We present a social context as a service (SCaaS) platform for managing adaptations in collaborative pervasive applications that support interactions among a dynamic group of actors such as users, stakeholders, infrastructure services, businesses and so on. Such interactions are based on predefined agreements and constraints that characterize the relationships between the actors and are modeled with the notion of social context. In complex and changing environments, such interaction relationships, and thus social contexts, are also subject to change. In existing approaches, the relationships among actors are not modeled explicitly, and instead are often hard-coded into the application. Furthermore, these approaches do not provide adequate adaptation support for such relationships as the changes occur in user requirements and environments. In our approach, inter-actor relationships in an application are modeled explicitly using social contexts, and their execution environment is generated and adaptations are managed by the SCaaS platform. The key features of our approach include externalization of the interaction relationships from the applications, representation and modeling of such relationships from the domain and actor perspectives, their implementation using a service oriented paradigm, and support for their runtime adaptation. We quantify the platform’s adaptation overhead and demonstrate its feasibility and applicability by developing a telematics application that supports cooperative convoy.
1. Introduction

Mobile computing has brought a wireless revolution in recent years, enabling mobile internet access as an indispensable way of modern life. It has radically changed the way people perform tasks, access information and interact with one another. At the same time, the emergence of service-oriented technology and interoperability standards (e.g., Web Services) has made it possible to develop systems intended to support people’s social activities and organizations’ work. This trend has paved the way of a new breed of software systems that can mediate tasks of both individuals and groups in pervasive environments.

We view a collaborative pervasive environment (CPE) as an interaction space among users where they collaborate with each other towards a common objective by sharing information that has been provided by other participating actors. These actors could be users or services or sensors embedded in the environment. Such collaboration is subject to the agreements and constraints relevant to the relationships among actors, which are dynamic and need to adapt in response to the changes in user requirements and environmental factors.

Collaborative pervasive applications (CPAs) mainly focus on collaborative interactions between actors such as users (with their devices). For example, sharing contents among nearby mobile users, collaboration among conference/workshop attendees, and collaboration among drivers in a cooperative convoy. Such applications raise major challenges in terms of their development and management, including the dynamic aspect of them and their environments. The use of services offers the possibility to design, deploy and manage these applications dynamically, providing the required flexibility.

Most of the approaches in the pervasive computing literature mainly focus on tasks of individual users, and provide very limited support to collaborative tasks of a group of actors. Furthermore, existing approaches in developing collaborative pervasive applications (e.g.,8,9,2,3,10), and middleware architectures and frameworks for pervasive computing11 are limited in supporting the dynamic relationships between actors, which themselves are an important aspect of context. In particular, there is a lack of support for managing the adaptation in such relationships in response to changes in the requirements and environments.

This research explores the concept of social context as a means to represent the relationships among actors and presents a platform to provide support for adaptation by managing such social contexts and their changes, in a service-oriented manner. Social context in computing is often used to refer to the people, groups and organizations that an individual interacts with12,13. Taking this view, we define social context as a representation of the interactions among the relevant actors. That is, social context defines the constructed relationships between social roles, and these relationships define and constrain the interactions between the actors playing those roles. To model social context, we employ Role-Oriented Adaptive Design (ROAD) among many different approaches to design role-based software.
systems using the agent paradigm (see section 8.1 for a detail discussion), as ROAD brings a number of design principles that support flexible management and runtime adaptations. One of the key principles of ROAD is the separation of functional and management operations. This principle is important for facilitating application development and runtime adaptation as the functional operations performed by the actors enable normal interactions while the management operations realize adaptation to the structure and parameters of the system. Thus, we adopt and extend the ROAD architectural style for modeling and managing social context. We model social context from two perspectives – domain-centric and player-centric – by specializing the ROAD composite structure \(^{16}\). A domain-centric social context (DCSC) model captures a collaborative view of the interaction relationships among the actors whereas a player-centric social context (PCSC) model captures an actor’s coordinated view of all its interactions (across different domains). These two perspectives of social context are inherently related to each other, i.e., a PCSC can be derived from its respective DCSC(s) and a DCSC also can be derived from all its corresponding PCSCs.

In this paper, we present a SCaaS platform for developing collaborative pervasive applications from their high-level specifications (represented in terms of DCSCs and PCSCs), and for managing their adaptations to cope with runtime changes. Figure 1 presents an overview of the SCaaS platform. SCaaS takes the DCSC and PCSC models (specified by an application designer) as inputs, and instantiates these models by generating management and interaction interfaces (as Web Services) for applications to use and invoke. Thus, using applications (running on mobile devices) actors can interact with each other and manage their social contexts. In particular, we focus on how the SCaaS platform supports runtime adaptation in social

![Figure 1. Overview of the SCaaS platform](image-url)
contexts to cope with changes in the evolving user requirements and environmental factors. In relation to the SCaaS platform, this paper makes the following four major contributions:

1. we identify different types of changes and the various adaptations to cope with such changes, and we propose management operations and present possible ways to perform adaptations using these operations,
2. we introduce states of the social context and its elements to enable adaptation in a safe manner,
3. we propose a protocol for adaptation propagation from DCSCs to PCSCs, and
4. finally, we implement a SCaaS platform for creating the execution environment of social contexts and managing their runtime adaptations, and quantify the platform’s adaptation overhead.

The paper is organized as follows. Section 2 presents a scenario where applications need to interact, collaborate and adapt in pervasive environments. After giving an overview of our social context models in section 3, we present the SCaaS platform for managing adaptations in social contexts in section 4. Section 5 discusses its prototype implementation, while section 6 presents a case study by revisiting the application scenario. Section 7 presents the experimental evaluation. After reviewing related research in section 8, we conclude the paper in section 9.

2. Application Scenario

In this section, we present an application scenario from the automotive domain, called cooperative convoy, where SCaaS supports adaptation. However, the SCaaS is generic and not limited to the cooperative convoy application but can be used to manage adaptation of social contexts in other applications.

Consider that two groups of tourists in two cars hired from two different rental companies want to drive together from Melbourne to Sydney. The car rental companies provide different types of support to their customers based on the customers’ insurance policies, service availability, and so on. Let us assume that the two cars, Car#1 and Car#2, are rented from Budget and AVIS respectively.

In a cooperative convoy, a vehicle interacts with other vehicles, service providers and infrastructure systems to make the travel safe and convenient. Through these interactions a vehicle can share information (acquired from the service providers and infrastructure systems) with other vehicles. Such interactions are subject to defined agreements and constraints among the entities (i.e., vehicle to vehicle, vehicle to service providers, and vehicle to infrastructure). For instance, drivers of Car#1 and Car#2 want to form a cooperative convoy to make their travel safe and convenient by collaborating and interacting with each other. These two cars have access to different types of services: Car#1 has access to a Travel Guide Service (TGS) while Car#2 has access to a real time Traffic Management Service (TMS). In the cooperative convoy, they decide that Car#1 is the leading car (LC) whilst
Car#2 is the following car (FC). The two cars follow the same route chosen by the leading car as it has access to a TGS. In addition, the drivers of both cars agree on a number of issues. For instance, they will always keep the distance between them less than 1000m. Car#2 (the following car) will send road blocks information (obtained from its TMS) to Car#1 (the leading car) if there is any. Car#1 gets the updated route plan from its TGS by specifying its preferences (e.g., avoid the route with blocked road) and notifies that route information to Car#2. Both cars notify each other of their positions every 10 seconds. If either vehicle experiences mechanical problems (e.g., flat tyre, engine issue) it needs to notify the other vehicle.

