Reputation-Based Trust Management [*]

Vitaly Shmatikov and Carolyn Talcott

Computer Science Laboratory
SRI International
Menlo Park, CA 94025 USA
{shmat,clt}@csl.sri.com

Abstract

We propose a formal model for reputation-based trust management. In contrast to credential-based trust management, in our framework an agent's *reputation* serves as the basis for trust. For example, an access control policy may consider the agent's reputation when deciding whether to offer him a license for accessing a protected resource. The underlying semantic model is an event semantics inspired by the actor model, and assumes that each agent has only partial knowledge of the events that have occurred. Restrictions on agents' behavior are formalized as *licenses*, with "good" and "bad" behavior interpreted as, respectively, license fulfillment and violation. An agent's reputation comprises four kinds of evidence: completely fulfilled licenses, ongoing licenses without violations or misuses, licenses with violated obligations, and misused licenses. This approach enables precise formal modeling of scenarios involving reputations, such as financial transactions based on credit histories and information sharing between untrusted agents.

1 Introduction

Reputation is a fundamental concept in many situations that involve interaction between mutually distrusting parties. Before issuing a credit card, a bank usually checks the applicant's credit history, which includes independently certified evidence that the applicant has fulfilled his prior financial obligations. At Internet auction sites such as eBay, the seller's reputation, *i.e.*, the evidence that past buyers were satisfied with his or her behavior, is considered an asset of great value. Many solutions for the "free-rider problem" [2] in peer-to-peer file sharing networks (the problem of users downloading a large number of files without contributing anything in return) rely on the evidence of past contributions when granting access to popular files [12].

We propose a formal model that gives a precise meaning to the notion of reputation and uses it in reasoning about trust. Our approach extends an event-based semantics inspired by the actor model of distributed computation [5, 10] to incorporate *incomplete*

[*] Supported by ONR grant N00014-01-1-0837.

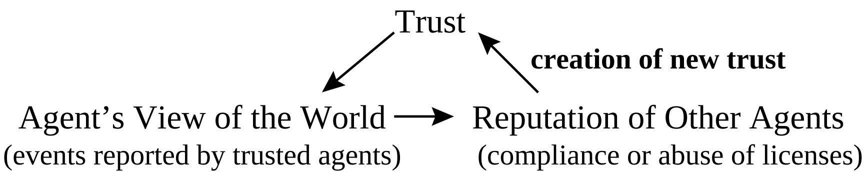
Trust
creation of new trust
Agent's View of the World
(events reported by trusted agents)
Reputation of Other Agents
(compliance or abuse of licenses)

Figure 1: Reputation-based trust management

knowledge. Agents are assumed to have only partial knowledge of the event history. When deciding whether to trust another agent or not, they rely on the evidence of past behavior supplied by trusted sources. By contrast, trust in conventional trust management [6] is based on access control credentials.

Inspired by license-based digital rights languages [13, 16], we use licenses to formalize both "good" and "bad" behavior. Each license restricts the behavior of the agent who accepts it by specifying obligations (what the agent *must* do) as well as forbidden actions (what the agent *must not* do). Evidence of compliance or violation then becomes part of the licensee's reputation. Another agent may decide to grant a new license on the basis of this evidence, even though he has not personally observed the licensee's good or bad behavior.

To illustrate by example, consider a consumer applying for an auto loan. The lender requests the consumer's credit history from a credit reporting bureau, and uses the information to decide whether to grant the loan and on what terms. The lender's trust in the consumer is not based on the consumer's identity credentials and, in contrast to conventional trust management, the credit bureau is not vouching for the consumer's creditworthiness (*i.e.*, there is no delegation of trust). The bank trusts the credit bureau to accurately report the *evidence*, *i.e.*, a summary of past events, such as the fact that the consumer had signed up for a credit card and fulfilled his obligations by making timely payments.

Our general approach to trust management is summarized in figure 1. We formalize our framework in an object-oriented "language," which is mapped to the rewriting logic based system, MAUDE [9]. Rewriting logic is a logical formalism well-suited for modeling and reasoning about concurrent and interactive systems [15]. It is based on two simple ideas: states of a system are represented as elements of an algebraic data type, and the behavior of a system is given by localized transitions between states described as abstractions called rewrite rules. MAUDE provides several strategies for simulating system behavior as well as search and model-checking capabilities for exploring the reachable state space. The MAUDE system implementation, documentation, examples, and related papers can be found on the MAUDE website `http://maude.cs.uiuc.edu`.

The main benefit of this approach is that it enables formal verification of reputations and trust policies. We can also use MAUDE to explore the dynamic aspects of any given configuration of trust management policies as well as their evolution over time. For example, we can obtain answers to questions such as "If a certain event occurs, what will be its impact on agents' reputations?," and "If some agent switches to a particular policy for granting new trust on the basis of reputation, how will this affect other trust relationships in the system?"

We also present two case studies, using our framework to formalize reputation-based trust in an anonymized peer-to-peer file distribution system and a multi-player game scenario, respectively. The former, described using a semi-formal object-oriented language, demonstrates how reputation is created from the evidence of license fulfillment, while the latter, presented as a MAUDE specification, illustrates the use of licenses in an environment with multiple untrusted agents.

The structure of the paper is as follows. In section 2, we introduce the basic concepts of our framework. We formalize licenses and reputations in sections 3 and 4, respectively. In section 5, we describe how reputations are formed on the basis of an agent's past behavior. In section 6, we present the peer-to-peer file distribution network example, and in section 7, the multi-player game example. We survey related work in section 8. Conclusions follow in section 9.

2 Model

We use a semi-formal "object-oriented" notation to present the constructs of our framework. After introducing basic types, we define concepts such as *event*, *license*, *principal*, *etc.* as object types with fields and methods. Fields may be interpreted simply as the corresponding object's data structures, and the methods as operations on these data structures. Some methods should be interpreted as messages as they correspond to communication between active objects rather than direct operations on the object structure. The framework is presented using an informal mathematical notation. Select segments of the MAUDE representation are included to give a flavor of the formal notation.

For some methods, we specify the semantics that we expect any policy to satisfy (unless stated otherwise, as in the example of section 6). For other methods, we deliberately leave the semantics unspecified. If the semantics is not specified, the method is unconstrained: policy and license writers are free to choose any semantics depending on their preferred trust policy, desired security properties, and so on. We use a special ***font*** for the names of unconstrained methods.

Our types are intended to serve as the roots of a type hierarchy in any actual implementation of a trust management system based on our framework. For example, we define a general-purpose *License* type, but only provide a type signature for the ***violated*** method, without specifying precisely what it means for a license to be violated. A specific policy — for example, in the context of an *OnlineAuction* trust management system — will refine our definition by introducing new types, such as *SellerLicense* and *BuyerLicense*, which are subtypes of *License* and supply semantics for the methods left unspecified in our *License* definition. Other policies may refine the semantics of *License* in different ways. In our examples, we use "A refines B" syntax to indicate that A is a refinement of B.

