Privacy Arguments:
Analysing Selective Disclosure Requirements for Mobile Applications

Thein Than Tun[1], Arosha K. Bandara[1], Blaine A. Price[1], Yijun Yu[1]
[1]The Open University, UK

Charles Haley[2]
[2]Frogfish Technologies
UK

Inah Omoronyia[3], Bashar Nuseibeh[1,3]
[3]Lero, Ireland

***Abstract*—Privacy requirements for mobile applications offer a distinct set of challenges for requirements engineering. First, they are highly dynamic, changing over time and locations, and across the different roles of agents involved and the kinds of information that may be disclosed. Second, although some general privacy requirements can be elicited *a priori*, users often refine them at runtime as they interact with the system and its environment. Selectively disclosing information to appropriate agents is therefore a key privacy management challenge, requiring carefully formulated privacy requirements amenable to systematic reasoning. In this paper, we introduce *privacy arguments* as a means of analysing privacy requirements in general and selective disclosure requirements (that are both content- and context-sensitive) in particular. Privacy arguments allow individual users to express personal preferences, which are then used to reason about privacy for each user under different contexts. At runtime, these arguments provide a way to reason about requirements satisfaction and diagnosis. Our proposed approach is demonstrated and evaluated using the privacy requirements of BuddyTracker, a mobile application we developed as part of our overall research programme.**

Keywords-privacy arguments; privacy requirements; mobile applications

I. INTRODUCTION

The prevalence of mobile computing devices and cloud-based services has generated new opportunities and new ways in which software systems ("apps") are used. These opportunities also present new challenges for requirements engineering. Although similar in many ways to the requirements for traditional software systems, requirements for mobile applications have a significant privacy dimension, because they capture, store and process large amounts of personal data. The consequences of inattention to privacy concerns is illustrated by many well publicised examples of privacy breaches, such as the automatic and unintentional uploading of users' personal contact databases from their mobile devices to the servers of the popular mobile social media service, Path [1].

Unlike privacy requirements centrally prescribed in regulations and site policy statements, user privacy requirements for mobile applications are difficult to describe and analyse. First, privacy requirements are highly dynamic and selective, changing over time and locations, and across the roles of agents involved, as well as the kind of information that may be disclosed. Second, although some general privacy requirements can be elicited, users are likely to refine their requirements at runtime as they interact with the mobile system in different contexts. As privacy requirements become more concrete and their contexts become clear, a system has to be adaptive to those requirements.

Selective disclosure defines the conditions for a piece of information to be communicated to a particular group of agents (people or software). Such conditions can be specified as policy rules called "norms" [2]. However, they are often specified without giving explicit rationale to answer questions such as why is certain information (not) shared? Is it worthwhile to sacrifice a low-priority policy by enlarging the disclosed scope in exchange for some important functionality or gaining trust from friends? Does information have different values at different times? What preferences among different norms need to be traded off at runtime?

To answer these questions, this paper proposes an extended argumentation language for such selective disclosure requirements, in order to reason about their satisfaction. First, the language allows the definition of classes of argument capturing how a generic privacy norm can be satisfied for a group of users within a particular context, with the goals of such groups explicitly expressed in the conditions. Individual users of the system can instantiate the argument class and, when necessary, augment it with privacy requirements specific to a particular location, time, or other context. Since there is clear traceability between the arguments and the underlying system architecture, changes in the argument can be used to adapt the system behaviour. Our proposed privacy arguments can be either formal or semi-formal. Users can interact with the semi-formal form of arguments at runtime, by specifying requirements for specific disclosure contexts. We use the Event Calculus [3] to formalise these privacy arguments to reason about requirements satisfaction and diagnosis. Our proposed approach is demonstrated using examples of selective disclosure of location information from the *BuddyTracker* mobile application developed by our research group.

The main contribution of the paper is therefore the use of privacy arguments to represent highly dynamic and changing selective disclosure requirements, and to relate them to the software architecture in order to enable system adaptation to runtime privacy requirements.

The remainder of the paper is organized as follows. By way of background, section II provides an overview of the BuddyTracker application, a discussion of the privacy

framework that helps us define privacy requirements, the Problem Frames approach, and the Event Calculus. Section III explains our proposed approach, by discussing how functional and privacy requirements are described and related using a generic architecture, and then introducing the privacy arguments and their formalisation and reasoning using the Event Calculus. Section IV provides some of BuddyTracker's implementation details. Related work and conclusions are presented in Sections V and VI respectively.

II. PRELIMINARIES

A. Case Study: BuddyTracker

BuddyTracker is a smartphone app that regularly updates a cloud-based server with the phone's location as determined by GPS, WiFi, or current cell mast location. At a given time, it allows the owner to set the precision of their reported location as well as who is allowed to access their location. BuddyTracker's architecture is based on Altman's bi-directional privacy theory [4] and the concept of social translucence [5] where sharing knowledge about location requests helps provide a kind of two-way visibility for actions. This means that when an authorized person (the tracking user) requests the location of another user (the tracked user, a colleague, a friend, etc.), the tracked user is notified of the request. *BuddyTracker* uses various algorithms to analyse the context of the tracked user and chooses the most appropriate interface method to notify her, including natural language, tone, vibrotactile, and a variety of visual notification methods.

B. Privacy Norms

There are several definitions of privacy requirements, and in this work we follow the notion of privacy as defined by the formalized *contextual integrity* framework [2]. In that formalization, agents with certain knowledge, who play different roles, communicate with each other. Their communication should follow certain norms. Agents may play different roles at different times, and make deductions based on the knowledge they have. Norms are established on the basis of who the personal information is about, how the information is transmitted, what the subject and the users of the information have done in the past and will do in future.

In particular, two types of norms have been distinguished. A positive norm permits communication between agents as long as its temporal condition is satisfied. For instance, a positive norm, may allow an agent Alice with the role of employee to communicate to another agent Bob playing the role of colleague, a message containing the current location information of Alice. This will allow Alice to tell her colleague Bob about where she currently is. The norm is violated if Bob knows Alice's location when she is not playing the role of colleague to Bob, for instance.

A negative norm permits communication only if its temporal condition is not satisfied. For instance, a negative norm may state that an agent with the role of user is not allowed to disclose the location information of another agent with the role of friend to a third party, unless consent from the second agent has been obtained by the first agent. For instance, Alice is not allowed to disclose Bob's location unless Bob's consent has been obtained by Alice. In this work we regard privacy requirements as both positive and negative norms that must be respected by the behaviour of the agents in the system.