Applications facilitating the cooperative convoy need to fulfill two major requirements. First, the applications should support interactions complying with the agreed interaction relationships (i.e., constraints and obligations). For the drivers to perform the additional tasks (e.g., forwarding information) may cause distraction and have undesirable consequences. Thus, to facilitate collaboration with less distraction, the applications need to provide a coordinated view of the interactions, allow drivers to specify their coordination preferences and perform the coordination in an automated manner. Second, the applications need to support runtime adaptation as the interaction relationships evolve over time, and need to adapt with the changes in requirements and environments. For instance, a mechanical problem of the leading car may require it to handover the leading car role to one of the following cars (assuming there are multiple following cars). Because of heavy rain, the maximum distance may need to be reduced from 1000m to 600m. A third vehicle could join when the convoy is on the way; or the break-down of a following vehicle might result in its leaving the convoy before reaching the destination.

To address the first requirement, in our previous work\textsuperscript{16}, we have proposed an approach to modeling interaction relationships from both the domain and player perspectives. The DCSC model allows the interactions associated with a domain such as Budget or AVIS or Cooperative Convoy to be captured, while the PCSC model provides an overall view of all the interactions of a particular individual (e.g., driver of Car#1) in all relevant domains and allows coordination among its interactions across these domains.

To address the second requirement, in this paper, we propose the SCaaS platform where the DCSC and PCSC are the basis of this platform. At runtime, the interaction relationships captured by DCSCs need to adapt with the changes in user requirements and environments. However, the PCSCs are dependent on DCSCs. The SCaaS platform manages both the DCSCs and PCSCs, and their dependencies in a consistent manner by supporting the adaptations in the DCSCs and the adaptation propagation from DCSCs to PCSCs as changes occur.


In this section, we briefly discuss the DCSC and PCSC models, and illustrate how these social context modeling perspectives allow us to capture the above cooperative
A detailed discussion can be found in [17].

3.1. Domain-Centric Social Context Models

The Domain-Centric Social Context (DCSC) model captures the relationships among social roles associated with a particular domain or environment such as a company, a cooperative convoy, and so on. A DCSC model comprises of four key elements: social role, relationship, player, and organizer role. A social role represents the expected functional interactions of a participating actor with respect to the social context. Social roles are loosely-coupled elements and are modeled as first class entities, and as such they are separated from their players (e.g., actors) who play those roles. A relationship is an association between two social roles, which represents the interactions and interdependencies between those roles (or their corresponding players or actors). It mediates the interactions between social roles by defining what functional interactions can occur between the social roles and the sequences of these interactions (namely conversations). In addition, a relationship also defines the non-functional requirements of the interactions in terms of opera-
tional parameters and obligations (e.g., time constraints on interactions) imposed on the players associated with that relationship. The organizer role and its player provide the capability for managing and adapting a social context to cope with changes in the requirements and dynamic environments. While social roles, players and relationships are entities at the social context’s functional layer, the organizer role and its player are entities at the social context’s management layer.

In the above scenario, there are three domains that need to be modeled, namely, the two car rental companies (Budget and AVIS) and the cooperative convoy. According to the agreements between the drivers of the two cars, the ConvoyDCSC model (see Fig. 2a) consists of two roles: LeadingCar (LC) and FollowingCar (FC), and their interactions are captured in the R4 (LC-FC) relationship (see Table 1) where an interaction is represented using a message signature, a direction of the message (e.g., LCtoFC, FCtoLC or bidirectional), and an optional return message (e.g., ack). Table 1 presents a partial description of R4 relationship between the LeadingCar and FollowingCar social roles in ConvoyDCSC. The R4 relationship includes five interactions (i.e., i7, i8, i9, i10 and i11), one obligation (i.e., o1) and one operational parameter (i.e., p1). Also, there may be other constraints and behavioral properties (e.g., obligation, operational parameter, conversation - the sequence of interactions), but for simplicity we do not include all of them. The i9 is a bidirectional positionUpdate interaction, i.e., both leading and following car should notify each other their positions. The time constraint related to this interaction is captured in o1 obligation, i.e., the time period of this position update frequency is 10 seconds. The i8 is a unidirectional (i.e., LCtoFC - leading car to following car) routeUpdate interaction with a return acknowledgment message. Figure 3 shows an XML representation of the i8 interaction.

<table>
<thead>
<tr>
<th>Specification from scenario</th>
<th>Notational representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC sends ahead road blocks information to the LC</td>
<td>i7: {notifyRoadBlock, FCtoLC, ack}</td>
</tr>
<tr>
<td>LC updates the route information to the FC</td>
<td>i8: {routeUpdate, LCtoFC, ack}</td>
</tr>
<tr>
<td>Both cars notify each other of their positions every 10 seconds</td>
<td>i9: {positionUpdate} // bi-directional o1: {i9, Time, periodic, =, 10, seconds}</td>
</tr>
<tr>
<td>One car notifies mechanical problems to the other car</td>
<td>i10: {notifyMechanicalIssue}</td>
</tr>
<tr>
<td>Either vehicle may leave the convoy</td>
<td>i11: {leaveConvoy}</td>
</tr>
<tr>
<td>Maximum distance between the LC and FC is 1000m</td>
<td>p1: {maxDistance = 1000m}</td>
</tr>
</tbody>
</table>

As Car#1 plays the leading car role, it is also designated to play the organizer...
Fig. 3. XML representation of the i8 social interaction

role of the ConvoyDCSC model. So, by playing the organizer role (through an application interface), the leading car driver can change the model (e.g., add, delete or update roles and relationships) at runtime. In a similar way, we can also model BudgetDCSC and AVISDCSC where Car#1 and Car#2 play the RentedCar role.

3.2. Player-Centric Social Context Models

In addition to the common view of a domain-centric social context model, an actor may have its own perception or view of the domain with respect to the role(s) it plays and the interactions it participates in that domain. Moreover, an actor may operate in different domains. Thus, we also model the social context from an actor’s perspective, namely Player-Centric Social Context (PCSC). The PCSC model provides an overall view of all the interactions of an individual (across different domains) and allows coordination of its interactions.

