A Coordinated Spatio-Temporal Access Control Model for Mobile Computing in Coalition Environments

Song Fu and Cheng-Zhong Xu
Dept. of Electrical & Computer Engg.
Wayne State University, Detroit, MI 48202
{oaksong, czxu}@wayne.edu

Abstract

A primary concern in mobile computing is security. Mobile clients often relocate between different networks and connect to different data servers at different times. This poses new challenges to the resource access control in mobile computing. The resource sharing in a coalition environment creates certain temporal and spatial requirements for the accesses by mobile devices. However, there is a lack of formal treatment of the impact of mobility to the shared resource access control. In this paper, we introduce the shared resource access language, SRAL, to model the behavior of mobile devices. The language is structured and compositional so that programs of a mobile device can be constructed recursively from primitive accesses. We prove that SRAL is expressive enough for most resource access patterns. In particular, it is complete in the sense that it can specify any program of regular trace model. A constraint language is defined to specify spatial constraints for shared resource accesses. The problem of checking if a mobile object satisfies a given spatial constraint can be solved in a polynomial-time algorithm. We apply duration calculus to express temporal constraints, and show the temporal constraint satisfaction problem is decidable as well. We extend the role-based access control (RBAC) model to specify and enforce spatio-temporal constraints. This coordinated access control model has been implemented in a mobile agent system, which emulates mobile computing by software agents.

1 Introduction

The emergence of powerful portable computing devices, along with the advances in wireless communication technologies, has made mobile computing a reality. Mobile clients often frequently relocate between different networks and connect to different data servers at different times. The mobile computing environment no longer requires users to maintain a fixed and universally known position in the network and enables unrestricted mobility of the users. Mobility and portability are creating an entire new class of applications and new massive markets combining personal computing and consumer electronics.

Even as the size of mobile computing devices has been drastically reduced, their computing capacity has grown significantly. Today’s handheld devices have computing power equivalent to their desktop-computing counterparts of only one generation earlier. This phenomenon, while driving more and more functionality into handheld wireless Internet-enabled devices, is also driving security risks endemic to desktop computing into mobile devices. This holds not only for the wireless communication, but for the control of resource accesses as well. It is necessary and possible for the portable devices to roam across different networks. In addition, multiple mobile devices may participate in a teamwork. These features of mobile computing create new challenges to the resource access control, because permissions may be granted based not only on the requesting subject, but also on the previous access actions of the device and even of its companions. This requirement is frequently encountered in a coalition environment, where the servers are generally cooperative and trustworthy and they provide shared resources to the outside.

In a coalition environment, the computation of a mobile device may spread across several hosting sites and each access to the resources of a site lasts for a certain period of time. These spatio-temporal features pose a need for coordinated access control over shared resources at different sites. Examples of such requirements are “if a mobile device accesses a resource \( r \) (e.g. a licensed software package or its trial version) on site \( s_1 \) for too many times during a certain time period, it is not allowed to access the resource on site \( s_2 \) forever” and “the editing deadline for an issue of a daily newspaper is by 3am”. We note that these security requirements contain both temporal and spatial constraints over time-sensitive resource access activities in more than one site. They are beyond the capability of today’s any formal reasoning tools and access control models for mobile computing.

In this paper, we present a logical framework to support a coordinated access control model enforcing both temporal and spatial constraints related to mobile computing. The framework includes a Shared Resource Access Language (SRAL) for the specification of access patterns by a mobile device. The access language is structured and expressive enough to specify various access structures. SRAL is complete in the sense that it can specify any program of regular trace model. We note that role-based access control (RBAC) approaches [8] simplify the permission management in mobile computing, the indirect assignment of permissions to subjects and the permission inheritance in role hierarchies facilitate the privilege delegation and security policy making. The constraint specification and enforcement mechanisms are potent for designing the access control schemes for mobile systems. In [9], RBAC model was applied to trust man-
agement and credential distribution in the coalition environment. Our coordinated access control model extends the basic RBAC model to specify and enforce spatio-temporal constraints. A constraint language is defined to specify spatial constraints for shared resource accesses. The problem of checking if a mobile object program satisfies a given spatial constraint can be solved in a polynomial-time algorithm. We apply a continuous time model and boolean-valued state functions to express temporal constraints, and prove the temporal constraint satisfaction problem is decidable. We implemented this coordinated access control model in a mobile agent system, which emulates mobile computing by software agents.