We use the following syntax for type definitions:

ObjectType
field $\mathit{FieldName}_1$: Type_1
. . .
field $\mathit{FieldName}_k$: Type_k
method $\mathit{MethodName}_{k+1}$ (Type^1_{k+1},. . . ,$\mathit{Type}^{r_{k+1}}_{k+1}$):$\mathit{Type}_{k+1}$ [= $\langle$*semantics*$\rangle$]
. . .
method $\mathit{MethodName}_n$ (Type^1_n,. . . ,$\mathit{Type}^{r_n}_n$): Type_n [= $\langle$*semantics*$\rangle$]

If we specify the field type as $?\mathit{Type}_i$ where Type_i is some type, this means that the value of the field may be undefined, but when it is defined, it must be of type Type_i.

In MAUDE, types are called *sorts* and we use mixfix notation to represent field selection. The corresponding declaration of an object type in MAUDE has the form

```
sort <ObjectType> .
op _. <FieldName_1> : <ObjectType> -> Type_1 .
...
op _. <FieldName_k> : <ObjectType> -> Type_k .

op _. <MethodName_k+1>(_,...,_) :
     <ObjectType> Type_k+1,1 ... Type_k+1,r_k+1 -> Type_k+1 .
...
```

declaring `<ObjectType>` to be a sort, and declaring operators for the fields and methods. The underscores (_) give the argument positions in a mixfix declaration.

To specify creation of a new object of type α, we use the following syntax:

new α($\mathit{FieldName}_1 \leftarrow \mathit{value}_1$, . . . , $\mathit{FieldName}_k \leftarrow \mathit{value}_k$)

where value_i is the initial value for the field $\mathit{FieldName}_i$. If we refer to the object's own methods or fields when specifying some method's semantics, we use "this" keyword as in C++. Finally, we use "let v = *value* in . . . " syntax for standard scoping.

2.1 Basic types

Our framework contains the following types with an infinite number of values:

Natural	Natural numbers
UniqueName	Unique agent names
Timestamp	Ordered time values
LicenseId	Unique license identifiers

Naming and authentication issues lie outside the scope of this paper. We assume that each agent is assigned a unique name, and that the communication medium used for information exchange between agents implements an appropriate authentication mechanism, thus ensuring that agents are not confused about each other's identity.

We also define the following enumerated types:

Bool = { True, False }
ActionType = { ResourceUse, LicenseOffer, LicenseAccept, ... }
ResourceStatus = { Public, Licensed, Protected }
LicenseKind = { ... } (user-defined, *e.g.*, eBaySeller)

We define the standard polymorphic set type:

Set α Set of values of type α

We define the standard projection and size operations on sets:

project(S:*Set* α,P:$\alpha\rightarrow$*Bool*): *Set* α = $\{s \in S \mid P(s) =$ True$\}$
size(*Set* α): *Natural*

To define the projection predicate P:$\alpha\rightarrow$*Bool*, we use standard λ-expressions. For example, *project*(globalHistory, $\lambda\ e.\ e$.*action*.*type*=LicenseAccept) extracts from globalHistory only the events in which some license was accepted.

2.2 Actions and events

An *action* is an atomic interaction between several agents, or an agent and a resource. We'll use the terms *principal* and *agent* interchangeably.

Action
field *type*: *ActionType*
field *actors*: *Set UniqueName*
field *subject*: *?Resource*
field *license*: *?LicenseId*

The *actors* field specifies agents involved in the action, *subject* specifies the resource (if any), and *license* specifies the identifier of the license associated with the action (if any). For example, an object a:*Action* such that a.*type*=ResourceUse, a.*actors*= $\{Pirate99\}$, a.*subject*=$HackedTunez$, a.*license*=L_{7198} models an agent called $Pirate99$ using a resource called $HackedTunez$ under a license whose identifier is L_{7198}.

The following is the MAUDE declaration of the *Action* object type.

```
sort Action .
op _. type : Action -> ActionType .
op _. actors : Action -> UNameSet .
op _. subject : Action ~> Resource .
op _. license : Action ~> LicenseId .
```

The possibility that a field value is undefined is represented by declaring the corresponding operator to be a partial function (indicate by ~>). MAUDE supports partial functions by associating a *Kind* (think "error" sort) with each sort, to represent error or undefined terms.

When constructing each agent's partial view of the event history, we assume that only agents listed in the *actors* field have direct knowledge of the event in which the action has occurred. We also provide a mechanism for agents to share information about events.

An *event* is a time-stamped action occurrence.

Event
  field *time*: *Timestamp*   (not necessarily unique for each event)
  field *action*: *Action*
  method <(e:*Event*): *Bool* = this.*time* $<$ e.*time*

To simplify presentation, we use infix notation for $<$. Because two events may have the same timestamp, $<$ defines only a partial order on events. We define event *history* simply as a set of events.

EventHistory = *Set Event*

2.3 Resources

A *resource* is an item of value (program, website, database, credential) that agents may wish to access or use.

Resource
  field *owner*: *?UniqueName*
  field *status*: *ResourceStatus*
  method ***useOk***(*EventHistory*,*License*): *Bool*

A resource may or may not have an owner. If there is an owner, he or she specifies how the resource may be used by defining the *useOk* method. If r.*useOk*(h,l)=True for some resource r, event history h, and license l, this is interpreted as "given event history h, the resource r may be used (accessed) on the basis of license l." The owner is free to define *useOk* in any way he or she sees fit. Normally, the definition of *useOk* depends on the access policy for the resource and involves checking whether the license l is valid (see section 3). In general, *useOk* defines both permissible and (via its negation) forbidden uses of the resource.

The status of the resource is Public, Licensed, or Protected. The owner must be specified for a Licensed or Protected resource. Given a resource r, if r.*status*=Public, then there should be no restrictions on the use of the resource. Ownership of r does not matter in this case, and it is assumed that r.*useOk*(h,l)=True for any event history h and any license l.

If r.*status*=Licensed, using the resource is always feasible, *i.e.*, the event history may contain an action a such that a.*type*=ResourceUse, a.*subject*=r and a.license=l for an arbitrary license l. If, however, r.*useOk*(h,l)=False, then such action constitutes a misuse of license l (see section 3).

If r.*status*=Protected, then using the resource is impossible if r.*useOk*(h,l)=False, *i.e.*, the corresponding ResourceUse event may not appear in the event history. This is used, for example, to model cryptographically protected resources, which cannot be feasibly accessed unless the license contains the right key.