Since agents can make deductions about the information they get, the knowledge structure is important. For instance, if a norm does not allow Carole to know the location of another agent, Bob, on weekends, then Carole must not know anything about Bob that might indicate his location. If his location can be deduced from the postcode, address, GPS location, or IP address, then agent Carole has to be prevented from acquiring such knowledge. This assumes that (i) the privacy models must describe all inferences agents can make on the information they can get, and (ii) agents will not acquire knowledge in ways other than as described in the model.

According to Barth et al [2], the behaviour of a system is described using the linear temporal logic (LTL) traces thus supporting verification against privacy requirements.

C. Problem Frames and Security Arguments

In this work, we will use the Problem Frames approach [6] to make four descriptions: the software (*S*), the problem world (*W*), the requirements (*R*) and their relationship through the entailment $W, S \vdash R$ [7]. This approach is suitable for representing and analysing privacy requirements for two main reasons: (i) it allows us to describe the structure and properties of the context in which the requirements are expressed, and (ii) it allows us to decompose and recompose requirements in a way that the concerns of the components in the system architecture are separately addressed. The use of the Problem Frames approach is not prescriptive in the sense that another requirements engineering approach that allows us to make the same four descriptions and separate concerns addressed by the components could be used instead.

Haley et al [8], have used Toulmin's argument structure to recursively represent the rebuttals and mitigations when reasoning about the satisfaction of security requirements. In their approach, security requirements are expressed as claims, and are supported by grounds and warrants. Rebuttals show evidence that contradicts other arguments, whilst mitigations describe how rebuttals may be avoided or tolerated. Franqueira et al [9] combine the process of security arguments with that of risk assessment in order to exploit the publicly available security catalogues [10]. This work extends the security arguments to address the distinct challenges of selective disclosure privacy problems.

D. Event Calculus

In order to facilitate formal reasoning, some of the artefacts will be described in Event Calculus, a logic based on first-order predicate calculus. It can be used to represent actions, their deterministic and non-deterministic

effects, concurrent actions and continuous change. We chose the Event Calculus as our formalism because it is suitable for describing and reasoning about event-based temporal systems [3].

Table I. Event Calculus predicates

Predicate	Meaning
Happens(a, t)	Action *a* occurs at time *t*
Initiates(a, f, t)	Fluent *f* starts to hold after action *a* at time *t*
Terminates(a, f, t)	Fluent f ceases to hold after action a at time *t*
HoldsAt(f, t)	Fluent *f* holds at time *t*
t1 < t2	Time point *t1* is before time point *t2*

The calculus relates events and event sequences to 'fluents' that denote states of a system. Table I gives the meanings of the elementary predicates of the calculus we use in this paper. There are several domain-independent rules, some of which are listed below (see [3] for other rules). *Clipped(t1, f, t2)* is equivalent to saying that the fluent *f* is terminated by the event instance *a* occurring between times *t1* and *t2*.

$$Clipped(t1, f, t2) \equiv \exists a, t\ [Happens(a, t) \wedge t1 \leq t < t2 \wedge Terminates(a, f, t)]$$

The next rule says that the fluent *f* that has been initiated by occurrence of an event *a* continues to hold until occurrence of a terminating event.

$$HoldsAt(f, t2) \leftarrow [Happens(a, t1) \wedge Initiates(a, f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)]$$

The last rule says that fluent *f* persists until an appropriate terminating event occurs.

$$HoldsAt(f, t2) \leftarrow [HoldsAt(f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)]$$

III. The Privacy Argumentation Approach

In this proposed approach, functional requirements and requirements relating to privacy norms are handled separately. Methodologically, functionality requirements are specified before privacy requirements are considered. This separation is analogous to the separation of system behaviour from that of access control.

In this section, we discuss (i) how we describe the functional and privacy requirements using the Problem Frames approach, (ii) how privacy arguments are used to specify norms that underlie privacy, and (iii) how formal reasoning are performed based on privacy arguments.

We envisage that the problem frames, arguments in the natural language and the Event Calculus are tools for the developers. Developers could also define classes of privacy requirements, which users can instantiate and personalise perhaps through special user interface metaphors, such as check buttons and dropdown lists.

A. Problem Description and Analysis

After identifying functional requirements, we relate them to the problem world domain and the machines using the Problem Frames approach, as a problem diagram. For example, the functional requirement of *BuddyTracker*, "Display the location information of a user on request" is modelled as a simple problem diagram in Figure 1.

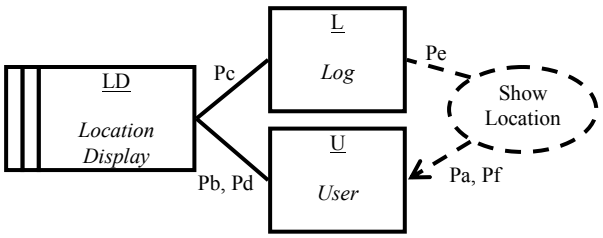

Figure 1. Problem Diagram: Show Location

It is an instance of the Information Display problem frame [6], in which the problem world domain User is the person interacting with the machine Location Display. The user taps the display when he wants to know where someone is. The Log has the schema *<userid, GPSLocation, time>* in the database that contains entries of GPS location recorded when users update their locations. The labels *Pa* to *Pf* are explained in Table II.

Table II. Meanings of the phenomena in Figure 1.

Label	Phenomena	Meaning
Pa	*WhereIs(s)*	The user wants to know the location of subject identified by *s*
Pb	*Tap(u, cIcon)*	The user *u* taps the icon representing the subject identified by *c*
Pc	*Query(s, GPSPos@t)*	The query operation returns the GPS position of subject *s* at time *t*, which is the time of last know location
Pd	*At (s, GPSPos@t)*	The machine indicates that the subject *s* is at the location *GPSPos* at time *t*
Pe	*Log*	Entries in log
Pf	*Know(u, s, GPSPos@t)*	The user *u* knows that subject *s* is at *GPSPos* at time *t*

In terms of the phenomena, the requirement says that *WhereIs(s)* (phenomenon *Pa*) should lead to *Know(u, s, GPSPos)* (phenomenon *Pf*), meaning that when the user wants to know the location of someone when tapping the icon (phenomenon *Pb*), she will know the position. In Event Calculus, it is expressed as follows:

$$Happens(WhereIs(s), time) \rightarrow HoldsAt(Know(u, s, GPSPos@t), time+4)$$

Notice the difference between time *t* in *GPSPos@t* and *time* in *time+4* in the *HoldsAt* predicate: the former is the time at which the GPS position is recorded, while the latter is the relative time difference between the user action tapping and the user knowing the location of the subject. The number *4* is the number of ticks on the logical clock.