Fig. 2b and Fig. 2c show the player-centric models of Car#1 and Car#2 respectively (and how they relate to the domain-centric models). Like a DCSC, a PCSC contains social roles, actors/players and an organizer role. In addition, it contains a coordinator role and role-centric relationships. In the PCSC, all social roles are played by the actor (application) in question and are connected with the coordinator role through the role-centric relationships. The coordinator role provides a means of achieving inter-domain coordination, i.e., interactions in one domain can be used for interactions in another domain. An actor can coordinate its interactions explicitly through the application’s user interface or it may define some rules or use an intelligent application to coordinate its interactions on its behalf. A role-centric relationship is the aggregation of all the relationships associated with a particular social role in a DCSC model, but localized in the player-centric model. For example, the Rv2 role-centric relationship in the PCSC of Car#1 is the aggregation of R1, R2 and R3 in the BudgetDCSC.
Social Context as a Service (SCaaS) Platform

4. Social Context as a Service Platform

4.1. Social Context at Runtime

Social contexts are not just a modeling or design-time construct, and they are also runtime entities that mediate runtime interactions between actors. Interactions among an actor, its player-centric social context model, and the relevant domain-centric social context models are loosely coupled and use a messaging style. A (runtime) social context acts as a message router (see Fig. 4) that (1) receives messages from an actor, (2) evaluates conditions specified in associated relationships, and (3) passes the messages to another actor or a social context (as a player) or notifies the actor(s) in case of any condition violation. For the PCSC, all the incoming and outgoing messages are intercepted by the coordinator role which is played by the actor’s coordination application. The application coordinates messages on behalf of the actor based on her preferences. For instance, all the player- and domain-centric social context models such as Car#1PCSC, Car#2PCSC, ConvoyDCSC, BudgetDCSC and AVISDCSC are runtime entities and they mediate interactions among drivers of Car#1 and Car#2 and other respective entities. When the driver of Car#1 sends through her application (as an actor/player) a message to the driver of Car#2, the message first goes to Car#1’s player-centric model, i.e., Car#1PCSC, and then the message is routed to the respective domain-centric model, i.e., ConvoyDCSC. After checking constraints specified in the $R_4$ relationship the message is forwarded to the Car#2’s player-centric model, i.e., Car#2PCSC, and finally to the Car#2’s driver application.

SCaaS facilitates the runtime realization of social contexts for supporting mediated interactions between collaborative actors, such as those of Car#1 and Car#2. However, the requirements of such actors or applications are subject to continu-
ous change. Thus, social contexts need to be managed and adapted to ensure the proper functioning and evolution of the applications in which the social contexts play a part. In the rest of this section, we discuss the management and adaptation support provided by the SCaaS platform in offering social context as a service.

4.2. Types of Changes

In general, there are two types of changes that require adaptation in a social context as well as across social contexts.

Changes in Environments. During convoy, it may start to rain heavily or the cars may move from one jurisdiction to another which operates a different traffic management system. Such changes cause social context adaptation and are referred to as changes in environments.

Changes in Requirements. A third vehicle could join when the convoy is already on the way; a broken-down following car might leave the convoy before reaching its destination; or the leading car might have a mechanical problem which requires to handover the leading car role to one of the following cars (assuming multiple following cars). Such situations also cause adaptation and are referred to as changes in requirements.

4.3. Adaptations in a Social Context

Runtime adaptation, often called dynamic adaptation, is a widely used term and is extensively studied in multiple disciplines. In pervasive computing, this term is used to denote any kind of modification at the running phase of the system. In general, such runtime adaptation can be classified into two categories: structure (compositional) adaptation and parameter adaptation. The adaptation in a social context to cope with the changes in user requirements and environments also can be of these two types: structural (compositional) and parametric.

The organizer role provides the management capability of a social context. It exposes a management interface that contains methods for manipulating the structure and parameter of the runtime social context model. In the following subsections these aspects are discussed in details.

4.3.1. Structural Adaptation

We achieve structural adaptation in two ways:

- Modifying topology – Social roles, players and relationships are added or removed. For instance, a third vehicle could join the convoy when it is already on the way, or a broken-down following car leaves the convoy before reaching the destination. These situations lead to the addition or removal of roles, players and relationships in the ConvoyDCSC.
- Modifying the binding between a social role and its player – The same social role can be played by different players at different times. The binding between
the role and the players is dynamic. For instance, in the AVISDCSC, the traffic management role can be played by different traffic management systems in the convoy at different times as the vehicle moves from one jurisdiction to another. Also due to a mechanical problem of the leading car (Car#1), the Car#1PCSC needs to unbind from the leading car role in the ConvoyDCSC and one of the following cars can be assigned to play the leading car role by binding that car’s PCSC to the leading car role in the ConvoyDCSC.

Table 2 presents the management operations provided by the organizer role to modify topology, i.e., add or delete social roles or relationships; and to modify bindings, i.e., binds or unbinds a player to or from a social role.

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifying topology</td>
<td>addSocialRole</td>
<td>Adds a social role to a runtime model</td>
</tr>
<tr>
<td></td>
<td>delSocialRole</td>
<td>Deletes a social role from a model</td>
</tr>
<tr>
<td></td>
<td>addSRelationship</td>
<td>Adds a social relationship between two roles</td>
</tr>
<tr>
<td></td>
<td>delSRelationship</td>
<td>Deletes a social relationship between two social roles</td>
</tr>
<tr>
<td>Modifying bindings</td>
<td>bindRP</td>
<td>Binds a player to a role</td>
</tr>
<tr>
<td></td>
<td>unbindRP</td>
<td>Unbinds a player from a role</td>
</tr>
</tbody>
</table>

4.3.2. Parametric Adaptation

We achieve parametric adaptation by modifying relationships where interactions, obligations, conversations and operational parameters can be added, removed or updated. For instance, in the ConvoyDCSC, the maxDistance parameter value in relationship R4 can be reduced from 1000m to 600m because of heavy rain. Table 3 presents the list of management operations provided by the organizer role to manipulate the parameters of the runtime model.

4.3.3. Social Context States and Safe Change

To perform adaptation in a safe manner without affecting the message flow and loss of messages, we maintain the state of each entity of the runtime model, i.e., social role, relationship and runtime social context as a whole. Figure 5 shows the states and their transitions. When a social context is deployed all of its entities enter into the Idle state. When a conversation starts, the associated social roles and relationship move to the Active state and remain there until the conversation...
Table 3. Operations provided by the organizer role to manipulate the parameters of the runtime model

