A Framework for Concrete Reputation-Systems with Applications to History-Based Access Control*

Karl Krukow† ‡
BRICS
University of Aarhus, Denmark
krukow@brics.dk

Mogens Nielsen
BRICS
University of Aarhus, Denmark
mn@brics.dk

Vladimiro Sassone§
Department of Informatics
University of Sussex, UK
vs@sussex.ac.uk

ABSTRACT

In a reputation-based trust-management system, agents maintain information about the past behaviour of other agents. This information is used to guide future trust-based decisions about interaction. However, while trust management is a component in security decision-making, many existing reputation-based trust-management systems provide no formal security-guarantees. In this extended abstract, we describe a mathematical framework for a class of simple reputation-based systems. In these systems, decisions about interaction are taken based on policies that are exact requirements on agents' past histories. We present a basic declarative language, based on pure-past linear temporal logic, intended for writing simple policies. While the basic language is reasonably expressive (encoding e.g. Chinese Wall policies) we show how one can extend it with quantification and parameterized events. This allows us to encode other policies known from the literature, e.g., 'one-out-of-k'. The problem of checking a history with respect to a policy is efficient for the basic language, and tractable for the quantified language when policies do not have too many variables.

*Extended Abstract. The full paper is available as a BRICS technical report, RS-05-23 [17], online at `http://www.brics.dk/RS/05/23`.

†Nielsen and Krukow are supported by SECURE: Secure Environments for Collaboration among Ubiquitous Roaming Entities, EU FET-GC IST-2001-32486. Krukow is supported by DISCO: Semantic Foundations of Distributed Computing, EU IHP, Marie Curie, HPMT-CT-2001-00290.

‡BRICS: Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

§Supported by MyThS: Models and Types for Security in Mobile Distributed Systems, EU FET-GC IST-2001-32617.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'05, November 7–11, 2005, Alexandria, Virginia, USA.
Copyright 2005 ACM 1-59593-226-7/05/0011 ...$5.00.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information**]: Security and Protection; D.4.6 [**Operating Systems**]: Security and Protection – Access controls

General Terms

Algorithms, Security, Languages

Keywords

Reputation, trust management, history-based access control, temporal logic, model checking

1. INTRODUCTION

In global-scale distributed systems, traditional authorization mechanisms easily become either overly restrictive, or very complex [2]. In part, this is due to the vast numbers of principals they must encompass, and the open nature of the systems. In *dynamic* and *reputation-based* trust-management systems, the problems of scale and openness are countered by taking a less static approach to authorization and, more generally, decision making. In these systems, principals keep track of the history of interactions with other principals. The recorded behavioural information is used to guide future decisions about interaction (see references [15, 24, 27] on reputation). This dynamic approach is being investigated as a means of overcoming the above mentioned *security* problems of global-scale systems. Yet, in contrast with traditional (cryptographic) security research, within the area of dynamic trust and reputation, no widely accepted security-models exist, and to our knowledge, few systems provide provable security guarantees (see, however, references [6, 18, 20] on general formal modelling of trust in global computing systems).

Many reputation systems have been proposed in the literature, but in most of these the recorded behavioural information is heavily abstracted. For example, in the EigenTrust system [16], behavioural information is obtained by counting the number of 'satisfactory' and 'unsatisfactory' interactions with a principal. Besides lacking a precise semantics, this information has abstracted away any notion of time, and is further reduced (by normalization) to a number in the interval $[0, 1]$. In the Beta reputation system [14], similar abstractions are performed, obtaining a numerical value in $[-1, 1]$ (with a statistical interpretation). There are many other examples of such information abstraction or aggregation in the reputation-system literature [15], and the only

non-example we are aware of is the framework of Shmatikov and Talcott [27] which we discuss further in the concluding section.

Abstract representations of behavioural information have their advantages (e.g., numerical values are often easily comparable, and require little space to store), but clearly, information is lost in the abstraction process. For example, in EigenTrust, value 0 may represent both "no previous interaction" and "many unsatisfactory previous interactions" [16]. Consequently, one cannot verify exact properties of past behaviour given only the reputation information.