Intuitively, the difference between Licensed and Protected is that *useOk* for Licensed resources simply determines whether the use of the resource is "good" or "bad," while *useOk* for Protected resources makes "bad" uses computationally infeasible. As explained in section 3, there are *two* kinds of misuse in our model: an action forbidden by the license, and an action permitted by the license but forbidden by the resource itself. The latter is only meaningful if the resource is Licensed. We do not treat a failed attempt to use a Protected resource without a proper license as a misuse.

For simplicity, neither ownership, nor status depends on event history. In a more sophisticated framework, *owner* and *status* may be viewed as updatable fields, whose values change as the system evolves.

3 Licenses

We use a license language inspired by [13, 16] to define the permissible behavior of agents. Licenses are very important in our framework since compliance with past licenses is the basis of an agent's reputation. Each license defines (i) what the licensee is permitted to do with the license, and (ii) the licensee's obligations.

```
License
  field id: LicenseId
  field kind: LicenseKind
  field issuer: UniqueName
  field licensee: UniqueName
  field resource: Resource
  method permits(EventHistory,Event): Bool
  method violated(EventHistory): Bool
  method misuse($h$:EventHistory,$e$:Event): Bool =
      (not permitted by the license itself)
    $\neg$ this.permits($h$,$e$) $\vee$
      (... or not permitted by the resource being used)
    $e$.action.license = this.id $\wedge$ this.licensee $\in e$.action.actors $\wedge$
    $e$.action.type = ResourceUse $\wedge$ $\neg$ $e$.action.subject.useOk($h$,this)
  method valid(h:EventHistory): Bool =
    $\exists e_1, e_2 \in h$ such that $e_1 < e_2$ $\wedge$
    $e_1$.action.type = LicenseOffer $\wedge$ $e_1$.action.license = this.id $\wedge$
    $\{$this.issuer, this.licensee$\} \subseteq e_1$.action.actors $\wedge$
    $e_2$.action.type = LicenseAccept $\wedge$ $e_2$.action.license = this.id $\wedge$
    $\{$this.issuer, this.licensee$\} \subseteq e_2$.action.actors
  method done(EventHistory): Bool
```

Given a license l, l.*issuer* identifies the agent who issued the license, l.*licensee* identifies the licensee, *i.e.*, the agent to whom obligations and permissions apply, and l.*resource* identifies the subject of the license.

A license is *valid* in some event history if it has been offered by the issuer and accepted by the licensee. Validity is monotonic: if $h \subseteq h'$, then l.*valid*(h) implies l.*valid*(h'). If the issuer wants the license to expire at some point (*e.g.*, once a certain time has been announced, or a particular combination of events has occurred), he or she can do this by defining a *done* method. If l.*done*(h) returns True on some event history h, this is interpreted as stating that license l has expired in history h. While the license issuer is free to choose any semantics for the *done* method, most licensees will expect that expiration is monotonic: if $h \subseteq h'$, then l.*done*(h) implies l.*done*(h'). Note that if $\neg$ l.*valid*(h), then l.*done*(h) is meaningless.

Permissions. By defining the *permits* method, the license issuer specifies the set of actions permitted by the license. There are no *a priori* constraints on the semantics of this method. If an action is not permitted, the licensee may still be able to perform it, but the corresponding event will be considered a misuse of the license and reported as such in the licensee's reputation.

We'd like to emphasize the difference between *useOk* methods, which are associated with resources, and *permits* methods, which are associated with licenses. This distinction is necessary in complex licensing scenarios, where it is difficult for the license issuer to foresee all possible situations in which the licensee may attempt to use the license. Consider, for example, a driver's license which entitles the holder to operate cars, but not motorcycles. The no-motorcycles restriction is explicitly spelled out in the driver licensing code, which can be thought of as the *permits* method associated with the driver's license. Note, however, that a typical licensing code does *not* contain an explicit prohibition on using the license to operate airplanes. In general, many improper uses of the license cannot be envisioned by the license issuer and are thus not spelled out as part of the license terms. Preventing such uses is the responsibility of the resource owners (in our example, the civil aviation authority). Therefore, the corresponding prohibitions should be encoded as part of the *useOk* method associated with each resource.

Intuitively, the *useOk* method specifies the access policy from the viewpoint of the *resource owner*. Therefore, it applies in the same way to all licenses. The resource owner is free to ignore permissions stipulated in the license, or else consider them by invoking the license's *permits* method when deciding whether to grant access to the license holder (as in the example of section 6).

On the other hand, the *permits* method is used by the *license issuer* to specify what he or she views as permissible behavior. Different licenses for the same resource may thus contain different permissions. The *permits* method may take into account the resource's access policy by invoking the *useOk* method of the resource (as in the example of section 7). In the extreme case, the same action may be permitted by the resource owner and forbidden by the license issuer, or vice versa.

Obligations. By defining the *violated* method, the license issuer specifies the obligations imposed on the licensee by the license. Formally, if l.*violated*(h)=True for some license l and event history h, this means that h does *not* contain the fulfillment of every obligation imposed by the license. For example, if license l models taking a loan, then l.*violated*(h)=True if h contains a timestamp event corresponding to the repayment deadline, but does not contain a preceding repayment event.

The license issuer is free to define the *violated* method as he wishes, but a well-defined license cannot be violated unless it has been accepted by the licensee. Formally, this means that $\neg$ l.*valid*(h) should imply $\neg$ l.*violated*(h).

We say that the licensee *partially fulfilled* the license if, up to date, he has performed all obligations specified by the license, but the license has not expired yet and may contain future obligations. There is thus a possibility that the licensee will violate the license in the future. If the licensee has fulfilled the license up to date and there are no future obligations, we say that the license is *completely fulfilled*.

Violation and misuse. Permissions associated with the license are independent of the obligations. A license circumscribes the licensee's behavior "from above" (*permits* restricts the set of actions he *may* do) as well as from "from below" (*violated* specifies what he *must* do). A license may be passively violated by *not* doing something (*e.g.*, not repaying a loan), or actively misused by doing something forbidden (*e.g.*, overdrawing a credit line), or both. We will call the former a *violation*, and the latter a *misuse*. Another form of misuse is one that is not associated with license terms at all. It occurs when the license is used to perform an action on a resource which is permitted by the license, but is forbidden by the resource owner.

The license issuer explicitly specifies what constitutes a violation (*i.e.*, an unfulfilled obligation) by defining the *violated* method. The *misuse* method is defined automatically by combining actions forbidden by the license itself (*permits* method returns False on the corresponding events) and those forbidden by resource owners (*useOk* method returns False). Having a single *misuse* method is a design decision. One could also envision a framework in which the two forms of misuse are separated, thus differentiating license misuse (performing an action forbidden by the license issuer) from resource misuse (performing an action forbidden by the resource owner). We plan to investigate this framework in other application scenarios.