The rest of the paper is organized as follows. Section 2 introduces the model for coalition mobile computing System and assumptions. Section 3 describe spatial constraints for coordinated mobile access control. The temporal constraints and satisfaction checking are presented in Section 4. Section 5 presents the implementation of the coordinated access control model in the Naplet mobile agent system. An application example for coordinated access control is discussed in 6. In Section 7, we present the related work. Section 8 concludes the paper with remarks on future work.

2 A Coalition Mobile Computing System Model and Assumptions

A mobile device may roam across networks to perform computation. At each site, the mobile component performs operations with access to resources of different types. Since private resources in a site can be accessed under local control, we focus on accesses to shared resources among the sites. A “coalition environment” could be commercial, in which corporations form a partnership, or governmental/military, in which several nations work together to achieve a common goal. In either case, the defining characteristic is the presence of multiple organizations or entities that are unwilling to rely on a third party to administer trust relationships. Consequently, the entities must cooperate to share the subset of their protected resources necessary to the coalition, while protecting the resources that they don’t want to share. To model a generic coalition environment, specify the coordinated access control scheme and reason its properties, we define the following syntactic sets:

- \( S \): a set of servers. Let \( s, s_1, s_2, \ldots \), range over \( S \).
- \( R \): a set of shared resources. Let \( r, r_1, r_2, \ldots \), range over \( R \).
- \( OP \): a set of operations on the shared resources \( R \). Let \( op, op_1, op_2, \ldots \), range over \( OP \), such as execute, read, write for the file system related applications.
- \( Z \): a set of channels. Let \( a, b, c, \ldots \), range over \( Z \).
- \( V \): a set of variables. Let \( x, y, z, \ldots \), range over \( V \).
- \( C \): a set of boolean expressions. Let \( c, c_1, c_2, \ldots \), range over \( C \).
- \( E \): a set of signals. Let \( e, e_1, e_2, \ldots \), range over \( E \).

A mobile device makes accesses to the shared resources and then moves to another server. It also needs to exercise input and output via the communication channels, synchronize its action with others in a teamwork. This execution behavior can be modelled by a mobile object \( o \) performing the same tasks among these coalition servers. Its program is composed of the equivalent instructions for shared resource accesses and synchronization operations by the mobile device.

An access is represented by a tuple \( a = (o, op, r, s) \), which means the mobile object \( o \) exercises operation \( op \) on resource \( r \) at server \( s \). And it is the consequence of executing an instruction of the mobile object’s program, and at the same time the corresponding permissions are granted by the access control mechanism. A set of all accesses by a mobile object can be defined as \( A = \{a, op, r, s\} \). We assume when an access request to a shared resource is executed by a coalition server, a execution proof will be issued to the mobile object. It records the information of \( (o, op, r, s) \) for the access, and the execution time. So the semantics of a execution proof, denoted by \( Pr_x(\cdot) \), is, for an access \( a \) to a shared resource by a mobile object, \( Pr_x(a) = true \) iff access \( a \) has been successfully carried out by server \( a/s \).

We note that there are two types of mobility. One is physical mobility, which concerns about roaming of physical devices. The other is logical mobility with the migration of executables, such as processes and agents [15]. In this paper, we discuss the access control for physical mobility. Since the movement of mobile devices can be emulated by the migration of software components, our modelling and discussion are based on the mobile objects. We also implemented our coordinated access control model in an emulating mobile agent system.

3 Spatial Constraints for Coordinated Mobile Access Control

In mobile computing, a mobile object roams across a network carrying out its computation on coalition servers according to its program. This motivates us to define a shared resource access language to specify the access trace that a mobile object will make during its execution, and to construct a logical framework to reason its safety properties.

3.1 Syntax of Shared Resource Access Language

Intuitively, the program of a mobile object specifies the accesses that the object tends to perform and the ordering. For example, a mobile object \( o \) might read resources \( r_1, \ldots, r_n \) in that order; or, it might read \( r_1 \) first and then, if \( x > 0 \) then write \( r_2 \) else write \( r_3 \). We argue that a shared resource access language and its semantics should be expressive enough so that various access patterns can be specified. The access language should also be structured and compositional so that a mobile object program can be constructed from primitive or other composite programs. This also facilitates the formalization of the semantics for the language.