In this paper, the concept of 'reputation system' is to be understood very broadly, simply meaning *any system in which principals record and use information about past behaviour of principals, when assessing the risk of future interaction.* We present a formal framework for a class of simple reputation systems in which, as opposed to most "traditional" systems, behavioural information is represented in a very concrete form. The advantage of our concrete representation is that sufficient information is present to check precise properties of past behaviour. In our framework, such requirements on past behaviour are specified in a declarative policy-language, and the basis for making decisions regarding future interaction becomes the verification of a behavioural history with respect to a policy. This enables us to define reputation systems that provide a form of provable "security" guarantees, intuitively, of the form: "If principal p gains access to resource r at time t, then the past behaviour of p up *until* time t satisfies requirement ψ_r."

To get the flavour of such requirements, we preview an example policy from a declarative language formalized in the following sections. Edjlali *et al.* [9] consider a notion of history-based access control in which unknown programs, in the form of mobile code, are dynamically classified into equivalence classes of programs according to their behaviour (e.g. "browser-like" or "shell-like"). This dynamic classification falls within the scope of our very broad understanding of reputation systems. The following is an example of a policy written in our language, which specifies a property similar to that of Edjlali *et al.*, used to classify "browser-like" applications:

$$\begin{aligned}\psi^{browser} \quad \equiv \quad & \neg \mathsf{F}^{-1}(\texttt{modify}) \quad \wedge \\ & \neg \mathsf{F}^{-1}(\texttt{create-subprocess}) \quad \wedge \\ & \mathsf{G}^{-1}\left(\forall x.\left[\texttt{open}(x) \rightarrow \mathsf{F}^{-1}(\texttt{create}(x))\right]\right)\end{aligned}$$

Informally, the atoms `modify`, `create-subprocess`, $\texttt{open}(x)$ and $\texttt{create}(x)$ are *events* which are observable by monitoring an entity's behaviour. The latter two are *parameterized* events, and the quantification "$\forall x$" ranges over the possible parameters of these. Operator F^{-1} means 'at some point in the past,' G^{-1} means 'always in the past,' and constructs $\wedge$ and $\neg$ are conjunction and negation, respectively. Thus, clauses $\neg\mathsf{F}^{-1}(\texttt{modify})$ and $\neg\mathsf{F}^{-1}(\texttt{create-subprocess})$ require that the application has never modified a file, and has never created a sub-process. The final, quantified clause $\mathsf{G}^{-1}\left(\forall x.\left[\texttt{open}(x) \rightarrow \mathsf{F}^{-1}(\texttt{create}(x))\right]\right)$ requires that whenever the application opens a file, it must previously have created that file. For example, if the application has opened the local system-file "`/etc/passwd`" (i.e. a file which it has not created) then it cannot access the network (a right assigned to the "browser-like" class). If, instead, the application has previously only read files it has created, then it will be allowed network access.

1.1 Contributions and Outline

We present a formal model of the behavioural information that principals obtain in our class of reputation systems. This model is based on previous work using event structures for modelling observations [21], but our treatment of behavioural information departs from the previous work in that we perform (almost) no information abstraction. The event-structure model is presented in Section 2.

We describe our formal declarative language for interaction policies. In the framework of event structures, behavioural information is modelled as sequences of sets of events. Such linear structures can be thought of as (finite) models of linear temporal logic (LTL) [22]. Indeed, our basic policy language is based on a (pure-past) variant of LTL. We give the formal syntax and semantics of our language, and provide several examples illustrating its naturality and expressiveness. We are able to encode several existing approaches to history-based access control, e.g. the Chinese Wall security policy [3] and a restricted version of so-called 'one-out-of-k' access control [9]. The formal description of our language, as well as examples and encodings, is presented in Section 3.

An interesting new problem is how to re-evaluate policies efficiently when interaction histories change as new information becomes available. It turns out that this problem, which can be described as dynamic model-checking, can be solved very efficiently using an algorithm adapted from that of Havelund and Roşu, based on the technique of dynamic programming, used for runtime verification [13]. Interestingly, although one is verifying properties of an *entire* interaction history, one needs not store this complete history in order to verify a policy: old interaction can be efficiently summarized relative to the policy. Descriptions of two algorithms, and analysis of their time- and space-requirements is given in the full paper [17]. The results are outlined in Section 3.

