as a situation [..] being able to put them in context and update them and coordinate them to create a common picture is the difficulty".

For example, a fire fighter commander (Washington, DC) highlights that it allows measuring the progress of the situation and managing shifting goals (end states):

"there is a couple things [about your approach], [..] it is a good way to measure progress, and the second thing is that you recognize that the end state [goal] will change because of the dynamic situation of the incident you are involved in [..] the end state may need to be modified or you might have intermediate type of objectives".

He further says that sharing of activities (missions) is important: "[You are able to] define specific objectives that need to be accomplished in order to meet that end state [..] and then [these] objectives [are] transmitted as mission statements to the ground [..] those folks at the ground level [define] the mission, [plan] the mission , [develop] the tasks and tactics [and have to] make time-critical decisions in order to meet that mission".

Another fire fighter commander (Southern California) confirms the previous statements: "[The approach] addresses the problem of coordination and sharing goals and objectives from one organizations to other and it is that communication [to update] information, [such as] progress as far as plan, changes made, it is that communication link that inherently seems to be the crux of all problems. [If this] sharing does not occurs [then] a lot of information stays within each independent organization [..] Without that knowledge we duplicate services, we actually implement plans that interfere with the others, goal and objectives".

Our approach can help them to have a more meaningful and accurate situation awareness on the current state of their own and others' activities. The fire fighter commander (South France) highlights that our approach can make a difference in situation where coordination is the issue: many organizations are involved, it is a geographically distributed situation, there needs to be time to plan and there needs to be time for communication between different actors.

VIII. Related Work

In [2] several process management systems, addressing a disaster response scenario, are compared. They do not take into account an inter-organizational setting. Inter-organizational coordination of activities has mostly been addressed in the area of business process management systems. Aalst and Weske [12] propose to split a previously defined public process in several parts and let every involved party execute its part in a distributed fashion. Grefen et al. define the public process as a contract between organizations [13]. Schulz et al. [14] describe a view-based approach which is similar to the previous approach. Fdhila et al. [15] propose to execute a public process as a choreography. The approaches in [16], [17] allow defining a public process and each involved party can deviate from it. All these systems require to some extent defining a global public process or choreography before the process execution. Based on our end user interactions, we believe this is not an acceptable assumption for autonomous organizations that may even work together for the first time. It is also not always obvious when and if they have to work together. The processes in our use case are not only defined top-down, but also bottom-up. They can be defined vertically and horizontally in a network of organizations. We do not require definition of a public process in advance. Furthermore, these approaches consider mostly sequential processes, which is not a realistic assumption for disaster response processes [2]. They are also limited with respect to detection of concurrent conflicts. Our approach can be compared with the unified activity management approach in [18]. However, they do not consider an inter-organizational distributed setting.

We find many approaches addressing detecting and handling of conflicts in distributed collaborative work (cf. [5]). For example, collaborative image [19] or text editing [20] with optimistic change propagation. These approaches deal with conflicts in unstructured documents (e.g. text or images) and not in structured models like our concept.

Other approaches deal with the consensus problems in distributed systems (e.g. [21], [9]). The consensus problem is different from our concept, because it deals with faulty processes (i.e. processes that send different values to different processes). The problem there is to find a consensus between processes what are correct values. In our approach, such a consensus is not possible, because each party has a different view on the real world (e.g. the command center has a more distant strategic view and the people in the field a more operational view) and consensus protocols cannot deal with this problem. Furthermore, the actors may have different goals and processes that interact with each other, but not in the sense of a distributed algorithm. In our scenario, the notion of faulty processes does not exist. This is why we proposed another approach based on governance roles to resolve conflicting views.

IX. Conclusion

We described in this paper how the coordination of autonomous organizations can be supported by an activity management system. The basic idea is to allow the organizations to share selected activities with each other by replicating them in the different workspaces of the organizations. We presented two different conflicts that can occur when optimistically propagating state changes of shared activities. We explained how they can be detected and also provided mechanisms for handling them. Pessimistic propagation would lead to inaction, requires a constant reliable communication and the dynamics of the situation cannot be captured in time. Using optimistic propagation allows capturing the situation (activities and their state) in time and it enables detection of conflicting views, which is seen as beneficial by disaster managers. Thus, even traditional means only allow optimistic propagation of information. The proposed solution can already overcome some problems with traditional means for ad-hoc coordination (e.g. email or phone) or even more formal BPM approaches (see related work), because they have only limited capabilities for detecting and handling concurrent conflicting views on dynamic processes which are only partially known by each organization. In the future, we want to explore how to establish the social network between actors sharing activities based on pre-defined plans. This also includes the definition of governance roles. The approach has been commented positively by disaster managers [11], [2]. We have already started first evaluations in form of experiments with students. We expect that our approach is applicable to other dynamic collaborative scenarios, such as organization of large events (e.g. Olympic Games) or development projects. Our concepts presented here can also be adapted to extend other constraint-based process management systems to the inter-organizational level (e.g. [22]).