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifying</td>
<td>addInteraction</td>
<td>Adds an interaction to a relationship</td>
</tr>
<tr>
<td>relationships</td>
<td>addObligation</td>
<td>Adds an obligation to a relationship</td>
</tr>
<tr>
<td></td>
<td>addConversation</td>
<td>Adds a conversation to a relationship</td>
</tr>
<tr>
<td></td>
<td>addOpParameter</td>
<td>Adds an operational parameter in a relationship</td>
</tr>
<tr>
<td></td>
<td>delInteraction</td>
<td>Deletes an interaction from a relationship</td>
</tr>
<tr>
<td></td>
<td>delObligation</td>
<td>Deletes an obligation from a relationship</td>
</tr>
<tr>
<td></td>
<td>delConversation</td>
<td>Deletes a conversation from a relationship</td>
</tr>
<tr>
<td></td>
<td>delOpParameter</td>
<td>Deletes an operational parameter from a relationship</td>
</tr>
<tr>
<td></td>
<td>updateInteraction</td>
<td>Updates an interaction in a relationship</td>
</tr>
<tr>
<td></td>
<td>updateObligation</td>
<td>Updates an obligation in a relationship</td>
</tr>
<tr>
<td></td>
<td>updateConversation</td>
<td>Updates a conversation in a relationship</td>
</tr>
<tr>
<td></td>
<td>updateOpParameter</td>
<td>Updates an operational parameter value in a relationship</td>
</tr>
</tbody>
</table>

![State machine for Social Role, Social Relationship and Social Context](image)

For Social Role and Relationship, \( x \) = started conversation, \( y \) = completed conversation
For Social Context, \( x \) = exist at least one active role or relationship, \( y \) = no more active role and relationship

Fig. 5. State machine for Social Role, Social Relationship and Social Context

completes. A social context enters into the Active state when any of its social roles or relationships becomes Active and remains there until all of its roles and relationships become Idle. When an adaptation operation (structural or parametric) starts, the entity enters into the Reconfiguration state.

The time when a change cannot be made is the period when the entity is in Active state. For instance, an adaptation operation cannot be performed in a social role or relationship when a conversation (request/response) associated with these entities is in progress. In that case, the adaptation request will be buffered and executed in the future after the entities enter into the Idle state.
Table 4 presents the list of operations provided by the organizer role to manipulate the state of the runtime model.

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting state</td>
<td>setRoleState</td>
<td>Sets a state to a social role</td>
</tr>
<tr>
<td></td>
<td>setRelState</td>
<td>Sets a state to a social relationship</td>
</tr>
<tr>
<td></td>
<td>setSCState</td>
<td>Sets a state to a runtime social context model</td>
</tr>
</tbody>
</table>

4.3.4. Monitoring Runtime Model and Performing Adaptation

The principle of separating a role from its player is also applied to the organizer role. The organizer role presents management rights over a runtime social context model, for example, to an actor/player who owns the runtime model (depicted in Fig. 6).

The organizer role also provides methods for acquiring information related to the
structure, parameter and state of the runtime social context model (see Table 5), and subsequently allows the organizer player to monitor the runtime model and perform adaptation safely. For example, before deleting or updating a relationship, the organizer player can acquire the current state of the relationship using the getRelState method to check whether the relationship is in the safe state (i.e., Idle) to perform adaptation.

Table 5. Operations provided by the organizer role to monitor a runtime model by acquiring information

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure related</td>
<td>getRoleList</td>
<td>Returns a list of role names in a social context model</td>
</tr>
<tr>
<td></td>
<td>getRelationshipList</td>
<td>Returns a list of relationship names in a social context model</td>
</tr>
<tr>
<td></td>
<td>getAllRel4Role</td>
<td>Returns a list of relationship associated with a role</td>
</tr>
<tr>
<td></td>
<td>getRelationship</td>
<td>Returns a relationship name between two roles</td>
</tr>
<tr>
<td>Parameter related</td>
<td>getInteractionList</td>
<td>Returns a list of interactions in a relationship</td>
</tr>
<tr>
<td></td>
<td>getConversationList</td>
<td>Returns a list of conversations related to an interaction</td>
</tr>
<tr>
<td></td>
<td>getObligationList</td>
<td>Returns a list of obligations related to a conversation or an interaction</td>
</tr>
<tr>
<td></td>
<td>getOpParameter</td>
<td>Returns an operational parameter related to an obligation</td>
</tr>
<tr>
<td>State related</td>
<td>getRoleState</td>
<td>Returns the state of a social role</td>
</tr>
<tr>
<td></td>
<td>getRelState</td>
<td>Returns the state of a social relationship</td>
</tr>
<tr>
<td></td>
<td>getSCState</td>
<td>Returns the state of a runtime social context model</td>
</tr>
</tbody>
</table>

The organizer role is internal to a social context and allows its player to manage both the structure and parameters of the social context. By playing the organizer role, a human can perform adaptation manually using a graphical interface. On the other hand, automatic adaptation can be defined and performed through a computer program or a software agent or a set of predefined adaptation rules as a player of the organizer. For example, the maxDistance parameter value can be reset to 600m using the following Event-Condition-Action (ECA) rule:
adaptation-rule "Update maximum distance"
when
   EnvironmentChangeEvent(name=="RainingStatusValueChanged") //Event
then
   if(rainingStatus == HEAVY_RAIN) //Condition
      callMethodInOrgInterface("updateOpParameter("R4",maxDistance,600m") //Action

4.4. Adaptations across Social Contexts

As stated in the previous section, a PCSC provides a coordinated view of all the interactions of an individual across different domains, and an individual plays roles in multiple domains/DCSCs through her PCSC. Thus, an adaptation in a DCSC should be propagated to its corresponding PCSCs. Figure 7 shows a basic protocol for such propagation. In this protocol, the DCSC organizer triggers the adaptation in a PCSC by invoking the following methods: triggerRoleAcquisition, triggerRoleRelinquishment and triggerUpdateRelationship.

The adaptation across social context models has three aspects:

1) Binding a player to a social role in a DCSC — When a player binds to a role in a DCSC, her PCSC should add that role and its role-centric relationship. Thus, for the bindRP request, the DCSC invokes computeRoleCentricRel method to compute the role-centric relationship of a particular social role which is the

Fig. 7. Cross-DCSC/PCSC adaptation propagation
aggregation of all the relationships associated with that social role in the DCSC (see Algorithm 4.1). Then the DCSC invokes the *triggerRoleAcquisition* method in the PCSC organizer with the social role and its role-centric relationship, as parameters. The PCSC organizer executes the *roleAcquisition* method to adapt its structure by adding a social role and relationship based on the received information.

**Algorithm 4.1** Computing Role-centric Relationship

```plaintext
1: procedure COMPUTEROLECENTRICREL(sc, r) \(\triangleright \) r is a social role and sc is a social context
2:   roleCentricRel ← empty \(\triangleright \) create an empty relationship
3:   relList ← getAllRel4Role(sc, r) \(\triangleright \) relationships connected to r
4:   for all rel ∈ relList do \(\triangleright \) rel is a relationship in the relList
5:     for all i ∈ rel do \(\triangleright \) i is an interaction in rel
6:       roleCentricRel.addInteraction(i)
7:     end for
8:   for all c ∈ rel do \(\triangleright \) c is a conversation in rel
9:     roleCentricRel.addConversation(c)
10: end for
11: for all o ∈ rel do \(\triangleright \) o is an obligation in rel
12:   roleCentricRel.addObligation(o)
13: end for
14: for all p ∈ rel do \(\triangleright \) p is an operational parameter in rel
15:   roleCentricRel.addOpParameter(p)
16: end for
17: end for
18: return roleCentricRel
19: end procedure
```

The *roleAcquisition* method (see Algorithm 4.2) first adds a social role based on the received information. After that it checks the state of the coordinator role using the *getRoleState* management method. When the coordinator role enters into the *Idle* state which is the safe state to perform adaptation, the state of the coordinator role is set to the *Reconfiguration* state. In this state, the coordinator role does not accept any incoming message and thus, it is safe to perform adaptation. A relationship between the added social role and the coordinator role is added based on the received information. Once the adaptation is completed, the state of the coordinator role is set back to the *Idle* state.