Inspired by the constructs in the current programming languages such as C++ and Java, we define a shared resource access language called SRAL as follows.
**Definition 3.1 (Shared resource access language)** Accesses of a mobile object to the shared resources are defined as follows using the BNF notation:

\[ a ::= op r @ s \mid a?X \mid a!e \mid \text{signal}(\xi) \mid \text{wait}(\xi) \]

\[ \mid a_1 ; a_2 \mid \text{if } c \text{ then } a_1 \text{ else } a_2 \mid \text{while } c \text{ do } a \]

where the symbol “::=” should be read as “is defined as”, the symbol “!” as “or” and the symbol “@” as “at”. □

We explain each construct intuitively as follows. A single resource access is represented by the primitive \( op r @ s \) which denotes an operation \( op \) on a shared resource \( r \) at server \( s \). \( a?X \) will receive a value from channel \( a \) and assigns the value to variable \( X \). If channel \( a \) is empty, then the mobile object instance will wait over the channel. \( a!e \) will append the value of arithmetic expression \( e \) to channel \( a \). If \( e \) is originally empty, then in addition, it will wake up all the mobile object instances that wait on it. \( \text{signal}(\xi) \) and \( \text{wait}(\xi) \) will perform an order synchronization such that \( \text{signal}(\xi) \) has to be performed first before \( \text{wait}(\xi) \) can be performed. \( a_1 ; a_2 \) is the sequential composition of accesses \( a_1 \) and \( a_2 \). \( \text{if } c \text{ then } a_1 \text{ else } a_2 \) is a conditional composition of \( a_1 \) and \( a_2 \). In particular, if condition \( c \) evaluates to true, then \( a_1 \) will be carried out, otherwise, \( a_2 \) will be carried out. \( \text{while } c \text{ do } a \) will carry out access \( a \) whenever \( c \) is still true.

### 3.2 Expressiveness of SRAL

In this section, we study the expressiveness of SRAL in terms of the trace model. Imagine we are observing the execution of a particular program \( p \) by a mobile object \( o \), and we record the shared resource accesses that are performed by mobile object \( o \) and the order that this set of accesses are performed. When the execution terminates, we get a sequence of access operations, called a trace of program \( p \). Given a program \( p \) of a mobile object, it is natural to model \( p \) by \( \text{traces}(p) \) – the set of all traces that program \( p \) can perform. We call \( \text{traces}(p) \) as the trace model of \( p \). For example, \( \text{traces}(a_1 ; a_2) = \{ a_1 , a_2 \} \).

Given two traces \( t \) and \( v \), we define the following operators:

- **Concatenation:** the concatenation of \( t \) and \( v \) is the trace in which \( t \) is followed by \( v \) and it is denoted by \( t \cdot v \).

- **Interleaving:** the interleaving of \( t \) and \( v \) is the trace model that results from all possible interleavings of \( t \) and \( v \). It is denoted by \( t \triangleright t \cdot v \) and defined recursively as follows:

\[ t \triangleright t \cdot v = \{ t \} \]

\[ t \triangleright t \cdot v = \{ v \} \]

If neither \( t \) nor \( v \) is an empty trace, then \( t \triangleright t \cdot v = \{ \text{head}(t) \cdot x \mid x \in \text{tail}(t) \} \cup \{ \text{head}(v) \cdot x \mid x \in \text{tail}(v) \} \)

\[ \triangleright t \cdot v = \text{t Newly added content:}

\[ \text{where } \text{head}(y) \text{ is the first access action of } y \text{ and } \text{tail}(y) \text{ is the trace consisting of the rest of accesses.} \]

- **Kleene closure:** the Kleene closure of \( t \) is the trace model consisting of zero or more concatenations of \( v \). It is denoted by \( t^* \) and defined as follows:

\[ \epsilon \in t^* \]

\[ q \in t^* \]

By treating a trace model as a set, we can easily extend these operators to trace models. In summary, we define a set of rules needed to calculate the trace models for all possible SRAL programs.

**Definition 3.2 (Trace model of accesses)** The trace model of a mobile object program \( p \) can be characterized by the following rules, where \( p_1 \) and \( p_2 \) are the subprograms.

- \( \text{traces}(a) = \{ a > \} \) where \( a \) is a shared resource access.

- \( \text{traces}(p_1 ; p_2) = \text{traces}(p_1) \cdot \text{traces}(p_2) \).

- \( \text{traces}(\text{if } c \text{ then } p_1 \text{ else } p_2) = \text{traces}(p_1) \cup \text{traces}(p_2) \).

- \( \text{traces}(p_1 \parallel p_2) = \text{traces}(p_1) \# \text{traces}(p_2) \).

- \( \text{traces}(\text{while } c \text{ do } p) = \text{traces}(p)^* \). □

**Definition 3.3 (Regular trace model)** We define regular trace models over a set of accesses \( A \) as follows:

- \( \{ a \} \) is a regular trace model where \( a \in A \).

- If \( p_1 \) and \( p_2 \) are regular trace models, then \( p_1 \cup p_2, p_1 \cdot p_2, p_1^* \) are regular trace models.