The definition of the *misuse* method in MAUDE is the following direct mapping of the mathematical notation.

```
op _. misuse(_,_) : License EventHistory Event -> Bool .
eq lic . misuse(h, e) =
  not(lic . permits(h,e))
  or
  ( e . action . license == lic . id
    and
    e . action . type == ResourceUse
    and
    e . action . actors ==  lic . licensee
    and
    not( (e . action . subject) . useOk(h,lic)) )
.
```

License issuers are free to define *violated* and *permits* methods in any way they wish, and the same is true for resource owners and *useOk* methods, respectively. Our framework *per se* does not enforce any consistency checks on these predicates. It is up to each licensee to decide whether the restrictions encoded in the license and the resource description are acceptable. These restrictions may even be inconsistent. Our objective is not to ensure that every license is meaningful or to prevent incompetence in license writing. Instead, we develop a general framework that supports any access policy and any license terms as specified by the resource owners and license issuers.

4 Reputation

An agent's *reputation* in our framework is simply the evidence of the agent's past behavior. Since we rely on fulfillment, violation, and misuse of licenses to give meaning to "good" and "bad" behavior, each piece of evidence consists of a license identifier together with the supporting event history. When the evidence is distributed as part of some agent's reputation, this history can be used by other agents to verify whether the reputation is accurate. For instance, when the reputation claims that the agent has violated some license, this event history can be used to verify that obligations associated with the license in question have indeed been left unfulfilled.

Evidence
field *license*: *LicenseId*
field *justification*: *EventHistory*

In scenarios where privacy and anonymity are important (such as that described in section 6), the event history listed in the *justification* field may be left incomplete or even empty. In this case, an agent who analyzes another agent's reputation can still identify each piece of evidence by the corresponding license id, but *cannot* verify that the reputation is accurate by inspecting the supporting event history, since this history is not made available as part of the evidence.

EvidenceSet = *Set Evidence*

Reputation is a tuple of four evidence sets, corresponding to four kinds of behavior that are used as the basis for reputation. The sets are, respectively, completely fulfilled licenses, partially fulfilled licenses (*i.e.*, licenses that are fulfilled to date but have outstanding future obligations), violations (*i.e.*, licenses with unfulfilled obligations), and misuses (*i.e.*, licenses used to perform an action which is forbidden either by the license issuer, or by the resource owner).

Reputation
field *name*: *UniqueName*
field *compFulfilled*: *EvidenceSet*
field *partFulfilled*: *EvidenceSet*
field *violations*: *EvidenceSet*
field *misuses*: *EvidenceSet*

We also define some auxiliary functions:

mergeEvidenceSets(e_1:*EvidenceSet*, e_2:*EvidenceSet*): *EvidenceSet* =
$\{ ev \mid ev \in e_1 \cup e_2 \wedge \nexists\, ev' \in e_1 \cup e_2$ s.t. $ev' \neq ev \wedge ev$.*license*=ev'.*license* $\})$ $\bigcup$
$\{$ new*Evidence*(*license* $\leftarrow ev$.*license*,
justification $\leftarrow ev$.*justification* $\cup\, ev'$.*justification*) $\mid$
$ev, ev' \in e_1 \cup e_2 \wedge ev' \neq ev \wedge ev$.*license*=$ev'$.*license* $\}$

```
mergeReputations($r_1$:Reputation, $r_2$:Reputation): Reputation =
  new Reputation(
    name $\leftarrow r_1$.name,        (not well-defined if $r_1$.name $\neq r_2$.name)
    compFulfilled $\leftarrow$ mergeEvidenceSets($r_1$.compFulfilled, $r_2$.compFulfilled),
    partFulfilled $\leftarrow$ mergeEvidenceSets($r_1$.partFulfilled, $r_2$.partFulfilled),
    violations $\leftarrow$ mergeEvidenceSets($r_1$.violations, $r_2$.violations),
    misuses $\leftarrow$ mergeEvidenceSets($r_1$.misuses, $r_2$.misuses) )
```

5 Reputation management

We are now ready to describe how reputations are formed, and how they are used by agents to construct trust relationships with each other. We do this by defining a *Principal* object type. The types of all agents in the system are intended to be subtypes of the *Principal* type.

```
Principal
  field name: UniqueName
  field licenses: Set License
  field trusted: Set UniqueName
  method exportEvents(): EventHistory
  method view(): EventHistory =
    project(globalHistory, $\lambda\ e.$ this.name $\in e$.action.actors) $\bigcup$
    $p$.exportEvents() $\forall p$ such that $p$.name $\in$ this.trusted
  method localRep($a$: UniqueName): Reputation =
    let $v$=this.view() in
    let $lsa$={ $l \in$ this.licenses $\mid l$.licensee=$a$ } in
    new Reputation(name $\leftarrow a$,
      compFulfilled $\leftarrow cf$, partFulfilled $\leftarrow pf$,
      violations $\leftarrow uo$, misuses $\leftarrow mu$)
    where
  (completely fulfilled licenses:)
    $cf$ = {
      new Evidence(license $\leftarrow l$.id, justification $\leftarrow le$) $\mid l \in lsa \wedge$
          (license is fulfilled and there are no future obligations)
      $l$.valid($v$) $\wedge \neg\ l$.violated($v$) $\wedge\ l$.done($v$) $\wedge$
          (evidence may be incomplete)
      $le \subseteq$ project($v$, $\lambda\ e.\ e$.action.license=$l$.id)
    },
  (partially fulfilled licenses:)
    $pf$ = {
      new Evidence(license $\leftarrow l$.id, justification $\leftarrow le$) $\mid l \in lsa \wedge$
          (license is fulfilled but there may be future obligations)
      $l$.valid($v$) $\wedge \neg\ l$.violated($v$) $\wedge \neg\ l$.done($v$) $\wedge$
          (evidence may be incomplete)
      $le \subseteq$ project($v$, $\lambda\ e.\ e$.action.license=$l$.id)
    },
  (licenses with unfulfilled obligations:)
    $uo$ = {
      new Evidence(license ← $l$.id, justification ← $le$) | $l \in lsa \wedge$
          (obligations associated with the license are violated)
      $l$.valid($v$) $\wedge$ $l$.violated($v$) $\wedge$
          (evidence may be incomplete)
      $le \subseteq$ project($v$, $\lambda\ e.\ e$.action.license=$l$.id)
    },
  (misused licenses:)
    $mu$ = {
      new Evidence(license ← $l$.id, justification ← $le$) | $l \in lsa \wedge$
          (history contains a misuse event)
      $\exists e \in v$ such that $l$.misuse($h$,$e$)
          (evidence may be incomplete)
      $le \subseteq$ project($v$, $\lambda\ e.\ e$.action.license=$l$.id)
    }
    method exportGlobalRep($a$:UniqueName, $excl$:Set UniqueName): Reputation =
      mergeReputations(this.localRep($a$), $p$.exportGlobalRep($a$, $excl \cup$ this.name))
      $\forall p$ such that $p$.name $\in$ this.trusted $\wedge$ $p$.name $\notin$ $excl$
    method isTrusted(Reputation): Bool
```

We explain the fields and methods of *Principal* in the order they are defined. The *name* field contains the principal's name. The *licenses* field contains the licenses issued by the principal. The *trusted* field contains the names of other agents trusted by this principal. When the principal computes the reputation of some agent, in addition to the local information he also collects evidence from the agents listed in the *trusted* field. The *trusted* field may be updated by the *isTrusted* method, which is used by the principal to encode the trust creation policy and to decide when to trust another agent on the basis of his or her reputation.