For instance, to add a new car, say Car#3, as a following car in the *Convoy-DCSC* (illustrated in Fig. 8), the *ConvoyDCSC* organizer player invokes (using organizer interface) the following management methods:

- *addSocialRole* ("FC2") – to add another following car role to the *Convoy-
Algorithm 4.2 Executing role acquisition
1: procedure roleAcquisition(r, rel, url) \(\triangleright\) r is a social role name, rel is a relationship, url is an endpoint URL of role r in DCSC
2: \hspace{0.5em} addSocialRole(r) \(\triangleright\) adding a social role
3: \hspace{0.5em} bindRP(r, url) \(\triangleright\) binding endpoint url to the social role r
4: \hspace{0.5em} crState \leftarrow getRoleState("CoordinatorRole")
5: \hspace{0.5em} \triangleright\ checking until CoordinatorRole enters into Idle state
6: \hspace{1em} while crState \(\neq\) Idle do
7: \hspace{1.5em} wait() \(\triangleright\) wait for some time
8: \hspace{1.5em} crState \leftarrow getRoleState("CoordinatorRole")
9: \hspace{1em} end while
10: \hspace{0.5em} setRoleState("CoordinatorRole", Reconfiguration) \(\triangleright\)
\hspace{0.5em} Setting CoordinatorRole state as Reconfiguration so that adaptation can be performed safely
11: \hspace{0.5em} addSRelationship(r, "CoordinatorRole", rel) \(\triangleright\) adding a relationship (rel) between roles r and CoordinatorRole
12: \hspace{0.5em} setRoleState("CoordinatorRole", Idle) \(\triangleright\) adaptation complete so set back the state to Idle
13: end procedure

DCSC,
- addSRelationship("LC", "FC2", relName) – to add a relationship between the LC and FC2 social roles,
- addInteraction, addConversation, addObligation and addOpParameter – to add functional and non-functional information to the relationship as required, and
- finally, bindRP – to bind the third vehicle to the FC2 role.

Fig. 8. Adaptation in the ConvoyDCSC and Car#3PCSC for adding Car#3 as a following car
Because the bindRP is invoked, the ConvoyDCSC (as part of the cross adaptation process) computes the role-centric relationship of the FC2 (say Rr5) using the Algorithm 4.1 and invokes the triggerRoleAcquisition method in Car#3PCSC with FC2 and Rr5 as parameters. As a result, the Car#3PCSC updates its structure by adding the FC2 role and the Rr5 role-centric relationship between the FC2 and Coordinator (Co) roles.

(2) Unbinding a player from a social role in a DCSC — When a player is unbound from the DCSC, her PCSC should be adapted by removing that role and its role-centric relationship. Thus, for the unbindRP request the DCSC invokes the triggerRoleRelinquishment method with the social role name as the parameter. The PCSC organizer executes the roleRelinquishment method to adapt its structure by removing the social role, and the relationship between that social role and the coordinator role, when those entities are in Idle state (i.e., safe to delete).

The roleRelinquishment method (see Algorithm 4.3) invokes the getRelationship management operation to get the relationship name between the target social role and the coordinator role. Then, it checks the state of the relationship (using the getRelState management method) and waits until it enters into the Idle state which is the safe state to perform adaptation. Once the relationship enters into the Idle state, the state of the relationship is set to Reconfiguration. After that the adaptation is performed by removing the relationship, unbinding player from the target social role, and finally removing the social role.

Algorithm 4.3 Executing role relinquishment

1: procedure ROLERELINQUISHMENT(r)  ▶ r is a social role name
2: relName ← getRelationship(r, “CoordinatorRole”)  ▶ getting the relationship name between the role r and CoordinatorRole
3: relState ← getRelState(relName)
4: ▶ checking until relationship relName enters into Idle state
5: while relState ≠ Idle do
6: wait()  ▶ wait for some time
7: relState ← getRelState(relName)
8: end while
9: delSRelationship(relName) ▶ deleting the relationship from the composite
10: unbindRP(r)  ▶ unbind the player from the role r
11: delSocialRole(r) ▶ delete the role r from the composite
12: end procedure

For instance, if the following car (Car#2) breaks down, the ConvoyDCSC organizer player invokes (using the organizer interface) the unbindRP method to unbind Car#2PCSC from the FC. As a consequence, the ConvoyDCSC (as part of the cross adaptation process) invokes the triggerRoleRelinquishment(“FC”)
method in Car#2PCSC. As a result, the Car#2PCSC updates its structure by removing the FC role and the relationship between the FC and Coordinator roles.

(3) Updating a social relationship in a DCSC — All the updates in a relationship and/or a social role in a DCSC should be propagated to the corresponding PCSC(s). Thus, for any modification request in a relationship (i.e., to add, delete or update an interaction, conversation or obligation), the DCSC organizer invokes the triggerUpdateRelationship method in the corresponding PCSCs. Then the PCSC organizer executes the updateRelationship method to reflect the changes.

The updateRelationship method (see Algorithm 4.4) invokes the getRelationship management operation to get the target relationship name which needs to be updated. After that, it checks the relationship state and waits until the relationship enters into the Idle state. When the relationship enters into the Idle state which is the safe state to perform adaptation, the state of the relationship is set to Reconfiguration. Based on the update request such as addInteraction, delConversation, updateOpParameter, and so on, the relationship is modified. Once the adaptation is completed, the state of the relationship is set back to the Idle state.