It is easy to see that a machine that queries the log (phenomenon *Pc*) when the user taps the display (phenomenon *Pb*) and immediately shows to the user where the subject is at the last known time (phenomenon *Pd*), will satisfy the requirement. This specification of Location Display can be formalized as follows:

Happens(Tap(u, cIcon), time) →
[Happens(Query(s, GPSPos@t), time + 1) ∧
Happens(At(s, GPSPos@t), time+2)]

Full formalisation of this and other examples in the Event Calculus is given in the appendices.

1) Describing the Norms in Relation to Functions: As discussed by Barth et al [2], much of the information flow between agents should conform to certain positive or negative privacy norms. Although the problem diagram in Figure 1 describes the physical contexts for the requirement including User and the location database Log, the norm that permits such information flow is in fact missing. Consider the following privacy norm*:*

"Location information can be disclosed between those who are colleagues."

A norm such as the one above is regarded as a privacy requirement in this work. This requirement is a positive norm because it states the condition (perhaps one of many conditions) under which showing location to a user is acceptable. Note that the requirement is a default requirement for the entire user class: it says that it is true for all colleagues. However, an individual user may choose to personalise it, for instance, by specifying a preference such as: Alice does not want her colleague Dave to know her location.

This requirement is related to the principle of selective disclosure. Three factors are recurrent in such requirements: agent roles, time and place. Consider, for instance, what makes two people to be considered as colleagues. Do they have to work in the same organization? Do they have to work at the same location during similar times? These questions require a clear distinction between these context domains.

The Problem Frames approach allows us to describe the privacy requirement that underlies the functional requirements in a modular way. Assuming that *userids* contain prefixes indicating the organization the user works for, we can extend the problem diagram in Figure 1 to reflect the norm, without modifying the existing machine, Location Display. This can be achieved by introducing a wrapper that intercepts events at the machine-world interface.

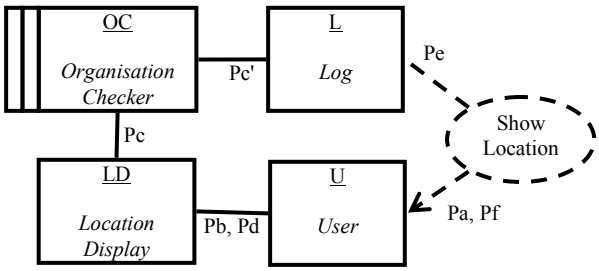

Figure 2. Problem Diagram: Colleague Norm 1

Figure 2 shows how the original problem context can be extended to implement the norm that location information shall be displayed if the user making the request and the subject of the request work for the same organization. The new machine Organization Checker superimposes itself at the interface between Location Display and Log. Its specification is to obtain and compare the organization prefixes of *c* in *Query(c, GPSPos@t)* and u which identifies the user. If the prefixes do not match, the Organization Checker machine should not to pass the query from Location Display on to Log. Assuming that the fluent *SamePF(u,c)* indicates that the organization prefixes of the user identified by *u*, and the subject identified by *s*, the condition Organization Checker has to check can be described in Event Calculus as follows:

HoldsAt(SamePF(u,s), time)

If the condition is not met, Organization Checker must block the access to Log by Location Display. As a result, the user can see locations of colleagues only.

One side issue that may arise here is that of information leak: the user may discover that when access is not successful, who is not a colleague. Therefore, the response message to the user must be phrased with care not to leak the identity of the person.

As well as introducing a wrapper machine such as Organization Checker, it is possible to add other world domains when implementing a norm. If, for instance, organizations that users work for are not defined through the *userid* prefix but in a lookup table, then the table may be included as an additional domain and the Organization Checker be linked to it.

It is likely that there will be several norms affecting a particular information transmission. Another norm, in this example, could be one that states that friends can share location information and may be implemented by a machine similar to Organization Checker in Figure 2.

Selective disclosure norms may crosscut several functional requirements, and hence the problem diagrams. A wrapper architecture can be used to compose functional requirements, and norms can be introduced to composed problem diagrams, as discussed in earlier work [11].

2) General Architecture: Notice that the specification of Location Display has not been changed because of the addition of a colleague norm. The new machine simply extends the current context of Location Display and introduces a condition on its access to Log.

Methodologically, this approach allows us to consider functional requirements before examining the privacy norms. Since machines that implement the privacy norms are introduced after the machines implementing the functional requirements have been specified, concerns of the components have been separated. This separation of functional concerns from privacy concerns is a common characteristic in access control systems.

Therefore, we consider this wrapper architecture to be a general solution to the problem of describing privacy requirements in the Problem Frames approach.

3) Formalising Norms: Formalising norms using the Event Calculus is relatively straightforward. A disjunction of conditions for positive norms are conjunct with the specification of the functional requirements [2]. For instance, if colleagues or friends can share location information, the specification in Event Calculus could be:

$$[HoldsAt(SamePF(u,s), time) \vee HoldsAt(Friends(u,s), time)] \wedge [Happens(Tap(u, cIcon), time) \rightarrow Happens(Query(s, GPSPos@t), time + 1) \wedge Happens(At(s, GPSPos@t), time+2)] \quad (CFN)$$

The above specification says that the tapping action of the user *u*, on the screen for location of user *c*, will lead to the machine querying for the last known GPS position of *c*, and showing the position within *2* time units, if either that the users *u* and *c* are colleagues, or that they are friends.

The specification of functional requirements must imply the disjunction of conditions for negative norms [2]. For instance, if only colleagues or friends can share their location information, the norm can be specified as:

$$[Happens(Tap(u, cIcon), time) \rightarrow Happens(Query(s, GPSPos@t), time + 1) \wedge Happens(At(s, GPSPos@t), time+2)] \rightarrow [HoldsAt(OrgPrefix(u)= OrgPrefix(c), time) \vee HoldsAt(Friends(u,c), time))]$$

The above specification says that if the tapping leads to the user knowing the location information of another user, they must either have some common organization prefix or they must be defined as friends. In other words, users with other relationships cannot share their location information in this system.