Our simple policy language can be extended to encompass policies that are more realistic and practical (e.g., for history-based access control [1, 9, 11, 29], and within the traditional domain of reputation systems: peer-to-peer- and online feedback systems [16, 24]). In the full paper we describe a form of *quantitative* policies, a notion of *policy referencing* to include other principals' data, and *quantified policies*. In Section 5 we illustrate the extension to quantified policies, and describe results regarding policy-checking algorithms and complexity.

Related work is discussed in the concluding section. Due to space restrictions no proofs are included in this paper. The interested reader is referred to the associated technical report [17] for proofs and additional examples.

2. OBSERVATIONS AS EVENTS

Agents in a distributed system obtain information by observing events which are typically generated by the reception or sending of messages. The structure of these message exchanges are given in the form of protocols known to both parties before interaction begins. By *behavioural observations*, we mean observations that the parties can make about specific runs of such protocols. These include information about the contents of messages, diversion from protocols, failure to receive a message within a certain time-frame, etc.

Our goal in this section, is to give precise meaning to the notion of behavioural observations. Note that, in the setting of large-scale distributed environments, often, a particular agent will (concurrently) be involved in several instances of protocols; each instance generating events that are logically connected. One way to model the observation of events is using a process algebra with "state", recording input/output reactions, as is done in the calculus for trust management, ctm [7]. Here we are not interested in modelling interaction protocols in such detail, but merely assume some system responsible for generating events.

We will use the event-structure framework of Nielsen and Krukow [21] as our model of behavioural information. The framework is suitable for our purpose as it provides a *generic* model for observations that is independent of any specific programming language. In the framework, the information that an agent has about the behaviour of another agent p, is information about a number of (possibly active) protocol-runs with p, represented as a sequence of *sets of events*, $x_1 x_2 \cdots x_n$, where event-set x_i represents information about the ith initiated protocol-instance. Note, in frameworks for history-based access control (e.g., [1, 9, 11]), histories are always sequences of *single* events. Our approach generalizes this to allow sequences of (finite) *sets* of events; a generalization useful for modelling information about protocol runs in distributed systems.

We present the event-structure framework as an abstract interface providing two operations, **new** and **update**, which respectively records the initiation of a new protocol run, and updates the information recorded about an older run (i.e. updates an event-set x_i). A specific implementation then uses this interface to notify our framework about events.

2.1 The Event-Structure Framework

In order to illustrate the event-structure framework [21], we use an example complementing its formal definitions. We will use a scenario inspired by the eBay online auction-house [8], but deliberately over-simplified to illustrate the framework.

On the eBay website, a seller starts an auction by announcing, via the website, the item to be auctioned. Once the auction has started the highest bid is always visible, and bidders can place bids. A typical auction runs for 7 days, after which the bidder with the highest bid wins the auction. Once the auction has ended, the typical protocol is the following. The buyer (winning bidder) sends payment of the amount of the winning bid. When payment has been received, the seller confirms the reception of payment, and ships the auctioned item. Optionally, both buyer and seller may leave feedback on the eBay site, expressing their opinion about the transaction. Feedback consist of a choice between ratings 'positive', 'neutral' and 'negative', and, optionally, a comment.

We will model behavioural information in the eBay scenario from the buyers point of view. We focus on the interaction following a winning bid, i.e. the protocol described above. After winning the auction, buyer (B) has the option to send payment, or ignore the auction (possibly risking to upset the seller). If B chooses to send payment, he may observe confirmation of payment, and later the reception of the auctioned item. However, it may also be the case that B doesn't observe the confirmation within a certain time-frame (the likely scenario being that the seller is a fraud). At any time during this process, each party may choose to leave feedback about the other, expressing their degree of satisfaction with the transaction. In the following, we will model an abstraction of this scenario where we focus on the following events: buyer pays for auction, buyer ignores auction, buyer receives confirmation, buyer receives no confirmation within a fixed time-limit, and *seller* leaves positive, neutral or negative feedback (note that we do not model the *buyer* leaving feedback).