For instance, assume that during convoy it starts raining heavily, and as a consequence the adaptation rule defined in section 4.3.4 is fired or the ConvoyDCSC organizer player (using organizer interface) invokes the updateOpParameter(R4, maxDesiredDistance, 600m) method in ConvoyDCSC to change the maxDesiredDistance parameter value from 1000m to 600m. As the Car#2PCSC and Car#3PCSC are related to the R4 relationship, the ConvoyDCSC then invokes (as part of the cross adaptation process) the triggerUpdateRelationship method in Car#2PCSC and Car#3PCSC to update the maxDesiredDistance value in both of these PCSCs.
Algorithm 4.4 Updating Relationship

1: procedure UPDATERELATIONSHIP(r, updateType, data) \(\triangleright\) r is a social role name related to the relationship that needs to be updated, updateType is the type of updates in the relationship such as addInteraction, delConversation, and so on (any of the operations listed in Table 3 under category of modifying social relationship), data is the information that needs to be updated

2: relName \(\leftarrow\) getRelationship(r, “CoordinatorRole”) \(\triangleright\) getting the relationship name between role r and Coordinator Role

3: relState \(\leftarrow\) getRelState(relName)

4: while relState \(\neq\) Idle do

5: \hspace{1em}wait() \(\triangleright\) wait for some time

6: relState \(\leftarrow\) getRelState(relName)

7: end while

8: setRelState(relName, Reconfiguration) \(\triangleright\) setting relationship relName state to the Reconfiguration so that the adaptation can be performed safely

9: if updateType = “addInteraction” then

10: addInteraction(relName, data)

11: else if updateType = “addObligation” then

12: addObligation(relName, data)

13: else if updateType = “addConversation” then

14: addConversation(relName, data)

15: else if updateType = “delInteraction” then

16: delInteraction(relName, data)

17: else if updateType = “delObligation” then

18: delObligation(relName, data)

19: else if updateType = “delConversation” then

20: delConversation(relName, data)

21: else if updateType = “updateInteraction” then

22: updateInteraction(relName, data)

23: else if updateType = “updateConversation” then

24: updateConversation(relName, data)

25: else if updateType = “updateObligation” then

26: updateObligation(relName, data)

27: else if updateType = “addOpParameter” then

28: addOpParameter(relName, data)

29: else if updateType = “updateOpParameter” then

30: updateOpParameter(relName, data)

31: else if updateType = “delOpParameter” then

32: delOpParameter(relName, data)

33: end if

34: setRelState(relName, Idle) \(\triangleright\) adaptation complete so set back the state to Idle

35: end procedure
5. Prototype Implementation

5.1. Implementing the SCaaS Platform

We have implemented the SCaaS platform by adopting and extending ROAD4WS\(^a\) which is an extension to the Apache Axis2\(^b\) web service engine for deploying adaptive service compositions. ROAD4WS is an implementation of the Role-Oriented Adaptive Design (ROAD) framework\(^c\) that supports flexible management and runtime adaptations. A detail discussion of other role-based approaches and the reasons to adopt ROAD can be found in section 8.1. The SCaaS platform exploits JAXB 2.0\(^b\) for creating the DCSCs and PCSCs runtime from their XML descriptors. JAXB helps the generation of classes and interfaces of runtime models automatically using an XML schema. The platform exposes each social role as a service, the associated interactions of the role as operations of that service. The conversation and obligations specified in the social relationship are evaluated as event-condition-action rules and implemented using Drools\(^c\). Actors (players) playing the roles invoke the operations which create messages. Such messages are routed to other players who they are collaborating with.

As illustrated in Fig. 9, the SCaaS platform generates runtime social contexts which are able to (1) handle requests received from players (i.e., applications); (2) check security settings for authorized access; (3) allocate requests into a message queue; (4) forward messages to corresponding social roles; (5) evaluate conditions

---
\(^a\)http://axis.apache.org/
\(^b\)http://jcp.org/en/jsr/detail?id=22
\(^c\)http://www.jboss.org/drools/
specified in the relationships; (6) send requests to relevant players. A runtime social context also can (7) receive a management request (from a user/application) and adapt itself accordingly, and (8) propagate the adaptation to other social contexts as necessary. The SCaaS Management module handles (9) platform level management requests such as create, delete, deploy and undeploy social contexts as required by the user/application. Interactions between the runtime models and their players (i.e., external interactions) are supported by exchanging messages.

The runtime adaptations are supported by the Java reflection mechanism and the Drools engine. To cope with the changes in environments and requirements, at runtime, Javassist allows generation of new classes and modification of existing classes, which helps to add new social roles/relationships and change existing roles/relationships, respectively. The Drools engine allows SCaaS to inject new rules and delete existing rules from the working memory which facilitate the addition and deletion of conversations, obligations and parameters in the relationships. This way of implementation provides flexible and easy runtime adaptation in a particular entity of a social context without interrupting the other entities of that social context.

5.2. Implementing the SocioTelematics Application

To demonstrate the real-world applicability and feasibility of our approach, we have developed an adaptive collaborative application using the SCaaS platform, called SocioTelematics, that enables multiple cars to form cooperative convoys. We have implemented the SocioTelematics application for Android mobile devices and Tablets. This application allows the drivers to see each other’s positions on the Google Maps. Using this application, drivers in the convoy can adapt and manage their social contexts and interactions.

Fig. 10a shows the convoy panel of the application. The map of the screen shows the cars’ positions in the cooperative convoy. The application uses GPS to show its car position on the map and notifies that position to others (as the notifyPosition interaction). The application gets the other car’s position once it receives a notifyPosition message from the other car. Also it shows a warning message as pop-up, for example, if the distance between the two cars violates their agreed distance. The adaptation panel (see Fig. 10b) allows the leading car driver to perform adaptation on the ConvoyDCSC. The current version of the application provides an interface for the organizer player, e.g., the driver of the leading car, to perform/trigger two structural adaptation: join a car to the convoy and remove a car from the convoy. As required, options to trigger other types of structural adaptation can easily be incorporated to the interface.

The SCaaS platform makes it easy to develop this application based on the supposed interactions of the actors’ (i.e., drivers, rental companies, and other services)

\[\text{http://www.jboss.org/javassist}\]
\[\text{http://www.ict.swin.edu.au/personal/akabir/SocioTelematics/index.htm}\]
and without worrying about the underlying message exchanges and the evaluation of the messages, as the runtime support and adaptation of these social contexts and interactions are externalized to and managed by the SCaaS platform. Moreover, the runtime adaptation capability provided by the SCaaS platform allows the application to respond to changes in requirements and environmental factors, without direct change to the application code.

The SCaaS platform is particularly useful for developing and deploying collaborative pervasive applications where interactions among entities are based on pre-defined agreements and constraints. These SCaaS-based applications work better than using other means such as text messaging. With our platform, the interactions and sharing between entities, e.g., the convoy members, are automatic according to predefined agreements. It also checks the conformance of the agreements at run-
time (when actual interactions happen) and has the ability to adapt (i.e., modify) the agreements at runtime as required. On the other hand, text messaging requires the involvement of drivers, resulting in driver distractions with significant safety consequences (as often reported in the media recently).

6. Case Study

Let us consider a number of use cases that require adaptation in the ConvoyDCSC and the associated PCSCs. The organizer player of the ConvoyDCSC, here the driver of the leading car, triggers the adaptation by using the SocioTelematics application, and the adaptation request is executed and managed by the SCaaS platform. Fig. 11 illustrates the modifications of the runtime DCSC and PCSCs as adaptation is carried out.