- A regular trace model can be obtained only by applying the above two rules in a finite number of times. □

Theorem 3.1 shows that any regular trace model can be expressed by a SRAL program.

**Theorem 3.1 (Regular completeness)** For each regular trace model \( m \) over a set of accesses \( A \), there exists a program \( P \) such that \( \text{traces}(P) = m \).

**Proof:** We prove it by induction.

1. **Induction basis:** if \( m = \{ a > \} \) where \( a \in A \), then \( \text{traces}(a) = m \).

2. **Induction hypothesis:** assume two regular trace models \( T \) and \( V \), for which there exist programs \( P_T \) and \( P_V \) such that \( \text{traces}(P_T) = T \) and \( \text{traces}(P_V) = V \).

3. **Induction step:** We have

\[ \text{traces}(\text{if } c \text{ then } P_T \text{ else } P_V) = T \cup V \text{ for some condition } c. \]

\[ \text{traces}(P_T;P_V) = T \cdot V. \]

\[ \text{traces}(\text{while } c \text{ do } P_T) = T^*. \]

Therefore, given an arbitrary regular trace model \( m \) over \( A \), there exists a mobile object program \( P \) such that \( \text{traces}(P) = m \).

Although SRAL is expressive enough for most applications, there exist some traces which cannot be specified by SRAL. For example, the trace model of resource \( r_1 \) being accessed \( n \) times followed by resource \( r_2 \) being accessed \( n \) times is not a regular trace model, and cannot be specified by SRAL. However, in practice, this can be achieved in an ad hoc fashion based on the underlying language which is usually Turing-complete.
3.3 Spatial Constraints and Constraint Satisfaction Checking

One important aspect of a mobile computing system is the spatial constraints concerning the order of shared resource accesses, and the satisfaction checking problem of program associated with spatial constraints. In this section, we propose a shared resource access constraint language, \( SRAC \), which is based on workflow temporal constraint language \( CONSTR \) [4]. Spatial constraints are defined over mobile object access actions. The constraint specification language must be expressible so that most security requirements related to synchronized access to shared resources can be represented. Meanwhile, the language must be simple and intuitive so that the constraints can be reasoned easily. A permission grant requires constraint satisfaction checking at run-time right after a mobile object is authenticated and its role is activated. In the following, we define \( SRAC \) and show that the spatial access satisfaction checking problem is decidable.

**Definition 3.4 (Shared resource access constraint language)**

We define a formula of shared resource access constraint language \( SRAC \) has the following form:

\[
C ::= T \mid F \mid a \mid a_1 \otimes a_2 \mid \#(m, n, A) \mid C_1 \land C_2 \\
| C_1 \lor C_2 \mid \neg C
\]

We also define the implication connective as \( C_1 \rightarrow C_2 ::= \neg C_1 \lor C_2 \). \( \square \)

**Example 3.5** The following examples illustrate some possible constraint expressions and their meanings:

- \( a \): access \( a \) must be performed by the mobile object.
- \( a_1 \land a_2 \): both \( a_1 \) and \( a_2 \) must be performed (in any order).
- \( a_1 \lor a_2 \): at least one of \( a_1 \) and \( a_2 \) is performed.
- \( a_1 \otimes a_2 \): one must first perform \( a_1 \) and then perform \( a_2 \) (possibly make other resource accesses in between).
- \( a_1 \rightarrow a_2 \): if \( a_1 \) is performed, then \( a_2 \) must be performed (either before or after \( a_1 \)).
- \( \#(0, 5, \sigma_{RSW}(A)) \): a restricted software (RSW) package, either licensed or trial version, can not be accessed by more than 5 times, no matter where the mobile object is run. \( \sigma \) is a selection operation over set \( A \) and returns a subset of accesses that meet certain conditions.

To establish the notion of shared resource access satisfaction, we first introduce the notions of trace satisfaction and trace model satisfaction.

**Definition 3.6 (Trace satisfaction)** We define the satisfaction relationship \( \models \) between a trace \( t \) and a constraint expression \( C \) as follows by structural induction on \( C \). If \( t \models C \) does not hold, we denote it by \( t \not\models C \).

- \( t \models T \) and \( t \not\models F \).
- For \( a \in \mathcal{A} \), \( t \models a \) if and only if \( a \in t \) and \( Pr_c(a) = true \), i.e. the access \( a \) is stated by the mobile code program and \( a \) has been performed before indicated by a execution proof.