B. Privacy Arguments

From a developer's perspective, a privacy argument justifies to an audience, such as users of mobile applications, that the user's privacy claim has been respected by the software system. In a formal setting, this justification may be a proof; in a less formal setting, this justification may be an argument. Furthermore, privacy arguments can be used to specify users' runtime privacy requirements that are highly contextual, specific to individual users, location, time and content. Therefore, although similar to other forms of arguments, such as security arguments [8], the argumentation language is now extended in three ways to better describe privacy specific problems.

First, privacy requirements or selective disclosure norms are now described as the claim of an argument that needs to be justified. The ground is the collection of facts that can be observed from the world domains, which supports the claim. The warrant is the collection of domain-specific rules that links the ground to the claim of the argument. The general structure of privacy arguments is: (Warrant, Ground → Claim). As with security arguments, problem diagrams provide the basic structure for the privacy arguments and traceability between high level privacy requirements, and access control policies that should implement those requirements. For instance, the Organisation Checker machine in Figure 2 is in effect an access control policy, but the problem diagram and subsequent privacy arguments will show how the privacy requirement may be achieved, whilst indicating how the system may fail to satisfy the requirement.

Second, we now distinguish between an argument class and an argument instance. An argument class may be constructed on the basis of general privacy norm for a set of users within a typical context. In a sense, an argument class defines a default privacy argument that a user may instantiate. Once instantiated, an argument instance may augment the norm with more specific requirements or preferences that are specific to a user, time, location and content. For instance, although the colleague norm allows sharing location information between colleagues, Bob may opt not to share his with the colleague Dave. Privacy arguments should be able to express such a preference.

Third, since there may be a number of norms applicable to a specific piece of information, users may wish to express a preference for a certain norm over others. For instance, although there are colleague and friend norms that allow a user to access and disclose his location information, Bob may want only his colleagues, not friends, to access his location information. Privacy arguments should be able to express such a preference.

In this work, privacy arguments are constructed always from the perspective of the users, and users may refine their privacy requirements at runtime in terms of preferences. Therefore, individual users need only decide their exact privacy requirements at the time of use.

One implication of this characteristic of privacy requirements is that privacy arguments cannot remain static at runtime: they should be responsive to preferences of individual users in different contexts. Since privacy arguments are linked to specifications through the shared phenomena, they can be used to adapt the system behaviour at runtime.

1) Privacy Argument Grammar: Before reasoning about the arguments specific to privacy requirements, we first represent them by extending our earlier security argument structures [8] [9], with the concepts of temporal and priority preferences *[12]*.

Figure 3 lists the production rules of an extended syntax in the *TXL* grammar definitions [13]. TXL is chosen instead of Backus Normal Form (BNF) for its brevity in expressing language extensions.

The "include" statement says that our new syntax reuses the existing syntax of arguments defined for our tool OpenArgue [14]. The new keywords in this extended language are `preferred`, `precedes`, `except` and `when`. As in our previous grammar, an argument in the new grammar still has a claim, zero or more grounds and zero or more warrants. The additional element preference can be used to define two things: (i) the precedence of two arguments over each other under certain optional conditions; and (ii) the exceptional conditions when a particular argument should not be applied.

```
include "OpenArgue.grm"
keys
 … preferred precedes except when
end keys
redefine argument
   argument [claim] {
      supported by
           [ground*]
      warranted by
           [warrant*]
      preferred by
           [preference*]
   }
end define

define preference
[id] precedes [id] when [bool_expr]
| [id] except when [bool_expr]
end define
```

Figure 3. Privacy Argument Grammar

2) Argument Classes: The following example shows an argument class stating how the privacy requirement is satisfied by the BuddyTracker system. Arguments shown here can be visualized graphically *[14]*, but we omit them here for space reasons. Note that in the argument A1, the positive norm for location sharing between colleagues is the claim itself. F1 and F2 are facts. Warrant provides rules R1 to R5 that can link the supporting facts to the claim. The person whose location the user is trying to find is referred to as a subject. The subject and the user are written in double brackets indicating that they can be replaced by specific values when an instance is created. This process of annotating subjects that appeared in the fact/rule descriptions is called parameterisation. A parameterised proposition becomes a predicate whose terms are bound to the grounded facts.

3) Argument Instances: Such arguments can be instantiated for Bob and Dave with their specific names and preferences. We now have personal privacy requirements for each user. For instance, the privacy argument for the positive norm can be instantiated for Bob as follows.

```
argument: Colleague_Norm_Class
A1 "<<User>> can find out location
information of his/her colleague
<<subject>>" {
  supported by
    F1 "<<User>> wants to know location
of <<subject>>"
    F2 "<<User>> taps the screen icon of
<<subject>>"
  warranted by
    R1 "If <<user>> taps the screen, the
machine checks whether <<user>> and
<<subject>> are colleagues"
    R2 "If <<user>> and <<subject>> are
colleagues, the machine queries the
location of <<subject>>"
    R3 "If the log is queried, it returns
the last known GPS location of
<<subject>>"
    R4 "If GPS location is obtained from
the log, the machine shows it to
<<user>>"
    R5 "If GPS location is shown to
<<user>>, <<user>> knows the location of
<<subject>>" }
```

Figure 4. Privacy Argument for the Colleague Norm

```
argument: Colleague_Norm_Bob
A2 "Bob can find out location information
of his/her colleague <<subject>>" {
  supported by
    F1 "Bob wants to know location of
<<subject>>"
    F2 "Bob taps the screen icon of
<<subject>>"
  warranted by
    R1 "If Bob taps the screen, the
machine checks whether Bob and
<<subject>> are colleagues"
    R2 "If Bob and <<subject>> are
colleagues, the machine queries the
location of <<subject>>"
    R3 "If the log is queried, it returns
the last known GPS location of
<<subject>>"
    R4 "If GPS location is obtained from
the log, the machine shows it to Bob"
    R5 "If GPS location is shown to Bob,
Bob knows the location of <<subject>>" }
```

Figure 5. Instantiation of the Colleague Norm by Bob

This argument when instantiated for Bob shows why Bob is able to see locations of certain subjects (because they are his colleagues). When there are multiple privacy norms, such as the friend norm A3 (not shown here for

space reason) in addition to the existing colleague norm A2, a user may express preference for one norm over other, or by adding conditions on time and location for when those preferences should be used.