Information sharing between agents. Method *exportEvents* is used by the principal to give other agents information about the events known to him. Together with *exportGlobalRep*, which is used to share information about reputations, *exportEvents* is the basis of information sharing in our framework. We impose no constraints on the semantics of the *exportEvents* method. A principal is free to export as much or as little information as he wishes. A malicious principal may even lie and export event histories that do not correspond to actual events. Our aim is to enable realistic information sharing between independent agents, which sometimes includes lying. In MAUDE, the *exportGlobalRep* method is implemented using message passing. In particular, the clause $\forall p$ such that p.*name* $\in$ this.*trusted* quantifying the recursive *exportGlobalRep* invocation becomes a multicast of *exportGlobalRep* messages addressed to unique names from the *trusted* set that aren't excluded.

Partial view of event history. The *view* method simply computes the entire event history known to the principal by combining the events that he knows about by virtue of having been one of the actors in the corresponding action with the events reported by trusted agents (and resolving inconsistencies, if necessary). Technically, we assume that there exists a single unified global history of all events that have occurred. We refer to this history as globalHistory. We use standard set projection to restrict this history to the events in which the principal participated directly. Information about other events is obtained from trusted sources by invoking their *exportEvents* methods (implemented in MAUDE as messages). As explained above, other agents are free to decide which events to share by defining the *exportEvents* method accordingly.

In the actual MAUDE specification, *view* is implemented as an updatable field rather than a method. Its value is updated every time an event occurs in which the principal is one of the participants, and when *exportEvents* message arrives from one of the principals in the *trusted* set.

Construction of reputation. The *localRep* method is used by the principal to construct *another* agent's reputation on the basis of the event history known to the principal. This method is the core of our framework. An agent's reputation comprises four parts, containing the evidence of, respectively, completely fulfilled licenses, partially fulfilled licenses, licenses with violated obligations, and misused licenses. Since permissions associated with a license are independent of the obligations (see section 3), the same license identifier may appear in several components of the reputation.

For each license used as evidence, the supporting event history may be included as *justification*. We do not require that the evidence include the entire event history associated with a particular license. To preserve privacy and anonymity, it may be necessary for the reputation object to hide all information about the events on which the agent's reputation is based (for an example of this, see section 6). If justification *is* included, then other agents may use it to verify that the reputation is accurate. For example, if the reputation object claims that the agent has misused some license, the event history supplied as part of the evidence object may be inspected to confirm that it indeed contains a misuse event.

For completely fulfilled, partially fulfilled, and violated licenses, the license must be valid in the event history (*i.e.,* offered by the issuer and accepted by the licensee) in order to be used as evidence. The only exception is misused licenses, since an agent may misuse a license (*e.g.,* by improperly using it to gain access to a licensed resource) even if he has not accepted its terms.

Reputation depends on the principal's partial view of the event history. New events, either observed directly by the principal, or reported by trusted agents via the *exportEvents* method, may cause the reputation to change. In particular, unfulfilled obligations (*e.g.*, absence of a promised payment) may be rectified by presenting the evidence (*e.g.*, a signed bank statement) that the event fulfilling the obligation has occurred. In this case, the license and the supporting evidence will move from the *violations* component of the reputation to the *compFulfilled* or *partFulfilled* component. While *violations* are based on the absence of a fulfilling event, *misuses* are based on the presence of a specific misuse event, and typically cannot be rectified, unless *permits* or *useOk* methods are nonmonotonic.

The *exportGlobalRep* method combines the principal's local reputation for another agent with the same agent's reputations reported by trusted sources. The $excl$ argument is necessary to avoid infinite recursion when constructing the spanning tree of the trust relationships graph.

Creation of new trust. Given an agent's reputation, the *isTrusted* method decides whether the agent should be trusted and, if so, whether he or she should be offered some license and/or his or her name added to the *trusted* list. We impose no constraints on the semantics of the *isTrusted* method. The policy writer is free to choose any local reputation-based trust policy for a given principal depending on the specific requirements and security objectives.

6 Example: Peer-to-Peer File Distribution System

In this case study, we use our framework to encode a reputation-based trust management policy for a peer-to-peer file distribution system, roughly similar to Gnutella or Freenet [8], implemented on top of a network of anonymizing routers such as an onion routing network [18].

The following types model generic upload and download resources:

```
Upload refines Resource
  field status: ResourceStatus = Public
  method useOk($h$:EventHistory,$l$:License): Bool = True
```

```
Download refines Resource
  field status: ResourceStatus = Licensed
  method useOk($h$:EventHistory,$l$:License): Bool =
    let $ua$ = new Action(
          type ← ResourceUse, actors ← { $l$.licensee },
          subject ← this, license ← $l$ ) in
    let $e$ = new Event(time ← new Timestamp, action ← $ua$) in
    $l$.valid($h$) $\wedge$ $l$.issuer=this.owner $\wedge$ $l$.permits($h$,$e$)
```

Intuitively, these definitions state that anybody can upload (since resources of type *Upload* are public), but downloads, modeled as uses of a *Download* resource, are only permitted with a license issued by the server's owner. Moreover, *useOk* checks whether, given an event history, the use is permitted by the license.

Consider a single file server consisting of two resources, dl and ul, of type *Download* and *Upload*, respectively, where dl and ul are unique resource identifiers. Also, assume that there exists a *lookupPrincipal*(*UniqueName*): *Principal* table that maps principals' names to the corresponding objects of type *Principal.*

A sample license for the dl resource can be defined as follows:

```
DownloadLicense refines License
  field resource: ResourceId = $dl$
  method done(EventHistory): Bool =
    $nU \geq 5$
  method permits($h$:EventHistory,$e$:Event): Bool =
    isDownload($e$) $\wedge$
    ($nD < nU \times 3 \vee p$.isTrusted($p$.exportGlobalRep(this.licensee,$\{\}$))
  method violated(EventHistory): Bool =
    $nD > nU \times 2$
  where
    $p$=lookupPrincipal($dl$.owner),
    isDownload(Event): Bool = $\lambda\ e$.
      ( $e$.action.type = ResourceUse $\wedge$ $e$.action.actors = $\{$ this.licensee $\}$ $\wedge$
      $e$.action.subject = $dl$ ),
    isUpload(Event): Bool = $\lambda\ e$.
      ( $e$.action.type = ResourceUse $\wedge$ $e$.action.actors = $\{$ this.licensee $\}$ $\wedge$
      $e$.action.subject = $ul$ ),
    $nD$=size(project($h$, isDownload)),
    $nU$=size(project($h$, isUpload))
```

By accepting this license, the licensee promises to upload at least once for every 2 downloads. If he fails to do this, however, he is not prevented from further downloads as long as the event history contains at least 1 upload for every 3 downloads. This means that *permits* allows some violating actions (*i.e.*, obligations accepted by the licensee may be left unfulfilled to a limited extent). For example, if 2 uploads and 4 downloads have occurred, the 5th download will be allowed ($5 < 2 \times 3$), even though it's a violation of the obligation ($5 > 2 \times 2$), but the 6th download will not be allowed.