- For \( a_1, a_2 \in \mathcal{A} \), \( t \models a_1 \otimes a_2 \) if and only if there exist traces \( t_1 \) and \( t_2 \) such that \( t_1 \circ t_2 = t \) and \( Pr_c(a_1) = true \) and \( t_2 \models a_2 \) and \( Pr_c(a_2) = true \).
- \( t \models \#(m, n, A) \) if and only if \( m \leq \#(A) \leq n \), where operator \# gets the cardinality of a set.
- \( t \models C_1 \land C_2 \) if and only if \( t \models C_1 \) and \( t \models C_2 \).
- \( t \models C_1 \lor C_2 \) if and only if \( t \models C_1 \) or \( t \models C_2 \).
- \( t \models \neg C \) if and only if \( t \not\models C \). \( \square \)

We extend the notion of satisfaction to one between mobile object programs and constraint expressions via the shared resource access trace models.

**Definition 3.7 (Mobile object execution satisfaction)** Given a mobile object program \( P \) and a constraint expression \( C \), we say \( P \) satisfies \( C \) (denoted by \( P \models C \)) if \( traces(P) \models C \). \( \square \)

Note that given a program \( P \), set \( traces(P) \) might be infinite, e.g. \( P \) contains a loop construct, and to check if \( P \) satisfies a constraint expression \( C \), we have to check for each \( t \in traces(P) \) if \( t \models C \). This seems to be undecidable when \( traces(P) \) is infinite. The following theorem shows that the mobile object execution satisfaction checking can be performed in polynomial time.

**Theorem 3.2 (Mobile object execution satisfaction checking)**

Given a mobile object program \( P \) specified in SRAL and a constraint expression specified in \( SRAC \), the time complexity of checking if \( P \models C \) is \( O(m \times n) \), where \( m \) is the size of \( P \) and \( n \) is the size of \( C \).

The proof of this theorem follows a similar approach proposed in [14].

3.4 Spatial Constraints Enforcement in RBAC

The role-based access control (RBAC) [8] model is widely-used in system security. It consists of four basic components: a set of users (User), a set of roles (Role), a set of permissions (Permission) and a set of subjects (Subject). A user is a human being, e.g. the security officer, or a mobile object. A role represents a collection of permissions needed to perform a certain job function, and a permission is an access operation that can be exercised on objects in the system. A subject relates a user to possibly many roles. When a user logs in the system after authentication, he establishes some subject(s), by which he can request activation of some of the roles he is authorized to perform. A role becomes active only if it is enabled at the time of the request and the user requesting its activation is entitled to activate it at that moment. After the activation of a role, the corresponding user obtains the permissions associated with that role under the restrictions of related constraints.

In the preceding sections, we have defined the SRAC language to express the spatial constraints for the shared resource access control in mobile computing. We extend the RBAC model to incorporate SRAC as one of the constraint definition languages. The security officer of a system defines the specific constraints to meet the spatial requirements for a coalition environment. To enforce these constraints, we need to introduce
a new type of state to permissions: a permission is active iff its associated role is assigned to subject in a session and the related spatial constraints are satisfied. That is

\[
(\forall perm \in Permission)active(perm) = true \iff (\exists r \in Role, \exists s \in Subject) r \in AR(s) \land perm \in RP(r) \land check(P, C) = true
\]

(3.1)

where \( AR(\cdot) \) and \( RP(\cdot) \) are functions, which map from a subject to a set of its active roles and from a role to its assigned permissions respectively, in the RBAC model. \( check(P, C) \) determines whether the mobile object’s program \( P \) can satisfy the spatial constraint \( C \) associated with the permission \( perm \). This procedure is carried out by a spatial constraint checking module which applies the approach in Section 3.3 to analyze the traces and execution proofs of a mobile object. So, when a mobile object does not meet the spatial constraints of a permission for shared resource accesses in a coalition environment, that permission will not be active and is not granted consequently.

4 Temporal Constraints

Timing constraints are essential for controlling time-sensitive activities in various applications. For example, in a workflow management system tasks usually have timing constraints and need to be executed in certain order. The existing timing constraints in TRBAC [3] and GTRBAC [12] are based on intervals of periodic events that are imposed on roles. Each interval uses the discrete time model with explicit beginning and ending time points, indicating the time when a role becomes enabled or disabled. Since the periodic constraints are imposed on roles in TRBAC, a disabling event of a role would revoke all of its granted privileges. Considering the fact that different permissions authorized to a role often have different temporal constraints, more roles need to be defined in TRBAC. Moreover, because there is no global clock in distributed systems and the arrival time of a mobile object on a server is unpredictable, the interval timing models are not appropriate to characterize the time-sensitive activities of mobile computing on different servers. To tackle these problems, we consider a continuous time model in the definition of temporal constraints, and use time durations (i.e., intervals with no beginning and ending time points). Instead of associating periodic events with roles, we change the states of permissions according to the result of constraint checking so as to avoid complicating permission management.