For instance, Bob may wish to use the colleague norm only in weekdays and the friend norm on weekends.

```
A4 "<<User>> can find out location
information of his/her colleague
<<subject>>" {
 preferred by
   A2 precedes A3
     when (day >= Monday & day <= Friday)
   A3 precedes A2
     when (day >= Saturday & day <=
Sunday) }
```

Figure 6. Bob's Preferences for the Colleague and Frien Norms

As well as defining conditional precedence of privacy arguments, Bob may also wish to deny Dave access to his location information by adding an exception to the colleague norm class A1, as shown below.

```
A1 except when
     (user==Dave & subject==Bob)
```

Figure 7. Bob's Exeception #1 to the Colleague Norm

The exception above says that A1 should not be applied when the user Dave attempts to find out location information of Bob. An alternative formulation to this shorthand will be for Bob to create a new argument denying Dave access to his location and to define a preference to the new norm over A1. The exception shorthand is also useful if, for instance, Bob wants to restrict himself from viewing location of certain colleagues such as Dave.

```
A1 except when
     (user==Bob & subject==Dave)
```

Figure 8. Bob's Exeception #2 to the Colleague Norm

So far, we have assumed that a user is able to prohibit other users accessing his or her location information, even when the norm is to share it, and the user can prohibit himself or herself from accessing location information of other users. We envisage that privacy norm classes are defined by developers, and then instantiated and adjusted by individual users on the basis of their individual priorities for norms and exceptions to norms. In general, norms allowing access to information are written either by the developer or the subject of the information.

4) Formalizing Privacy Arguments: We now discuss how arguments can be formalized using Event Calculus. Since we are using the reasoning tool decreasoner, the discussion here follows the syntax of the tool ([3], Chapter 13). An argument class is defined using sorts, and an instance of the class is created as constants for the sorts.

```
sort agents
sort subject: agents
sort user: agents
sort loc

user Bob
subject Dave
loc GPSPos
```

In the above listing, subject and user are defined as subclasses of the agents sort. Alice, Dave and GPSPos are constants. For built in sorts, such as *integer*, value ranges can be given. The claim of an argument class such as Colleague_Norm_Class can be written as follows (see also the earlier rule labelled (CFN)):

```
[time,user,subject,loc]
(HoldsAt(SamePF(user,subject),time) |
HoldsAt(Friends(user,subject),time)) &
(Happens(Tap(user,subject),time) ->
(Happens(Query(subject,loc),time+1) &
Happens(At(subject,loc),time+2))).
```

In the above listing, the square brackets "[" and "]" denote universal quantification of the variables inside. Facts are written as observations. The fact that the user Bob wants to know at time 1 where Dave is written as:

```
Happens(WhereIs(Dave),0).
```

Warrants are written as a set of domain-specific rules. For instance the warrant rule (R5) can be written as:

```
[time,subject,loc,user]
Initiates(At(subject,loc),
            Know(user, subject, loc),time).
```

The rule above says that when the event *At(subject, loc)* happens, the user will know the location *loc* of the colleague *subject* at the next time point.

In the Event Calculus, exceptions to norms can be described using the abnormality predicate Ab_i [3] (Chapter 12). For instance, the following says that the norm Colleague_Norm_Class holds, unless there are exceptions:

```
[time,user,subject,loc]
(( HoldsAt(SamePF(user,subject),time) |
HoldsAt(Friends(user,subject),time)) &
Happens(Tap(user,subject),time) &
!Ab1(subject,user,time) ) ->
(Happens(Query(subject,loc),time+1) &
Happens(At(subject,loc),time+2)).
```

It is now possible to provide a list of conditions, in an elaboration tolerant way, when the norm Colleague_Norm_Class should not hold. For instance, Dave can say that Bob should never know his location:

```
[time] Ab1(Dave,Bob,time).
```

This statement can be extended so that the abnormality predicate is true for all users, rejecting the entire norm.

C. Reasoning about Privacy Arguments

Formalization of privacy norms and arguments are useful because they can be used to check some important privacy properties in the system. These properties include:

1. **Information availability**: Is access to information according to norms possible? If Bob and Dave are colleagues according to some norm of location sharing, can they find out where each other is? This could be useful if there are other norms preventing the sharing of information. In the Event Calculus, this is done by means of deduction or temporal projection of the claim in the privacy argument.

2. **Denial**: Is access to information contrary to norms possible? If Dave and Carole are not colleagues, and if non-colleagues cannot share location information according to a norm, can they find out where each other is? This checking is useful in order to find out possible violation of privacy requirements. In the Event Calculus, this is done by means of deduction or temporal projection that the negation of the claim in the privacy argument leads to contradiction.

3. **Explanation**: Why was access to certain information successful or unsuccessful? If Carole was not allowed to find out where Bob was, why? If Bob could find out where Dave was, why? This reasoning gives explanation in terms of action sequences, and is useful for diagnostics. In the Event Calculus, this is done by means of abductive reasoning.

The reasoning tool we use in this work, *decreasoner* [3], supports all these types of reasoning.

IV. The BuddyTracker Case Study

BuddyTracker is currently implemented on iPhone and Android platforms, although the advanced context sensitive privacy notification features are only implemented in the Android version. The architecture of the system is such that the mobile application regularly updates a cloud-based server with the phone's location as determined by GPS, WiFi, or current cell mast location. BuddyTracker uses a number of available sensors, such as the GPS, accelerometer, light sensor, system logs, information about currently running applications and other methods to collect the most accurate information about the user's context. Calendar entries can be used to determine the user's current activity; and Google Geo Service is used to translate GPS coordinates into more meaningful text descriptions. The BuddyTracker server has a database of tracker-trackee relationships and for each tracker, the trackee can choose to reveal either her exact (street) location, the city she is in, the country she is in, or reveal nothing at all (invisible). This last feature provides the selective disclosure capabilities we are focussing on in this paper (Figure 4). The BuddyTracker server also integrates a machine learning system that can automatically infer constraints on these privacy settings based on user behaviour [15]. These learned constraints could be used as warrants in the privacy arguments.

Our case study took a dozen functional requirements of BuddyTracker to analyse the composition to the selective disclosure privacy norms defined earlier. We also implemented the logged events (both contextual information and the preference changes) during runtime for an offline analysis because currently the *decreasoner* reasoning module only supports a Linux command line interface. Using the logged events, we found that *decreasoner* can reveal whether the selective disclosure norms, adapted to the preferences specified by individuals, are violated or not. This finding gives us confidence that our formalisation of privacy requirements and arguments can provide useful information regarding the runtime satisfaction of privacy. Therefore we plan to integrate *decreasoner* into a common gateway interface so that the reasoning system can be accessed at runtime.