The basis of the event-structure framework is the fact that the observations about protocol runs, such as an eBay transaction, have structure. Observations may be in *conflict* in the sense that one observation may exclude the occurrence of others, e.g. if the seller leaves positive feedback about the transaction, he can not leave negative or neutral feedback. An observation may *depend* on another in the sense that the first may only occur if the second has already occurred, e.g. the buyer cannot receive a confirmation of received payment if he has not made a payment. Finally, if two observations are neither in conflict nor dependent, they are said to be *independent*, and both may occur (in any order), e.g. feedback-events and receiving confirmation are independent. Note that 'independent' just means that the events are not in conflict nor dependent (e.g., it does *not* mean that the events are independent in any statistical sense). These relations between observations are directly reflected in the definition of an event structure. (For a general account of event structures, traditionally used in semantics of concurrent languages, consult the handbook chapter of Winskel and Nielsen [30]).

Definition 2.1 (Event Structure). An *event structure* is a triple $ES = (E, \leq, \#)$ consisting of a set E, and two binary relations on E: $\leq$ and $\#$. The elements $e \in E$ are called *events*, and the relation $\#$, called the *conflict relation*, is symmetric and irreflexive. The relation $\leq$ is called the *(causal) dependency relation*, and partially orders E. The dependency relation satisfies the following axiom, for any $e \in E$:

$$\text{the set } \lceil e \rceil \stackrel{(\text{def})}{=} \{e' \in E \mid e' \leq e\} \text{ is finite.}$$

The conflict- and dependency-relations satisfy the following "transitivity" axiom for any $e, e', e'' \in E$

$$\big(e \# e' \text{ and } e' \leq e''\big) \text{ implies } e \# e''$$

Two events are *independent* if they are not in either of the two relations.

We use event structures to model the possible observations of a single agent in a protocol, e.g. the event structure in Figure 1 models the events observable by the buyer in our eBay scenario.

The two relations on event structures imply that not all subsets of events can be observed in a protocol run. The following definition formalizes exactly what sets of observations are observable.

Definition 2.2 (Configuration). Let $ES = (E, \leq, \#)$ be an event structure. We say that a subset of events $x \subseteq E$ is a *configuration* if it is *conflict free* (C.F.), and *causally closed* (C.C.). That is, it satisfies the following two properties, for any $d, d' \in x$ and $e \in E$

$$\text{(C.F.) } d \not\# d'; \text{ and (C.C.) } e \leq d \Rightarrow e \in x$$

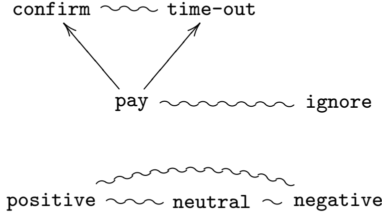

Figure 1: An event structure modelling the buyer's observations in the eBay scenario. (Immediate) Conflict is represented by $\sim$, and dependency by $\to$.

Notation 2.1. $\mathcal{C}_{ES}$ denotes the set of configurations of ES, and $\mathcal{C}^0_{ES} \subseteq \mathcal{C}_{ES}$ the set of *finite* configurations. A configuration is said to be *maximal* if it is maximal in the partial order $(\mathcal{C}_{ES}, \subseteq)$. Also, if $e \in E$ and $x \in \mathcal{C}_{ES}$, we write $e \# x$, meaning that $\exists e' \in x.e \# e'$. Finally, for $x, x' \in \mathcal{C}_{ES}, e \in E$, define a relation $\to$ by $x \xrightarrow{e} x'$ iff $e \notin x$ and $x' = x \cup \{e\}$. If $y \subseteq E$ and $x \in \mathcal{C}_{ES}, e \in E$ we write $x \overset{e}{\not\to} y$ to mean that either $y \notin \mathcal{C}_{ES}$ or it is not the case that $x \xrightarrow{e} y$.