We assume a continuous model of time, which is isomorphic to the set of real numbers (\( \mathbb{R} \)) with a total order relation \(<\). The entire life-cycle of a mobile object, i.e., its overall execution time spreading multiple servers in a coalition environment, constitute a time line for us to describe its behaviors via temporal logic [6].

To meet the temporal requirements in accessing the time-sensitive resources, each permission is associated with a validity duration, which specifies the length of time period when the permission can be granted to a subject. The validity duration can be any positive real number and even infinity which means the corresponding resource is time-insensitive. With respect to the validity of time duration, each permission can be in one of three different states for a mobile object: inactive, active-but-invalid, and valid. A permission is inactive if it is not assigned to any role of the requesting subject or it is assigned to a role of the subject but that role has not been activated in a session. An active permission becomes invalid when the validity duration of the permission expires.

As used in [11], we model the states of permissions by boolean-valued functions over time: Time \( \rightarrow \{0, 1\} \) where Time is the set of real numbers. The function reflecting whether a permission of a role is valid at certain time point is: valid, \( \in Permission \times Time \rightarrow \{0, 1\} \), where \( valid(perm, t) = 1 \) means that the permission \( perm \) of role \( r \) is valid at time \( t \), and \( valid(perm, t) = 0 \) means that permission is in an invalid state at time \( t \). Then, the duration for a permission in the valid state over a time interval is the accumulated time in which the state is present in the interval.

Let \([b, e]\) be an interval, i.e., \( b, e \in Time \) and \( e \geq b \). The duration of state valid for a permission \( perm \) of a role \( r \) over \([b, e]\) equals the integral

\[
\int_b^e valid(perm, t)dt.
\]

The temporal constraints specify that the duration when a permission remains in the valid state must be no more than that permission’s validity duration, i.e.,

\[
(\forall perm \in Permission)(\forall t \in R) valid(perm, t) = 1 \iff active(perm, t) = 1 \land \int_b^e valid(perm, u)du \leq dur(perm),
\]

(4.1)

where \( dur(\cdot) \) returns the specified validity duration associated with a permission, \( active(perm, t) \) is a variant of (3.1) by using a boolean-valued function similar to \( valid(perm, t) \). We can apply two control schemes by assigning different values to base time \( t_b \). Suppose a mobile object has visited servers \( s_1, \ldots, s_{i-1} \) in order before roaming to server \( s_i \). If \( t_b = t_i \) which denotes the arrival time of the mobile object at the current server \( s_i \), then the temporal constraint restricts the validity of a permission only on the server \( s_i \). On the other hand, if \( t_b = t_1 \) which is the arrival time at the first server \( s_1 \), i.e. the starting time of execution by the mobile object, then the temporal constraint is applicable to the object’s entire execution on different servers.

According to Theorem 3.2 and the decidability theorem of Duration Calculus [11], we obtain the following theorem.

**Theorem 4.1 (Permission validity checking)** Let constraint \( C \) = \( active(perm, t) \land \int b^e valid(perm, u)du \leq dur(perm) \) for the validity of a permission \( perm \) at time \( t \). \( P, [t_b, t] \vdash C \) iff Expression 4.1 is satisfied. Then Given a mobile object program \( P \) specified in SRAL, a constraint expression specified in SRAC and a time interval \([t_b, t]\), the problem of checking if \( P, [t_b, t] \vdash C \) is decidable.

This permission validity checking was implemented as a module in our extended RBAC system. It receives the specification of a mobile object’s program \( P \), the time interval \([t_b, t]\), and the index of a permission in question, as inputs. Then it calls the spatial
constraint checking module described in Section 3.4 to determine whether the related spatial constraints are satisfied. The temporal constraints will be checked by comparing the integral of valid state function with the validity duration of the permission. A boolean result is return as an indication of whether the spatio-temporal constraints are satisfied or not. If not satisfied, an event will be triggered to set validr(·, ·) to 0, indicating denial of the permission to the mobile object.

5 Implementation

Based on the above formal framework, a coordinated access control prototype has been implemented in Naplet mobile agent system [17]. The Naplet system is an experimental framework in support of Java-compliant mobile agents. It provides constructs for agent declaration, confined agent execution environments, and mechanisms for agent monitoring, control, and communication. It features a structured navigation facility, reliable agent communication mechanism, proportional-share resource management strategies, and secure interface between agents and Naplet servers. We use the Naplet system to emulate mobile computing. A extended role-based access control model has been realized in Naplet and it provides the specification and enforcement mechanisms for spatio-temporal constraints.