Figure 9. BuddyTracker selective disclosure features

V. Related Work

This section discusses existing work in the areas of (i) requirements engineering for privacy, (ii) mobile privacy and (iii) formal frameworks for requirements analysis. We separate the discussion of how these areas are covered by the contextual integrity framework [2] from other approaches to privacy and security requirements.

A. Contextual Integrity

There are different justificatory frameworks for information disclosure that are applicable to mobile applications. The most common are techniques that resort to private-public dichotomy to justify scenarios where privacy is preserved or threatened [16]. Value based trade-offs is another privacy justifying framework that does not see privacy as a moral right, but as preference over other values [17]. Thus, the rationale for a mobile usage scenario

posing a privacy threat is based on its supports or conflicts with other functionality of the system such as performance or usability. The challenge with these approaches to privacy is that software systems provide phenomenal ways to track and aggregate user's information in a manner where neither private-public dichotomy nor value-based trade-offs is able to capture ensuing privacy implications.

Contextual integrity [2] is another privacy justificatory framework. This framework posits that the transfer of information about a subject from a sender to a receiver in a specific context is tied to certain transmission principles. Such transmission principles are represented as norms that define the expected behaviour of interacting agents in a defined context. Examples of such transmission principles include notice, consent, confidentiality, fiduciary, secrecy, and reciprocity [2]. In this paper, we argue that contextual integrity is a more suitable justificatory framework for modelling mobile privacy in software systems. This is because contextual integrity represents an explicit model of a sender, receiver and a subject when disclosing personal information, and the transmission principles that guard the interaction process between these entities [2]. Additionally, contextual integrity provides a means to identify points in the behaviour of a system where the tracking and aggregation of private attributes of users can lead to privacy violation. However, we also found the notion of contextual integrity needs to be extended to the "why" dimension, by indicating the exchange for desired functionality and the avoidance of undesired functionality as the source of the positive and negative norms.

B. *Other Work on Privacy and Security Requirements*

Privacy has been commonly viewed as a dialectic and dynamic boundary regulation process [4]. Palen and Dourish [18] had gone on to argue that the dialectic nature of privacy suggests that it is conditioned by individual subjective experiences and expectations. The dynamic nature of privacy on the other hand suggests that it is always under continuous negotiation and management. Typically, an individual might choose to change her privacy requirements in exchange for certain benefits or under certain operational context.

Privacy requirements have been analysed from different perspectives by the requirements engineering community. Breaux and Anton [19] have developed a methodology for extracting access rights and obligations from regulatory texts to ensure statement-level coverage for an entire health-care regulation (HIPAA). Similarly, Yu and Cysneiros [20] modelled privacy as a non-functional requirement in i* using OECD guidelines. While these methods are useful in extracting privacy requirements from existing laws and regulations (e.g. OECD guidelines, FIP and EU Directives), they do not specifically address the privacy problems experienced by mobile users. For example, Mancini et al [21] show that when mobile users accessed their personal information in public places such as public transport, fellow commuters were able to read personal information off the mobile screen causing privacy issues for the mobile user.

One of the ways to capture behaviour requirements for a software system is through the use of Use Cases. Seyff et al [22] developed a software environment called ART-SCENE to discover and document stakeholder requirements by walking through scenarios that are automatically generated from use case specifications. They created an extended mobile version called Mobile Scenario Presenter (MSP) using a mobile browser and wireless access to connect to the server-side ART-SCENE scenario system. In some ways, this system could help in discovering the missing privacy requirements that are closely linked to the functional requirements of existing systems; however, there are certain drawbacks in using this system. First, it is difficult to design scenarios *a priori* for mobile privacy as it depends on the users' changing perception of the (emerging) context. Second, it may not be practical to ask users to type their privacy requirements into a PDA or mobile device as mobile users may be in transit or in a situation where they may be constrained to use their mobile devices.

Considering the analysis of privacy requirements, Liaskos et al [12] present a formal reasoning framework after representing preferences as HTN and PDDL3 rules. They classify preferences into temporal and priorities, in addition to the AND/OR semantics of goal refinement hierarchies. Adopting similar means (i.e. formal representation of requirements for reasoning), our work has an additional purpose to compose the requirements with those selective disclosure constraints expressing the positive or negative privacy norms.

Recently a Scala programming language extension, Jeeves, for privacy policies have been proposed [23] to enforce the privacy controls as a wrapper to filter the output of any function in the implemented system with respect to the policies. This is similar to aspect weaving in traditional programming languages, and the privacy policies expressible are in the form of privacy norms. Currently there are difficulties in adopting this language-level implementation because the rationales to the norms are not explicitly documented and it requires a runtime meta-adaptation to enforce the runtime maintenance of privacy requirements.

In summary, privacy has been researched from many perspectives but what has not been adequately addressed are the privacy needs of end-users and in particular mobile system users. This paper demonstrates how privacy arguments can be used to capture these end-user privacy requirements while supporting their run-time evolution.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced privacy arguments as a way to represent and reason about privacy requirements in mobile applications. Like privacy norms, privacy requirements underlie other functional requirements. They are highly dynamic, selective, and changing, according to the information being disclosed, the time, place, and social context of the disclosure. The paper used the Problem Frames approach to separate the concerns of functional

and privacy requirements, and to relate them using a generic architecture.

Privacy arguments were proposed to show how privacy requirements are satisfied by a system, as specified by functional requirements. A class of privacy argument and their composition into satisfaction arguments of functional requirements shows how the general privacy norm is respected by a system. Individual users can instantiate argument classes and specify additional conditions for information disclosure, depending on the place, time, content and other contexts. Since there may be multiple privacy norms, users can also specify their preference for certain privacy norms over others. Therefore, privacy arguments enable users to elaborate their privacy requirements at runtime, and to allow the system to adapt according those privacy requirements elicited at runtime. Privacy arguments can have both formal and semi-formal syntax. We illustrated our approach using an example of selective disclosure from the BuddyTracker application and demonstrated how it is feasible to design such mobile apps to maintain satisfaction of privacy concerns. However, we suggest that arguments can also be formulated for other privacy norms such as informed consent and audit logging.

The main benefits of privacy arguments are: (i) they can be used to relate software component, context and privacy requirements so that the requirement satisfaction can be reasoned about, (ii) they allow users to provide more fine-tuned requirements at runtime, and (iii) they can be used to give diagnostic information to the users when the privacy requirements have been violated.