Note that *permits* allows unlimited downloads if the licensee is trusted by the resource owner on the basis of his reputation (*i.e.*, the resource owner's *isTrusted* method evaluates to True). The licensee can thus gain access to dl in two ways: by maintaining the proper ratio of uploads to downloads, or by relying on a previously acquired reputation.

A possible trust policy for an owner of a *Download* resource is as follows:

```
DownloadOwner refines Principal
  method exportEvents(): EventHistory = this.view()
  method isTrusted($r$:Reputation): Bool =
    size($r$.compFulfilled) $\geq 3$ $\wedge$ size($r$.violations) $= 0$ $\wedge$ size($r$.misuses) $= 0$
```

According to *DownloadLicense*, a principal is permitted unlimited uses of the dl resource if he is trusted by the resource owner. Note, however, that he cannot improve his reputation by doing so, *i.e.*, reputation cannot be "amplified" by using it repeatedly. Only if the licensee complies with at least 3 different licenses the "hard" way, with 1 upload for every 2 downloads and at least 5 uploads per license, will he be trusted by *another* principal, who may then permit him to access some other resource.

Anonymization. An anonymized reputation can serve as the proof that the licensee has fulfilled multiple licenses (in particular, that he has uploaded multiple files), but it cannot be linked to specific uses of an *Upload* resource, specific license identifiers, or specific files. This can be achieved by routing all upload and download requests via a chain of anonymizing routers, each of which is similar to Chaum's MIX [7]. At each link of the chain, the user is known under a different pseudonym. As reputation is passed down the chain, each router re-maps the user's pseudonym and license identifiers contained in the reputation, and purges event histories on which the reputation is based. This prevents eavesdroppers from relating requests that arrive to and leave from each router. Even if some routers in the chain are corrupt, reputations will be anonymized as long as at least one router is honest.

We assume that each router has two secret internal tables:

```
nameTranslate(UniqueName): UniqueName
licenseTranslate(LicenseId): LicenseId
```

The *nameTranslate* table is set up when the router chain is initialized. It tells the router the correspondence between the user's pseudonyms on the incoming and outgoing link. If the user is known on the outgoing link as p, then *nameTranslate*(p) returns his pseudonym on the incoming link.

We define *purgeEvidence* function as follows:

```
purgeEvidence($e$:EvidenceSet): EvidenceSet =
  { newEvidence(license ← licenseTranslate($ev$.license), justification ← $\{\}$) | $ev \in e$ }
```

Our policy for anonymizing routers changes the semantics of the *exportGlobalRep* method in order to anonymize reputations.

```
AnonymizingRouter refines Principal
  method nameTranslate(UniqueName): UniqueName
  method licenseTranslate(LicenseId): LicenseId
  method exportGlobalRep($a$:UniqueName, $excl$:Set UniqueName): Reputation =
    let $r$ = mergeReputations (this.localRep(this.nameTranslate($a$)),
                                 $p$.exportGlobalRep($a$, $excl \cup$ this.name))
          $\forall p$ such that $p$.name $\in$ this.trusted $\wedge$ $p$.name $\notin excl$
    in
    new Reputation(
          name ← $a$,
          compFulfilled ← purgeEvidence($r$.compFulfilled),
          partFulfilled ← purgeEvidence($r$.partFulfilled),
          violations ← purgeEvidence($r$.violations),
          misuses ← purgeEvidence($r$.misuses) )
```

In the new policy for *exportGlobalRep*, the router constructs the user's reputation, but then issues it under a different pseudonym, with license identifiers re-mapped, and event histories purged.

7 Example: Untrusted Allies

Instead of the semi-formal object-oriented notation, in this example we use MAUDE directly to model an online role-playing game (inspired by Clan Lord [14]), in which characters belong to clans that are competing in a search for valuable items. One clan can impede another by setting traps. Maps of regions that must be traversed help make the search safer and faster. A player may also be a free agent. A clan leader wants to avoid traps, and for this purpose might trade information in order to discover where traps have been placed and then send scout groups to disable them. When a trap is found and successfully disabled, the scout group leader reports this to the clan leader. A free agent wants map information to aid his own search or to trade. The agent may discover traps or learn about them by hanging out with other clans.

In the following we give MAUDE versions of resource and license definitions that a clan leader and an free agent might use to build trust in order to interact for mutual benefit, and sketch an interaction scenario illustrating how they might be used. The resource is a clan map owned by the clan leader. We abstract access to map data to a simple *use* action. The clan leader issues single-use licenses for the clan's map in exchange for confirmed good information about traps. Accepting the license *obliges* the agent to provide information whether or not he accesses the map. Confirmation is in the form of the scout group leader saying that good information was received from the licensee, that is, the trap was found and disabled. The clan leader trusts the scout group leader to report receipt of good information, and for simplicity we omit consideration of bad information.

We now present the main components of the MAUDE specification. We assume the following variable declarations:

```
vars u u0 u1 u2 sc cl fa : UniqueName .
var  lic : License .
vars  li li1 : LicenseId .
var h : EventHistory .
var ev : Event .
var act : Action .
```

In general, instances of object types are defined in MAUDE by declaring a constructor and giving equations defining the field selection methods. In order to allow for multiple clans, we define a `clanMap` constructor that takes the name of the owner (clan leader) as an argument, and give equations defining the field selectors.

```
op clanMap : UniqueName -> Resource [ctor] .
eq clanMap(cl) . owner = cl .
eq clanMap(cl) . status = Licensed .
```

Constructors for the basic *use*, *offer*, *accept* actions are defined as follows

```
op use : UniqueName Resource LicenseId -> Action .
eq use(u, r, li) . type = ResourceUse .
eq use(u, r, li) . actors = u .
eq use(u, r, li) . subject = r .
eq use(u, r, li) . license = li .

ops offer accept : License -> Action .
eq offer(lic) . type = LicenseOffer .
```

```
  eq accept(lic) . type = LicenseAccept .
  eq offer(lic) . actors  = (lic . issuer) (lic . licensee) .
  eq offer(lic) . subject = lic . resource .
  eq offer(lic) . license = lic . id .
  ...
```