A finite configuration models information regarding *a single* interaction, i.e. a single run of a protocol. A maximal configuration represents complete information about a single interaction. In our eBay example, sets $\emptyset$, $\{\texttt{pay}, \texttt{positive}\}$ and $\{\texttt{pay}, \texttt{confirm}, \texttt{positive}\}$ are examples of configurations (the last configuration being maximal), whereas

$$\{\texttt{pay}, \texttt{confirm}, \texttt{positive}, \texttt{negative}\}$$

and $\{\texttt{confirm}\}$ are non-examples.

In general, the information that one agent possesses about another will consist of information about *several* protocol runs; the information about each individual run being represented by a configuration in the corresponding event structure. The concept of a local interaction history models this.

Definition 2.3 (Local Interaction History). Let ES be an event structure, and define a *local interaction history* in ES to be a sequence of finite configurations, $h = x_1 x_2 \cdots x_n \in {\mathcal{C}^0_{ES}}^*$. The individual components x_i in the history h will be called *sessions*.

In our eBay example, a local interaction history could be the following:

$$\{\texttt{pay}, \texttt{confirm}, \texttt{pos}\}\{\texttt{pay}, \texttt{confirm}, \texttt{neu}\}\{\texttt{pay}\}$$

Here `pos` and `neu` are abbreviations for the events `positive` and `neutral`. The example history represents that the buyer has won three auctions with the particular seller, e.g. in the third session the buyer has (so-far) observed only event `pay`.

We assume that the actual system responsible for notification of events will use the following interface to the model.

Definition 2.4 (Interface). Define an operation **new** : ${\mathcal{C}^0_{ES}}^* \to {\mathcal{C}^0_{ES}}^*$ by $\textbf{new}(h) = h\emptyset$. Define also a partial operation **update** : ${\mathcal{C}^0_{ES}}^* \times E \times \mathbb{N} \to {\mathcal{C}^0_{ES}}^*$ as follows. For any $h = x_1 x_2 \cdots x_i \cdots x_n \in {\mathcal{C}^0_{ES}}^*$, $e \in E$, $i \in \mathbb{N}$, $\textbf{update}(h, e, i)$ is undefined if $i \notin \{1, 2, \ldots, n\}$ or $x_i \overset{e}{\not\to} x_i \cup \{e\}$. Otherwise

$$\textbf{update}(h, e, i) = x_1 x_2 \cdots (x_i \cup \{e\}) \cdots x_n$$

Remarks. Note, that while the order of sessions is recorded (a local history is a *sequence*), in contrast, the order of *independent* events within *a single session* is not. For example, in our eBay scenario we have

$$\textbf{update}(\textbf{update}(\{\texttt{pay}\}, \texttt{neutral}, 1), \texttt{confirm}, 1) =$$
$$\textbf{update}(\textbf{update}(\{\texttt{pay}\}, \texttt{confirm}, 1), \texttt{neutral}, 1)$$

Hence independence of events is a choice of abstraction one may make when designing an event-structure model (because one is not interested in the particular order of events, or because the exact recording of the order of events is not feasible). However, note that this is not a limitation of event structures: in a scenario where this order of events is relevant (and observable), one can always use a "serialized" event structure in which this order of occurrences is recorded. A serialization of events consists of splitting the events in question into different events depending on the order of occurrence, e.g., supposing in the example one wants to record the order of `pay` and `pos`, one replaces these events with events `pay-before-pos`,`pos-before-pay`, `pay-after-pos` and `pos-after-pay` with the obvious causal- and conflict-relations.

When applying our logic (described in the next section) to express policies for history-based access control (HBAC), we often use a special type of event structure in which the conflict relation is the maximal irreflexive relation on a set E of events. The reason is that *histories* in many frameworks for HBAC, are sequences of single events for a set E. When the conflict relation is maximal on E, the configurations of the corresponding event structure are exactly singleton event-sets, hence we obtain a useful specialization of our model, compatible with the tradition of HBAC.