We plan to deploy our privacy arguments framework through a web service such that mobile apps can look up privacy arguments for runtime adaptation. Additionally, we plan to extend our case study to cover other privacy norms, such as informed consent and control requirements.

ACKNOWLEDGMENTS

This research is partially funded by a Microsoft Software Engineering Innovation Foundation (SEIF) Award, by Science Foundation Ireland grant 10/CE/I1855 and by the European Research Council.

REFERENCES

[1] Associated Press, “CEO apologizes after Path uploads contact lists,” *Yahoo! News*.

[2] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum, “Privacy and Contextual Integrity: Framework and Applications,” in *Proc. of the IEEE Symp. on Security and Privacy*, Washington, DC, USA, 2006, pp. 184–198.

[3] E. T. Mueller, *Commonsense reasoning*. Morgan Kaufmann, 2006.

[4] I. Altman, “Privacy Regulation: Culturally Universal or Culturally Specific?,” *Journal of Social Issues*, vol. 33, no. 3, pp. 66–84, Jul. 1977.

[5] T. Erickson and W. A. Kellogg, “Social translucence: an approach to designing systems that support social processes,” *ACM Trans. Comput.-Hum. Interact.*, vol. 7, no. 1, pp. 59–83, Mar. 2000.

[6] M. Jackson, *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley/ACM Press, 2001.

[7] M. Jackson, “Problems and requirements [software development],” in *Proc. of 2nd Int. Symp. on Requirements Engineering*, 1995, pp. 2– 8.

[8] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, “Security Requirements Engineering: A Framework for Representation and Analysis,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 133–153, 2008.

[9] V. N. L. Franqueira, T. T. Tun, Y. Yu, R. Wieringa, and B. Nuseibeh, “Risk and argument: A risk-based argumentation method for practical security,” in *Proc. of 19th Int. Conf. on Requirements Engineering*, Trento, Italy, 2011, pp. 239–248.

[10] S. Barnum, “Common Attack Pattern Enumeration and Classification (CAPEC) Schema Description,” 2008.

[11] T. T. Tun, T. Trew, M. Jackson, R. Laney, and B. Nuseibeh, “Specifying features of an evolving software system,” *Software: Practice and Experience*, vol. 39, no. 11, pp. 973–1002, Aug. 2009.

[12] S. Liaskos, S. A. McIlraith, S. Sohrabi, and J. Mylopoulos, “Representing and reasoning about preferences in requirements engineering,” *Requir. Eng.*, vol. 16, no. 3, pp. 227–249, Aug. 2011.

[13] J. R. Cordy, “The TXL source transformation language,” *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, Aug. 2006.

[14] Y. Yu, T. T. Tun, A. Tedeschi, V. N. . Franqueira, and B. Nuseibeh, “OpenArgue: Supporting argumentation to evolve secure software systems,” in *Proc. of 19th Int. Conf. on Requirements Engineering*, 2011, pp. 351–352.

[15] D. Corapi, O. Ray, A. Russo, A. Bandara, and E. Lupu, “Learning Rules from User Behaviour,” in *Proc. of AIAI*, Boston, MA, 2009, vol. 296, pp. 459–468.

[16] T. Fahey, “Privacy and the Family: Conceptual and Empirical Reflections,” *Sociology*, vol. 29, no. 4, pp. 687–702, Nov. 1995.

[17] M. L. Yiu, C. S. Jensen, X. Huang, and H. Lu, “SpaceTwist: Managing the Trade-Offs Among Location Privacy, Query Performance, and Query Accuracy in Mobile Services,” in *Proc. of 24th Int. Conf. on Data Engineering*, 2008, pp. 366–375.

[18] L. Palen and P. Dourish, “Unpacking ‘privacy’ for a networked world,” in *Proc. of SIGCHI Conf. on Human factors in computing systems*, New York, NY, USA, 2003, pp. 129–136.

[19] T. D. Breaux and A. I. Anton, “Analyzing Regulatory Rules for Privacy and Security Requirements,” *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 5–20, Feb. 2008.

[20] E. Yu and L. M. Cysneiros, "Designing for Privacy and Other Competing Requirements," *Proc. of 2nd Symp. on RE for Inf. Security*, pp. 15–16, 2002.
[21] C. Mancini, K. Thomas, Y. Rogers, B. A. Price, L. Jedrzejczyk, A. K. Bandara, A. N. Joinson, and B. Nuseibeh, "From spaces to places: Emerging contexts in mobile privacy," in *Proc. of 11th Int. Conf. on Ubiquitous Computing*, 2009, pp. 1–10.
[22] N. Seyff, N. Maiden, K. Karlsen, J. Lockerbie, P. Grunbacher, F. Graf, and C. Ncube, "Exploring how to use scenarios to discover requirements," *Requir. Eng.*, vol. 14, no. 2, pp. 91–111, Apr. 2009.
[23] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," in *Proc. of 39th Symp. on Princ. of Prog. Lang.*, New York, NY, USA, 2012, pp. 85–96.

I. APPENDIX I: FORMALISATION OF THE FUNCTIONAL REQUIREMENT EXAMPLE

```
load foundations/Root.e
load foundations/EC.e

sort agents
sort subject: agents
sort user: agents
sort loc

user Alice
subject Bob
loc GPSPos_at_t

event WhereIs(subject)
event Tap(user,subject)
event Query(subject,loc)
event At(subject,loc)

;user knows that subject is at location
fluent Know(user, subject, loc)

;Location Display specification
[time,user,subject,loc]
Happens(Tap(user,subject),time) ->
(Happens(Query(subject,loc),time+1) &
Happens(At(subject,loc),time+2)).


;If the display shows that the subject is
;at a location, the user will know where
;the subject is
[time,subject,loc,user]
Initiates(At(subject,loc), Know(user, subject, loc),time).

;The requirement saying that tapping
;the subject (icon) leads to the
;user knowing the location of the subject
[time,user,subject,loc]
Happens(WhereIs(subject),time) ->
HoldsAt(Know(user, subject, loc),time+4).

;The following assertion checks
;whether the negation of the
;the requirement is satisfiable
;[time,user,subject,loc]
;Happens(WhereIs(subject),time) &
```

```
;!HoldsAt(Know(user, subject, loc),time+4).


;When the user wants to know the location of
;a subject, the user taps the icon of
;the subject
[time,user,subject]
Happens(WhereIs(subject),time) ->
Happens(Tap(user,subject),time+1).

;Initial states
; Alice wants to know where Bob is
Happens(WhereIs(Bob),0).

;Alice does not know where Bob is
!HoldsAt(Know(Alice, Bob, GPSPos_at_t),0).

;the following command tells the reasoner
;to perform deduction
completion Happens

range time 0 6
range offset 1 1
```