Naplet-based mobile distributed systems are built upon a first-class Naplet object. It is an abstract of agents, defining hooks for application-specific functions to be performed in different stages of its life cycle in each server and an itinerary for its way of travelling among the servers. We can use the agent itinerary to describe the roaming agenda of a mobile device, i.e. the list of servers to be visited and their ordering.

5.1 Role-based access control in Naplet

To realize role based access control in the Naplet system, we defined an agent as an authenticated subject representing the source of a request by taking advantage of the latest Java subject-based security features. It comprised of a set of principals, based on which access control is implemented.

We defined a set of additional permission types to control access rights of naplet subjects over restricted services on the naplet servers. They are NapletRuntimePermission, NapletServicePermission, and NapletSocketPermission. Role is an important component in the RBAC model. The Principals in Java provide a mechanism to gather permissions for a job function. We defined roles in the RBAC model based on Principals in the Naplet system. Currently, there are three types of Principals: Naplet-Principal, NaplentOwnerPrincipal, and NapletServerAdministrator. For the detailed syntax and semantics definitions of these permissions and roles, please refer to [18].

On arrival at a server, the naplet must be authenticated based on the certificate of its owner issued by an authority or via a priori registration. Upon the success of authentication, a naplet subject will be created and it has a principal of class Naplet-Principal. The naplet will then be able to perform operations allowed by the NapletPrincipal role. That is the naplet server delegates the naplet execution to the subject of the naplet itself. This process realizes the role activation in the Naplet system. The role-permission assignment is achieved by defining the Java policy files. The grant statements associate the permissions to principals.

5.2 Coordinated access control in Naplet

To realize the coordinated access control in Naplet, we extended the above role-based access control system by incorporating a construction of the agent programs and using the SecurityManager to enforce the spatio-temporal constraints.

The SRAL prototype has been implemented in recursively constructed resource access patterns. Its base is a Singleton pattern, comprising of a single shared resource access at a server guarded by a pre-condition. Over the set of access patterns, we define three composite operators: SeqPattern and ParPattern, and Loop, as defined in SRAL to recursively form resource accesses of regular trace models. The SeqPattern and ParPattern constructs implement a sequential accessing pattern (p1; p2) and a concurrent accessing pattern (p1 || p2), respectively. The Loop construct defines repetition of an access program until a pre-condition no longer holds.

To show the programmability of the constructs, we give an example of access patterns recursively constructed from accesses and conditional accesses. Consider a mobile application over n servers s1, ..., sn. The following class ApplAgentProg defines a parallel execution pattern by the use of k cloned naplets, each for an equal share of the servers (for simplicity, we assume n can be divided by k). Each naplet runs a guardian function defined in the Checkable object ResultVerify before each access. The naplets report their results to home at the end of their execution. Note that the class ApplAgentProg is declared as a private inner class of the naplet so that the access pre-condition can be defined based on the naplet states.

private class ApplAgentProg extends Access {
    private ApplAgentProg(String[] accesslist, int k) {
        Checkable guard = new ResultVerify();
        Observable report = new ResultReport();
        int n = accesslist.length;
        for (int i=0; i < k; i++) {
            new AccessPattn(guard, accesslist[i*k + j], report);
        }
    }
    setProgram(new ParPattn(seqProg));
}
to enforce the coordinated access control policies when a resource access request is received. The Naplet system defines a NapletSecurityManager as an extension of SecurityManager for customized permission checks. For example, the following SecurityManager checks the permissions for looking up the yellow page service.

```java
public class NapletSecurityManager extends SecurityManager {
    public void checkPermission(Permission perm) {
        if ((perm instanceof NapletServicePermission)) {
            AccessControlContext acc = AccessController.getContext();
            Subject sbj = Subject.getInstance(acc);
            String p_serv = perm.getName();
            String p_acts = perm.getActions();
            if (sbj.getPrincipals() contains "song\wayne.edu/MyNaplet"
                && p_serv == "yellow-page" && p_acts contains acts)
                if (NapletAccessControl.spatialConsCheck(sbj, acts)
                    && NapletAccessControl.temporalConsCheck(sbj, acts)) {
                    super.checkPermission(perm);
                    return;
                }
        } else
            throw new SecurityException("MyNaplet from song is denied");
    }
}
```

### 6 An Example in Coalition Environment

Nowadays, softwares are becoming more and more complex. They may become hundreds of MegaBytes or even GigaBytes in size. Usually, a software is composed of a large number of modules, like the objects in Python or the toolkits in Matlab. To save the storage space of a server and balance the usage requests from sharing users, software modules could be distributed over multiple servers in an enterprise network. These enterprise-wise servers constitute a coalition environment.