Here `(lic . issuer) (lic . licensee)` denotes a set consisting of the unique names `(lic . issuer)` and `(lic . licensee)` (which could be the same). In addition we introduce a new action type, `Tell`, and two `Tell` actions.

```
  op Tell : -> ActionType .

  op giveInfo : UniqueName UniqueName UniqueName LicenseId
                -> Action .
  eq giveInfo(sc, fa, cl, li) . type = Tell .
  eq giveInfo(sc, fa, cl, li) . actors = sc fa .
  eq giveInfo(sc, fa, cl, li) . subject = clanMap(cl) .
  eq giveInfo(sc, fa, cl, li) . license = li .

  op goodInfo : UniqueName  UniqueName LicenseId -> Action .
  eq goodInfo(cl, sc, li) . type = Tell .
  eq goodInfo(cl, sc, li) . actors = cl sc .
  eq goodInfo(cl, sc, li) . subject = clanMap(cl) .
  eq goodInfo(cl, sc, li) . license = li .
```

The term `giveInfo(sc,fa,cl,li)` abstracts the action in which an agent `fa` gives information to scout `sc` as required by the license with identifier `li` issued by `cl`. The term `goodInfo(cl,sc,li)` names the action in which the scout `sc` reports receipt of good information to the clan leader `cl`. The term `mapLic(li,cl,fa,sc)` denotes a clan map license with fields defined as follows.

```
  op mapAccess : -> LicenseKind .
  op mapLic : LicenseId UniqueName UniqueName UniqueName
             -> License [ctor] .
  eq mapLic(li,cl,fa,sc) . id = li .
  eq mapLic(li,cl,fa,sc) . kind = mapAccess .
  eq mapLic(li,cl,fa,sc) . resource  = clanMap(cl) .
  eq mapLic(li,cl,fa,sc) . issuer = cl .
  eq mapLic(li,cl,fa,sc) . licensee  = fa .
```

The argument `sc` specifies the scout to whom information is to be given. To simplify definition of some license methods, a partial function `validate(h,li)` is defined. If the license with identifier `id` has been offered and accepted in history `h`, then `validate(h,li)` is a triple `oas(e-offer,e-accept,l-events)` of sort `OfferAcceptSplit` where `e-offer` is the offer event, `e-accept` is the accept event, `l-events` is the set of events after the accept event that are associated with the license. Otherwise `validate(h,li)` is an element of the "error" supersort `[OfferAcceptSplit]`. Map licenses are single-use only. A clan map license is *done* in an event history if it is validated—`oas :: OfferAcceptSplit` tests membership in sort OfferAcceptSplit—and the associated event set contains an event whose action is a use of the license by the licensee fa.

```
var oas : [OfferAcceptSplit] .
ceq mapLic(li,cl,fa,sc) . done(h) =
     if (oas :: OfferAcceptSplit)
     then isUsed(events(oas),li, fa)
     else false  fi
 if oas := validate(h,li)  .
```

It is ok for an agent fa to use a clan map owned by leader cl with a map license issued by that leader in the context of history h if the license is valid in the history and the scout sc has verified that the agent has provided good information.

```
eq clanMap(cl) . useOk(h,mapLic(li,cl,fa,sc)) =
   ( mapLic(li,cl,fa,sc) . valid(h)
    and
    not(mapLic(li,cl,fa,sc) . done(h))
    and
    hasGoodInfoFor(h,li,cl,sc) ) .
eq clanMap(u) . useOk(h,lic) = false [owise] .
```

The [owise] attribute of the second equation says that for any map and license not matching the previous equation, useOk is false.

The *permits* method is defined as follows:

```
eq mapLic(li,cl,fa,sc) . permits(h, ev)
  = mapLic(li,cl,fa,sc) . permits(before(ev, h), ev.action) .

eq mapLic(li,cl,fa,sc) .  permits(h, use(fa, clanMap(cl),li))
    =  clanMap(cl) . useOk(h,mapLic(li,cl,fa,sc)) .
eq mapLic(li,cl,fa,sc) . permits(h, act) = false [owise] .
```

Permission for a *use* action reduces to the *useOk* test, and no other action is permitted by a map license.

A map license is violated in a history if it can be validated and there is a use event with no report of good information.

```
ceq mapLic(li,cl,fa,sc) . violated(h) =
     if (oas :: OfferAcceptSplit)
     then (isUsed(events(oas),li, fa)
           and
           not(hasGoodInfoFor(events(oas), li,cl,sc)))
     else false  fi
  if oas := validate(h,li)  .
```

To see how these policies work in practice, we defined *active* principals — actors that communicate via message passing or joint actions — and gave rules for the behavior of a clan leader, a scout, and a free agent. A configuration is a set of actors and messages together with a clock object used to generate time stamps. Active principals and other objects have the form

```
    [ name : ClassId | attributes ]
```

where `attributes` (of sort `Atts`) consists of the set of field values and other internal state information.

The following is an initial configuration with a clock, a scout (*fred*), a clan leader (*joe*, ClassId is `CL`), a free agent (*sam*), and a message to start things off. For convenience, to define specific configurations, the operator `u` is defined to map strings to unique names.

```
  op ic : -> Conf .
  eq ic =
    [u("clock") : Clock | time(0)]
    [u("fred") : ScoutC |
       licenses(mt),trusted(mt),view(mt),pend(nil)]
    [u("joe") : CL |
       licenses(mt),trusted(mt),view(mt),pend(nil),lctr(0)]
    [u("sam") : freeAgentC |
       licenses(mt),trusted(mt),view(mt),pend(nil)]
    msg(u("joe"), u("sam"), mapReq(u("fred")))
```

The clock has a `time` attribute. The principals have the required `licenses`, `trusted`, and `view` attributes, and in addition a `pend` attribute whose value is a list of pending actions, initially `nil` (empty). The clan leader also has an attribute `lctr` used to generate fresh license identifiers. There are rules for each of the actions that a principal can participate in. When such a rule is applied, an event is created with a new timestamp and each participant adds the event to its `view`. As an example, here is the rule for an `offer` action.

```
 vars al0 al1 : ActionList .
 vars aatts catts : Atts .
 rl[offer]:
  [ clk : Clock | time(m) ]
  [ cl : CL | catts, view(v0),
       pend(offer(mapLic(l(cl,n),cl,fa,sc)) al0)]
  [ fa : freeAgentC | aatts, view(v1), pend(al1) ]
  =>
  [ clk : Clock | time(s m) ]
  [ cl : CL | catts,  pend(al0),
       view(v0 event(t(m),offer(mapLic(l(cl,n),cl,fa,sc)))) ]
  [ fa : freeAgentC | aatts,
       pend(al1 accept(mapLic(l(cl,n),cl,fa,sc))),
       view(v1 event(t(m),offer(mapLic(l(cl,n),cl,fa,sc)))) ]
  .
```

Timestamps are isomorphic to natural numbers, with `t(m)` being the timestamp corresponding to the number `m`. The additional rule action rules are for

- `accept(mapLic(l(cl,n),cl,fa,sc))` for clan leader and free agent
- `giveInfo(sc,fa,cl,l(cl,n))` for free agent and scout

- `goodInfo(cl,sc,l(cl,n))` for scout and clan leader
- `use(fa,clanMap(cl),l(cl,n))` for free agent and clan leader

In addition, there are application-independent rules for principals to exchange event and reputation information, which implement the *exportEvents* and *exportGlobalRep* methods.