II. Appendix II: Formalisation of the positive norm example

```
load foundations/Root.e
load foundations/EC.e

sort agents
sort subject: agents
sort user: agents
sort loc

user Alice
subject Bob
loc GPSPos

event WhereIs(subject)
event Tap(user,subject)
event Query(subject,loc)
event At(subject,loc)

;user knows that subject is at location
fluent Know(user, subject, loc)

;user and subject has the same prefix
fluent SamePF(user,subject)

;user and subject are friends
fluent Friends(user,subject)

;If the display shows that the subject is
;at a location, the user will know where
;the subject is
[time,subject,loc,user]
Initiates(At(subject,loc), Know(user, subject, loc),time).
```

```
;The requirement saying that tapping
;the subject (icon) leads to the
;user knowing the location of the subject
[time,user,subject,loc]
Happens(WhereIs(subject),time) ->
HoldsAt(Know(user, subject, loc),time+4).

;The following assertion checks
;whether the negation of the
;the requirement is satisfiable
;[time,user,subject,loc]
;Happens(WhereIs(subject),time) &
;!HoldsAt(Know(user, subject, loc),time+4).


;When the user wants to know the location of
;a subject, the user taps the icon of
;the subject
[time,user,subject]
Happens(WhereIs(subject),time) ->
Happens(Tap(user,subject),time+1).

;CFN Specification
[time,user,subject,loc]
(HoldsAt(SamePF(user,subject),time) |
HoldsAt(Friends(user,subject),time)) &
(Happens(Tap(user,subject),time) ->
(Happens(Query(subject,loc),time+1) &
Happens(At(subject,loc),time+2))).


;Initial states
; Alice wants to know where Bob is
Happens(WhereIs(Bob),0).

;Alice does not know where Bob is
!HoldsAt(Know(Alice, Bob, GPSPos),0).

;Alice and Bob have some prefixes in their IDs
HoldsAt(SamePF(Alice,Bob),0).
;If the above statement is negated,
;then the query will not return the location

;Alice and Bob are not friends
!HoldsAt(Friends(Alice,Bob),0).

;the following command tells the reasoner
;to perform deduction
completion Happens

range time 0 6
range offset 1 1
```

III. APPENDIX III: FORMALISATION OF THE NEGATIVE NORM EXAMPLE

```
load foundations/Root.e
```

```
load foundations/EC.e

sort agents
sort subject: agents
sort user: agents
sort loc

user Alice
subject Bob
loc GPSPos

event WhereIs(subject)
event Tap(user,subject)
event Query(user,subject,loc)
event At(subject,loc)

;user knows that subject is at location
fluent Know(user, subject, loc)

;user and subject has the same prefix
fluent SamePF(user,subject)

;user and subject are friends
fluent Friends(user,subject)

;If the display shows that the subject is
;at a location, the user will know where
;the subject is
[time,subject,loc,user]
Initiates(At(subject,loc), Know(user, subject, loc),time).

;The requirement saying that tapping
;the subject (icon) leads to the
;user knowing the location of the subject
[time,user,subject,loc]
Happens(WhereIs(subject),time) ->
HoldsAt(Know(user, subject, loc),time+4).


;When the user wants to know the location of
;a subject, the user taps the icon of
;the subject
[time,user,subject]
Happens(WhereIs(subject),time) ->
Happens(Tap(user,subject),time+1).

;CFN Specification - Negative Norm
[time,user,subject,loc]
Happens(Tap(user,subject),time) ->
(Happens(Query(user,subject,loc),time+1) &
Happens(At(subject,loc),time+2) &
(HoldsAt(SamePF(user,subject),time) |
HoldsAt(Friends(user,subject),time))).

;Initial states
; Alice wants to know where Bob is
Happens(WhereIs(Bob),0).
```

```
;Alice does not know where Bob is
!HoldsAt(Know(Alice, Bob, GPSPos),0).

;Alice and Bob have some prefixes in their IDs
HoldsAt(SamePF(Alice,Bob),0).

;Alice and Bob are not friends
!HoldsAt(Friends(Alice,Bob),0).

;the following command tells the reasoner
;to perform deduction
completion Happens

range time 0 6
range offset 1 1
```

IV. Appendix IV: Formalisation of the exception example

```
load foundations/Root.e
load foundations/EC.e

sort agents
sort subject: agents
sort user: agents
sort loc

user Bob
subject Dave
loc GPSPos

event WhereIs(subject)
event Tap(user,subject)
event Query(subject,loc)
event At(subject,loc)

;user knows that subject is at location
fluent Know(user, subject, loc)

;user and subject has the same prefix
fluent SamePF(user,subject)

;user and subject are friends
fluent Friends(user,subject)

;abnormality predicate
predicate Ab1(subject,user,time)

;If the display shows that the subject is
;at a location, the user will know where
;the subject is
[time,subject,loc,user]
Initiates(At(subject,loc), Know(user, subject, loc),time).

;The requirement saying that tapping
;the subject (icon) leads to the
```

```
;user knowing the location of the subject
[time,user,subject,loc]
Happens(WhereIs(subject),time) ->
HoldsAt(Know(user, subject, loc),time+4).


;When the user wants to know the location of
;a subject, the user taps the icon of
;the subject
[time,user,subject]
Happens(WhereIs(subject),time) ->
Happens(Tap(user,subject),time+1).

;CFN Specification
[time,user,subject,loc]
( ( HoldsAt(SamePF(user,subject),time) |
HoldsAt(Friends(user,subject),time) ) &
Happens(Tap(user,subject),time) &
!Ab1(subject,user,time) ) ->
(Happens(Query(subject,loc),time+1) &
Happens(At(subject,loc),time+2)).

;Initial states
; Bob wants to know where Dave is
Happens(WhereIs(Dave),0).

;Bob does not know where Dave is
!HoldsAt(Know(Bob, Dave, GPSPos),0).

;Bob and Dave have some prefixes in their IDs
HoldsAt(SamePF(Bob,Dave),0).

;Bob and Dave are not friends
!HoldsAt(Friends(Bob,Dave),0).

;Dave does not want Bob to know his location
[time] Ab1(Dave,Bob,time).

;the following command tells the reasoner
;to perform deduction
completion Happens

range time 0 6
range offset 1 1
```