X. Acknowledgements

The research was partially funded by the French Ministry of Research within the RESCUE-IT project [23]. We thank the domain experts for providing us detailed insights.

References

[1] E. Olding and C. Rozwell, "Expand your bpm horizons by exploring unstructured processes," Gartner, Tech. Rep. G00172387, 2009.

[2] J. Franke, F. Charoy, and P. El Khoury, "Collaborative coordination of activities with temporal dependencies," in *18th International Conference on Cooperative Information Systems*, 2010.

[3] S. Döweling, F. Probst, T. Ziegert, and K. Manske, "Soknos - an interactive visual emergency management framework," in *Workshop on Geographical Information Processing and Visual Analytics for Environmental Security*, 2009.

[4] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[5] S. Greenberg and D. Marwood, "Real time groupware as a distributed system: Concurrency control and its effect on the interface," in *ACM CSCW Conference on Computer Supported Cooperative Work*, 1994.

[6] J. Franke, F. Charoy, and C. Ulmer, "Coordination and situational awareness system for inter-organizational disaster response," in *10th IEEE International Conference on Technologies for Homeland Security*, 2010.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[8] F. Mattern, "Virtual time and global clocks in distributed systems," in *Workshop on Parallel and Distributed Algorithms*, 1989.

[9] U. Bartlang and J. P. Müller, "Dhtflex: A flexible approach to enable efficient atomic data management tailored for structured peer-to-peer overlays," in *3rd International Conference on Internet and Web Applications and Services*, 2008.

[10] Google, "Wave in a box, http://www.waveprotocol.org, retrieved 25.08.2010."

[11] J. Franke, F. Charoy, and C. Ulmer, "A model for temporal coordination of disaster response activities," in *7th International Conference on Information Systems for Crisis Response and Management*, 2010.

[12] W. van der Aalst and M. Weske, "The p2p approach to interorganizational workflows," in *13th International Conference on Advanced Information Systems*, 2001.

[13] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig, "Cross-flow: Cross-organizational workflow management in dynamic virtual enterprises," ESPRIT project No, 28635, European Comission, Tech. Rep., 2000.

[14] K. A. Schulz and M. E. Orlowska, "Facilitating cross-organisational workflows with a workflow view approach," *Data & Knowledge Engineering*, vol. 51, pp. 109–148, 2004.

[15] W. Fdhila, U. Yildiz, and C. Godart, "A flexible approach for automatic process decentralization using dependency tables," in *7th IEEE International Conference on Web Services*, 2009.

[16] J. C. Grundy and J. G. Hosking, "Serendipity: integrated environment support for process modelling, enactment and work coordination," *Automated Software Engineering*, vol. 5, no. 1, pp. 27–60, 1998.

[17] S. Rinderle, A. Wombacher, and M. Reichert, "Evolution of process choreographies in dychor," in *On the Move Conferences*, 2006.

[18] B. L. Harrison, A. Cozzi, and T. P. Moran, "Roles and relationships for unified activity management," in *International ACM SIGGROUP Conference on Supporting Group Work*, 2005.

[19] C. Sun and D. Chen, "Consistency maintenance in real-time collaborative graphics editing systems," *ACM Transactions on Computer-Human Interaction*, vol. 9, no. 1, pp. 1–41, 2002.

[20] A. Imine, M. Rusinowitch, G. Oster, and P. Molli, "Formal design and verification of operational transformation algorithms for copies convergence," *Theoretical Computer Science*, vol. 351, no. 2, pp. 167–183, 2006.

[21] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[22] W. van der Aalst and M. Pesic, "Decserflow: Towards a truly declarative service flow language," in *Web Services and Formal Methods*, 2006.

[23] A. Schaad, C. Ulmer, and L. Gomez, "Rescueit - sécurisation d'une chaine logistique internationale," in *Workshop Interdisciplinaire sur la Sécurité Globale*, 2010.