3. A LANGUAGE FOR POLICIES

The reason for recording behavioural information is that it can be used to guide future decisions about interaction. We are interested in binary decisions, e.g., access-control and deciding whether to interact or not. In our proposed system, such decisions will be made according to interaction policies that specify exact requirements on local interaction histories. For example, in the eBay scenario from last section, the bidder may adopt a policy stating: "only bid on auctions run by a seller which has never failed to send goods for won auctions in the past."

In this section, we propose a declarative language which is suitable for specifying interaction policies. In fact, we shall use a pure-past variant of linear-time temporal logic, a logic introduced by Pnueli for reasoning about parallel programs [22]. Pure-past temporal logic turns out to be a natural and expressive language for stating properties of past behaviour. Furthermore, linear-temporal-logic models are linear Kripke-structures, which resemble our local interaction histories. We define a satisfaction relation $\models$, between such histories and policies, where judgement $h \models \psi$ means that the history h satisfies the requirements of policy ψ.

3.1 Formal Description

3.1.1 Syntax.

The syntax of the logic is parametric in an event structure $ES = (E, \leq, \#)$. There are constant symbols $e, e', e_i, \ldots$ for each $e \in E$. The syntax of our language, which we denote $\mathcal{L}(ES)$, is given by the following BNF.

$$\psi \quad ::= \quad e \mid \Diamond e \mid \psi_0 \ op \ \psi_1 \mid \neg\psi \mid \mathsf{X}^{-1}\psi \mid \psi_0 \ \mathsf{S} \ \psi_1$$

Meta-variable *op* ranges over $\{\wedge, \vee\}$. The constructs e and $\Diamond e$ are both *atomic* propositions. In particular, $\Diamond e$ is *not* the application of the usual modal operator $\Diamond$ (with the "temporal" semantics) to formula e. Informally, the formula e is true in a session if the event e has been observed in that session, whereas $\Diamond e$, pronounced "e is possible", is true if event e *may still occur* as a future observation in that session. The operators X^{-1} ('last time') and S ('since') are the usual past-time operators.

3.1.2 Semantics.

A *structure* for $\mathcal{L}(ES)$, where $ES = (E, \leq, \#)$ is an event structure, is a non-empty local interaction history in ES, $h \in {\mathcal{C}^0_{ES}}^+$. We define the satisfaction relation $\models$ between structures and policies, i.e. $h \models \psi$ means that the history h satisfies the requirements of policy ψ. We will use a variation of the semantics in linear Kripke structures: satisfaction is defined from the *end* of the sequence "towards" the beginning, i.e. $h \models \psi$ iff $(h, |h|) \models \psi$. To define the semantics of $(h, i) \models \psi$, let $h = x_1 x_2 \cdots x_N \in {\mathcal{C}^0_{ES}}^*$ with $N > 0$, and $1 \leq i \leq N$. Define $(h, i) \models \psi$ by structural induction in ψ.

$$\begin{array}{lll}
(h,i) \models e & \text{iff} & e \in x_i \\
(h,i) \models \Diamond e & \text{iff} & e \not\# \, x_i \\
(h,i) \models \psi_0 \wedge \psi_1 & \text{iff} & (h,i) \models \psi_0 \text{ and } (h,i) \models \psi_1 \\
(h,i) \models \psi_0 \vee \psi_1 & \text{iff} & (h,i) \models \psi_0 \text{ or } (h,i) \models \psi_1 \\
(h,i) \models \neg\psi & \text{iff} & (h,i) \not\models \psi \\
(h,i) \models \mathsf{X}^{-1}\psi & \text{iff} & i > 1 \text{ and } (h, i-1) \models \psi \\
(h,i) \models \psi_0 \ \mathsf{S} \ \psi_1 & \text{iff} & \exists j \leq i. \big[(h,j) \models \psi_1 \text{ and} \\
 & & \quad \forall k. (j < k \leq i \Rightarrow (h,k) \models \psi_0) \big]
\end{array}$$