To ensure the licensed modules of a software are well-installed and not compromised, its system auditor needs to verify their integrity. In order to reduce the impact on the software usage, the verification procedure should be completed within a pre-specified period of time. In this example, we consider the verification of subordinate modules of an application software in an enterprise-wise coalition environment.

It is convenient and efficient to utilize code mobility to achieve this end, by exploiting data locality. The auditor dispatches a mobile code to roam in the network. The mobile code is programmed to calculate hash values of the software modules by some hash algorithm, e.g. SHA-1. After verifying the integrity of modules located on one server, the mobile code migrates to another server for its mission.

More often than not, there are dependencies among these software modules, e.g. a call to a method of another module and references to an instance of another module. Figure 1 illustrates the module dependency. A directed line from module A to D represents module A depends on D. The dependency relationship can be depicted by a digraph. It also poses the following implication, a module is verified as correct if and only if all of its depended modules and itself are correct. This type of spatial requirements can be specified by our SRAC language. For example, the spatial

![Figure 1. The module dependency digraph. Each circle represents a module. The dotted lines show the distribution of modules on different servers.](image)

### 7 Related Work

There were recent temporal access control models for monitoring time-sensitive activities. For example, Bertino, et al. [2] presented a time-based scheme by using periodic authorizations and derivation rules. Each authorization is assigned a periodicity constraint, which specifies a set of intervals when the authorization is enabled. Later on, the authors integrated such interval-based temporal constraints into role-based access control and developed a temporal RBAC (TRBAC) model to efficiently manage permissions of temporal roles [3]. TRBAC was recently generalized by Joshi, et al. [12] by incorporating a set of language constructs for the specification of various temporal constraints on roles, user-role assignments and role-permission assignments. TRBAC models can be tailored to support temporal constraints in mobile computing, but they can’t accommodate spatial constraints because it doesn’t consider the relevance between authorization and performed access actions. In addition, the discrete time structure in these access control models affects their applicability to real systems, because in most cases time is continuous.

History-based access control may be viewed as a pragmatic approximation to information-flow control that keeps track of
code execution. In [1], the run-time rights of a piece of code are determined systematically by examining its execution history, i.e. the attributes of the code that have run before and any explicit requests to augment rights. But this mechanism only inspects the execution history on the local site. As a result, it can not be applied to access control in a coalition environment, where the authorization decision depends on the access actions on other related sites. Little work has considered the spatial requirements for mobile computing security. In [5], Guy, et al. presented a mechanism to use a selective history of the access requests of a mobile code in access control. Their history-based access control model is limited to one-hop mobile codes. Because it has no concept of shared resource (objects), this model is hardly extensible to include the access history of a mobile computation in a coalition environment.

Mobile agent provides a special model for mobile computing, where autonomous agents travels across networks carrying out their tasks on behalf of their owners. There is much work focusing on the security issues of mobile agent systems (see [10] for a comprehensive review). Among them, protecting the docking service from the attacks by malicious or malfunctioned agents is important for the service providers. A most noteworthy approach is proof-carrying-code (PCC) [16], which allows a host to determine with certainty if it is safe to execute a remote code. To safely grant the privileges to a mobile agent, Farmer et al. [7] proposed a state appraisal approach to ensure that an agent with corrupted states won’t be granted any privilege. In [18], we prototyped an agent-oriented authorization mechanism that realized privilege delegation and agent-oriented access control. However, these approaches do not consider the spatio-temporal requirements for host protection in mobile computing systems, either. Our shared resource access language bears certain resemblance to the mobile agent itinerary language proposed in [14]. But, we focus on the resource access behaviors in mobile computing, and investigate the access control with spatial constraints.

8 Conclusions

In this paper, we present a coordinated access control model with temporal and spatial constraints for secure mobile computing. The various access patterns of mobile devices are expressed by the expressive SRAL language. The constraint satisfaction problem is decidable. We extend RBAC model to express and enforce the spatio-temporal constraints, and have implemented it in the Naplet system. To extend our work, we will look into some other implementation issues, such as how to classify the temporal permissions and aggregate their validity durations.

Acknowledgement This work is supported in part by the U.S. National Science Foundation under Grants CCR-9988266 and ACI-0203592.

References