- To add a new car, say Car#3, as a following car in the convoy, the leading car driver (the ConvoyDCSC organizer player) presses the respective button in the adaptation panel (Fig. 10b) of the SocioTelematics application that invokes the following management methods (see Fig. 11a): (1) addSocialRole (“FC2”)
– to add another following car role, namely FC2, to the *ConvoyDCSC*, (2) 
addSRelationship("LC", "FC2", relName) – to add a relationship between the 
LC and FC2 social roles, (3) addInteraction, addConversation, addObligation 
and addOpParameter – to add functional and non-functional information to 
the relationship as required, and finally (4) bindRP – to bind the third vehicle 
to the FC2 role. Because the bindRP is invoked, the *ConvoyDCSC* (as part 
of the cross adaptation process) computes the role-centric relationship of the 
FC2 (say Rr5) using the Algorithm 4.1 and invokes the triggerRoleAcquisition 
method in *Car#3PCSC* with FC2 and Rr5 as parameters. As a result, the 
*Car#3PCSC* updates its structure by adding the FC2 role and the Rr5 role-
centric relationship between the FC2 and Coordinator roles (see Fig. 11(b)). 
As a result of this sequence of adaptations, *Car#3PCSC* is linked with the 
*SocioTelematics* application the driver of Car#3 can perform interactions with other car drivers of the convoy through the 
*Car#3PCSC*.

- If the leading car (Car#1) breaks down, the *ConvoyDCSC* organizer player 
(the leading car driver) presses the respective button in the adaptation panel 
of the *SocioTelematics* application that invokes the unbindRP method to un-
bind *Car#1PCSC* from the LC. As a consequence, the *ConvoyDCSC* (as part 
of the cross adaptation process) invokes the triggerRoleRelinquishment("LC") 
method in *Car#1PCSC*. Then, the *Car#1PCSC* updates its structure by delet-
ing the LC role and the relationship between the LC and Coordinator roles (see 
Fig. 11(c)). Furthermore, to assign a following car (say Car#2) to play the lead-
ing car role, the leading car driver presses the respective button that invokes 
unbindRP("FC", urlCar#2PCSC) followed by bindRP("LC", urlCar#2PCSC) to 
first unbind *Car#2PCSC* from the following car role and then bind it to the 
leading car role. As a result, the *ConvoyDCSC* (as part of the cross adapta-

- The heavy rain situation can be detected by Car#2’s (LC) wet-sensor and 
an adaptation rule (defined in section 4.3) embedded in the leading car’s *So-
cioTelematics* application can trigger adaptation automatically or the driver 
of the leading car can trigger adaptation manually by pressing a button in 
the application. This adaptation request invokes the updateOpParameter(R4,maxDesiredDistance,600m) method in *ConvoyDCSC* to change the 
maxDesiredDistance parameter value from 1km to 600m. Both the *Car#2PCSC* 
and *Car#3PCSC* are related to the R4 relationship. Thus, the *ConvoyDCSC*
(as part of the cross adaptation process) invokes the \texttt{triggerUpdateRelationship} method in \texttt{Car\#2PCSC} and \texttt{Car\#3PCSC} to update the \texttt{maxDesiredDistance} value in both of these PCSCs (see Fig. 11©).

7. Experimental Evaluation

The goal of our experiment is to quantify the SCaaS platform’s adaptation overhead. We installed the SCaaS platform on a machine with Core i3 2.2 GHz CPU, 8GB RAM and Windows 7 OS. We used Java 1.6, Drools 2, Tomcat 7.0.21 and Axis2 1.6.1 in this experiment. We also measured the application’s adaptation performance in real-life experiment to demonstrate the feasibility and applicability of the SCaaS-based application.

7.1. Quantifying SCaaS’s Adaptation Overhead

To quantify the adaptation overhead, we deployed 100 domain-centric social context models (DCSCs) where each DCSC consists of 10 social roles connected in a ring topology using 10 social relationships. Each relationship is comprised of 6 interactions, 6 conversations and 6 obligations. We created 10 PCSCs for 10 players where each player plays a role in each of the 100 DCSCs. Thus, each PCSC contained 100 social roles and 100 role-centric relationships. We executed each of the structural and parametric adaptation operations 1000 times over the 100 DCSCs. We measured the time from the moment the adaptation was requested, to the moment Axis2 updated the services. The box plots in Fig. 12 and Fig. 13 show the summary of the results where the horizontal line inside each of the boxes represents the median. The results show that the deletion operations (e.g., \texttt{delSocialRole}, \texttt{delSRelationship}, and \texttt{unbindRP}) take less time compared to the addition operations (i.e., \texttt{addSocialRole}, \texttt{addSRelationship}, and \texttt{bindRP}).

Fig. 12a and Fig. 12b present the time required to perform different structural and parametric adaptation in a social context model at runtime. The results show that the structural adaptation takes more time than the parametric adaptation. Among the structural adaptation operations, adding a social relationship (\texttt{addSRelationship}) in the model takes the longest time, around 68 millisecond (ms) (on average), as it needs to update the configuration of two social roles, whereas deleting a social role (\texttt{delSocialRole}) takes the least time, around 5 milliseconds. For different parametric adaptation operations, the required time is related to rule injections and deletions in the Drools engine and lies between 162 and 486 microseconds.

The box plots in Fig. 13 illustrate the adaptation overhead results across a DCSC and a PCSC. Fig. 13a shows the total time required for the \texttt{bindRP} (bind role-player) and \texttt{unbindRP} (unbind role-player) structural adaptation. Fig. 13b shows the time required for each step in the \texttt{bindRP} adaptation, including the time to add a URL to a social role (\texttt{addURLtoSR}), to compute a role centric relationship (\texttt{compRoleCenRel}) using Algorithm 4.1, to send a request to a PCSC (\texttt{sendReqToPCSC}), and to execute the role acquisition method (\texttt{exeRoleAcq}). Fig. 13c shows the time
required for each step in the unbindRP adaptation, including the time to delete an URL from a social role (delUR4SR), to send a request to a PCSC (sendReqToPCSC), and to execute the role relinquishment method (exeRoleRel). The results show that on average the bindRP and unbindRP operations take 100ms and 33ms, respectively, which we believe is an acceptable overhead in collaborative applications.

### 7.2. Quantifying Application’s Performance

We also evaluated the application’s adaptation overhead using two cars in a cooperative convoy over 50 kilometres of driving where the SCaaS platform was deployed in Amazon EC2 and the two client SocioTelematics applications were running on two in-car Android Samsung Galaxy Tabs with 3G connections. The results in Table 6 show that given the 1.12 second communication latency between the application...
and the server, on average the time to add and remove a car to and from the convoy at runtime take 1.33 sec (i.e., 1.121+0.209) and 1.167 sec, respectively, which we believe are acceptable times in a cooperative convoy.