Delivering the message to the clan leader *joe* results in an offer action being put in the pending action list.

We can use MAUDE's rewrite engine to see one way in which the configuration might evolve. In the resulting configuration the pending action lists are all empty and five events have happened.

```
  event(t(0),
    offer(mapLic(l(u("joe"),0),u("joe"),u("sam"),u("fred"))))
  event(t(1),
    accept(mapLic(l(u("joe"),0),u("joe"),u("sam"),u("fred"))))
  event(t(2),giveInfo(u("fred"),u("sam"),u("joe"),l(u("joe"),0)))
  event(t(3),goodInfo(u("joe"),u("fred"),l(u("joe"),0))),
  event(t(4),use(u("sam"),clanMap(u("joe")),l(u("joe"),0)))
```

We can ask if it is possible for the map license to be misused starting with the above initial configuration using the MAUDE search command as follows:

```
search ic =>! C:Conf
  [u("joe") : CL | atts:Atts, licenses(lic:License),
    view(v e:Event) ]
 such that  (e:Event . action . type == ResourceUse and
             not(lic:License . permits(v e:Event, e:Event))) .
```

The answer is yes, if the `use` event occurs before the `goodInfo` event.

For simplicity, the above scenario focuses on a setting with one clan leader, one scout group, and one free agent. Scenarios involving multiple clan leaders, free agents, and/or scouts can easily be analyzed by starting with larger configurations. The clan map license model can be used as a starting point for modeling how the clan leader and the free agent might build mutual [dis]trust and use this reputation-based trust to develop simple strategies for deciding when to trade information. For example, since the clan leader trusts the scout group leader to reliably report when good information has been given and the map use license has thus been complied with, such reports could be used to add agents to the leader's *trusted* set.

8 Related work

The framework for reputation-based trust developed in this paper is most closely related to trust systems for peer-to-peer and ubiquitous computing. The existing techniques mainly focus on differentiating and quantifying levels of trust assigned to different agents in the system, whereas our objective is to give precise formal semantics to the notion of reputation, while leaving trust decisions to individual agents. In this sense, our approach is complementary to those explored in the literature.

Abdul-Rahman and Hailes [1] model reputation as a tuple of “very good,” “good,” “bad,” and “very bad” experiences, which is similar to our model for reputation described in section 4. Precise semantics of “good” and “bad” is left unspecified in [1]. By contrast, we interpret “good” and “bad” as compliance with and misuse of licenses. The focus of [1] is on computing weighted trust values based on the relative trustworthiness of information sources.

Shankar and Arbaugh [17] also focus on assigning different values of trust to agents depending on their identity and physical context. Their approach is thus complementary to ours. We plan to investigate extensions of our framework with multiple and possibly dynamically evolving levels of trust.

Azzedin and Maheswaran [3, 4] interpret reputation as expectation of behavior based on collective information. Their model takes into account trustworthiness of information sources. They view information about reputation supplied by individual agents as numerical values, whereas we focus on low-level interpretation of reputation as fulfillment and misuse of licenses.

Damiani *et al.* [11] propose an overlay protocol for peer-to-peer networks, in which reliability of a resource can be established by distributed polling. Again, the semantics of reputation is not as refined as in our framework.

9 Conclusions

We have presented a formal model for reputation-based trust management that allows mutually distrusting agents to develop a basis for interaction even in the absence of a central credential authority. The model can be applied in the context of peer-to-peer applications, online games, or military simulation, among others.

We have started with a very simple model and there are several elaborations that can be considered, such as treating temporal aspects in more detail, mechanisms for allowing reputation (good or bad) to degrade over time, and taking trustworthiness of the source into account when evaluating evidence and reputation.

We plan to develop a set of standard high-level policies for creating new trust judgments on the basis of reputation. Another direction of future work is to introduce economic notions such as cost-benefit ratios and their relation to reputation and trust.

Acknowledgements. The authors thank the anonymous reviewers for helpful comments and Tim McCarthy for suggesting the game application.

References

[1] A. Abdul-Rahman and S. Hailes. Supporting trust in virtual communities. In *Proc. 33rd IEEE Hawaii International Conference on System Sciences (HICSS) - Volume 6*. IEEE Computer Society, 2000.

[2] E. Adar and B. Huberman. Free riding on Gnutella. *First Monday*, 5(10), 2000.

[3] F. Azzedin and M. Maheswaran. Integrating trust into grid resource management systems. In *Proc. 31st International Conference on Parallel Processing*, pages 47–54. IEEE Computer Society, 2002.

[4] F. Azzedin and M. Maheswaran. Trust modeling for peer-to-peer based computing systems. In *Proc. 12th IEEE Heterogeneous Computing Workshop*, 2003.

[5] H. Baker and C. Hewitt. Laws for communicating parallel processes. In *Proc. IFIP Congress*, pages 987–992. IFIP, 1977.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society, 1996.

[7] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[8] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *LNCS*, pages 46–66. Springer-Verlag, 2000.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. The Maude 2.0 system. In *Proc. 14th International Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *LNCS*, pages 76–87. Springer-Verlag, 2003.

[10] W. D. Clinger. *Foundations of Actor Semantics*. PhD thesis, MIT, 1981. MIT Artificial Intelligence Laboratory AI-TR-633.

[11] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 207–216. ACM, 2002.

[12] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Proc. Financial Cryptography*, volume 2357 of *LNCS*, pages 120–135. Springer-Verlag, 2002.

[13] C. Gunter, S. Weeks, and A. Wright. Models and languages for digital rights. In *Proc. 34th IEEE Hawaii International Conference on System Sciences (HICSS) - Volume 9*. IEEE Computer Society, 2001.

[14] Clan Lord. `http://www.clanlord.com/`.

[15] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[16] R. Pucella and V. Weissman. A logic for reasoning about digital rights. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 282–294. IEEE Computer Society, 2002.

[17] N. Shankar and W. Arbaugh. On trust for ubiquitous computing. In *Proc. Workshop on Security for Ubiquitous Computing*, 2002.

[18] P. Syverson, D. Goldschlag, and M. Reed. Anonymous connections and onion routing. In *Proc. IEEE Symposium on Security and Privacy*, pages 44–54. IEEE Computer Society, 1997.