Table 6. Time required to perform adaptation operations

<table>
<thead>
<tr>
<th>Operations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send an adaptation request from the <em>SocioTelematics</em> application to the Amazon server over a 3G network</td>
<td>1.121 sec</td>
</tr>
<tr>
<td>Add a new car to the <em>ConvoyDCSC</em> at runtime, i.e., addRole, addRel and bindRP</td>
<td>0.209 sec</td>
</tr>
<tr>
<td>Remove a car from the <em>ConvoyDCSC</em> at runtime, i.e., unbindRP, delRel and delRole</td>
<td>0.046 sec</td>
</tr>
<tr>
<td>Change the $a_1$ operational parameter in $R_4$ relationship</td>
<td>0.422ms</td>
</tr>
</tbody>
</table>
8. Related Work

We review and discuss related work from various areas within the scope of this paper to analyse and position our work with respect to the existing literature. In particular, we group our discussion into three main streams: (1) modeling and managing runtime social interactions, (2) platform for collaborative pervasive applications and (3) middleware support for runtime adaptation.

8.1. Modeling and Managing Runtime Social Interactions

According to the sociologist Max Weber, “an action is social if the acting individual takes account of the behaviour of others and is thereby oriented in its course” \(^{21}\). Social interaction refers to the sequence of social actions between individuals (or groups) by considering their relationships.

Social interaction has been studied in multi-agent systems. In a multi-agent system, agents usually interact in order to achieve better goals of them or the system in which they exist. Kalenka et al. \(^{22}\) incorporate social concepts into computer systems level paradigm to enhance the agent’s interaction performance. Rather than defining social relationships, the work proposes multi-furious social interaction attitudes ranging from self interested to the purely altruistic.

Role-based design has been extensively researched in the last few decades and adopted in multiple disciplines (e.g., agent-oriented software engineering (AOSE), distributed systems (DS), object-oriented methodologies (OOM), and so on) as a key concept for modeling and managing interactions. In AOSE, many frameworks (e.g., BRAIN \(^{23}\), Gaia \(^{24}\), and so on) have been proposed to design role-based software systems using the agent paradigm. In their survey paper, Cabri et al. \(^{25}\) compare the strength of existing AOSE proposals based on some evaluation criteria and showed BRAIN is more expressive compared to other frameworks. The existing proposals in AOSE (including BRAIN), DS (e.g., \(^{26}\)) and OOM (e.g., \(^{27}\)), however, mainly consider up to the design phase of software engineering and give no indication of how these roles are to be realized. Even though some of the works (e.g., ROADMAP \(^{28}\)) further extended to consider the runtime/implementation phase by incorporating different interaction language such as AUML \(^{29}\), they do not consider roles as an implemented runtime entity and provide very limited support for managing runtime adaptation. In such approaches roles are used in modeling and to inform the design, but as entities disappear during implementation. Thus, they are not suitable for managing adaptations in runtime models.

Compared to the above mentioned frameworks, Role-Oriented Adaptive Design (ROAD) \(^{30}\) brings a number of design principles that support flexible management and runtime adaptations, which has also been recognized and appreciated by Ceruzzi and Zambonelli \(^{31}\). One of such principles is the separation of functional and management operations. This principle is important for facilitating application development and runtime adaptation as the functional operations performed by the actors enable normal interactions while the management operations realize adap-
8.2. Platform for Collaborative Pervasive Applications

The need for supporting collaboration in pervasive computing environments has emerged in recent years (e.g., 8, 10). Such research has focused on collaborative interactions between different types of actors such as user-user and device-device, for various purposes. The SAPERE 8 middleware exploits social network graph to establish collaboration for sharing data among spatially collocated user devices. CoCA 9 is an ontology-based context-aware service platform for sharing of computational resources among devices in a neighbourhood. Both SAPERE and CoCA focus on collaboration among devices, where SCaaS focuses on collaboration among users. Similar to SCaaS, MoCA 2, a middleware architecture for developing context-aware collaborative applications, focuses on collaboration among users. But unlike SCaaS, the collaborations among users in MoCA are not based on predefined goals or tasks, rather driven by spontaneous and occasional initiatives. CASMAS 3 and UseNet 10 focus on collaborative activities among users to achieve a common goal like SCaaS. But none of them explicitly model the interactions among users.

Moreover, all of the above approaches lack support for managing the dynamicity and complexity of the social context as highlighted in this paper. The relationships between actors and their adaptations are not modeled explicitly, and instead are often hard-coded directly into the applications. To the best of our knowledge, there is no work to date that addresses the runtime adaptation of social context models in response to the changes in requirements and environments.

8.3. Middleware Support for Runtime Adaptation

Much research has been carried out into middleware support for runtime adaptation in context-aware systems (e.g., MADAM 32 and 3PC 33) and service-oriented systems (e.g., MUSIC 34 and MOSES 35). These middleware solutions mainly target the tasks of individual users/applications and have focused on reconfiguring applications’ settings (rather than interaction relationships) based on physical context information (e.g., place, time)/quality of service requirements (e.g., performance, reliability), rather than interaction relationships. Moreover, their proposed runtime models are application-specific and cannot be used to model interaction-relationships among collaborative actors.

In contrast to these solutions, the SCaaS platform targets collaborative pervasive applications, and focuses on executing adaptation by explicitly realizing interaction relationships using social contexts and providing an organizer interface to change such social contexts. On the other hand, SCaaS does not address the monitoring of environment changes (i.e., physical context information), analyzing such information or making adaptation decisions. In that sense, the SCaaS middleware is not....
a substitute for existing middleware solutions that manages physical context information, rather can complement those solutions as appropriate, in order to manage (as a service) social interactions and context adaptation for collaborative pervasive applications.

9. Conclusion

We have presented a novel Social Context as a Service (SCaaS) platform for supporting application-level adaptations and enabling mediated-interactions among actors (individuals with their applications) in collaborative pervasive environments. Our approach externalizes interaction-relationships from the application implementation, explicitly models the interactions in terms of social contexts, separates functional interactions from management operations, and provides runtime realization and adaptation of social contexts. All these facilitate the systematic management of dynamic interaction-relationships between actors and support their adaptation to cope with the changes in user requirements and environments.

SCaaS facilitates both structural and parametric adaptations in social contexts which are realized through the management (organizer) interface of the social contexts. SCaaS also maintains the inherent dependencies among social contexts and keeps them consistent through coordinated cross-social context adaptation. Our model-driven approach and service-oriented implementation make it easier to develop different adaptive collaborative applications on top of SCaaS. We have quantified the adaptation overhead of the SCaaS platform through an experimental evaluation and demonstrated its applicability with a cooperative convoy telematics application.

As future work, we aim to study the development overhead (e.g., required time, line of code) of such SCaaS-based applications.

References


22. S. Kalenka and N. Jennings, Distinguishing social agent behaviour: a formal frame-


**Appendix - Extending CoopIS2013 paper**

This journal submission revises and extends our CoopIS2013 paper 14 by enhancing it in several directions:

1. providing adaptation operations in detail (section 4.3),
2. adding three algorithms related to adaptation across social context (section 4.4),
3. adding prototype implementation of SocioTelematics application (section 5),
4. discussing a number of use case scenarios as a case study (section 6), and
enhancing related work by discussing existing approaches to modeling and managing social interactions (section 8.1)