Remarks. There are two main reasons for restricting ourselves to the *pure-past* fragment of temporal logic (PPLTL). Most importantly, PPLTL is an expressive and *natural* language for stating requirements over *past* behaviour, e.g. history-based access control. Hence in our application one wants to speak about the past, not the future. We justify this claim further by providing (natural) encodings of several existing approaches for checking requirements of past behaviour (c.f. Example 3.2 and 3.3 in the next section). Secondly, although one could add future operators to obtain a seemingly more expressive language, a result of Laroussinie *et al.* quantifies exactly what is lost by this restriction [19]. Their result states that LTL can be *exponentially more succinct* than the pure-future fragment of LTL. It follows from the duality between the pure-future and pure-past operators, that when restricting to finite linear Kripke structures, and interpreting $h \models \psi$ as $(h, |h|) \models \psi$, then our pure-past fragment can express *any* LTL formula (up to initial equivalence), though possibly at the cost of an exponential increase in the size of the formula. Another advantage of PPLTL is that, while Sistla and Clarke proved that the model-checking problem for linear temporal logic with future- and past-operators (LTL) is PSPACE-complete [28], there are very efficient algorithms for (finite-path) model-checking pure-past fragments of LTL, and (as we shall see in Section 4) also for the dynamic policy-checking problem.

Note that we have defined the semantics of the logic only for *non-empty* structures, $h \in {\mathcal{C}^0_{ES}}^+$. This means that policies cannot be interpreted if there has been no previous interaction. In practice it is up to each agent to decide by other means if interaction should take place in the case of no past history. For the remainder of this paper we shall define $\epsilon \models \psi$ iff $\emptyset \models \psi$, that is we (arbitrarily) identify the empty sequence (ϵ) with the singleton sequence consisting of only the empty configuration. Finally, we define standard abbreviations: $\mathsf{false} \equiv e \ \wedge \neg e$ for some fixed $e \in E$, $\mathsf{true} \equiv \neg\mathsf{false}$, $\psi_0 \rightarrow \psi_1 \equiv \neg\psi_0 \vee \psi_1$, $\mathsf{F}^{-1}(\psi) \equiv \mathsf{true} \ \mathsf{S} \ \psi$, $\mathsf{G}^{-1}(\psi) \equiv \neg\mathsf{F}^{-1}(\neg\psi)$. We also define non-standard abbreviation $\sim\!e \equiv \neg\Diamond e$ (pronounced 'conflict e' or 'e is impossible').

3.2 Example Policies

To illustrate the expressive power of our language, we consider a number of example policies.

Example 3.1 (eBay). Recall the eBay scenario from Section 2, in which a buyer has to decide whether to bid on an electronic auction issued by a seller. We express a policy for decision 'bid', stating "only bid on auctions run by a seller that has never failed to send goods for won auctions in the past."

$$\psi^{\mathsf{bid}} \equiv \neg\mathsf{F}^{-1}(\texttt{time-out})$$

Furthermore, the buyer might require that "the seller has never provided negative feedback in auctions where payment was made." We can express this by

$$\psi^{\mathsf{bid}} \equiv \neg\mathsf{F}^{-1}(\texttt{time-out}) \wedge \mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore})$$

Example 3.2 (Chinese Wall). The Chinese Wall policy is an important commercial security-policy [3], but has also found applications within computer science. In particular, Edjlali *et al.* [9] use an instance of the Chinese Wall policy to restrict program accesses to database relations. The Chinese Wall security-policy deals with subjects (e.g. users) and objects (e.g. resources). The objects are organized into *datasets* which, in turn, are organized in so-called *conflict-of-interest classes*. There is a hierarchical structure on objects, datasets and classes, so that each object has a unique dataset which, in turn, has a unique class. In the Chinese-Wall policy, any subject initially has freedom to access any object. After accessing an object, the set of future accessible objects is restricted: the subject can no longer access an object in the same conflict-of-interest class unless it is in a dataset already accessed. Non-conflicting classes may still be accessed.

We now show how our logic can encode any instance of the Chinese Wall policy. Following the model of Brewer *et al.* [3], we let S denote a set of *subjects*, O a set of *objects*, and L a labeling function $L : O \rightarrow C \times D$, where C is a set of *conflict-of-interest classes* and D a set of *datasets*. The interpretation is that if $L(o) = (c_o, d_o)$ for an object $o \in O$, then o is in dataset d_o, and this dataset belongs to the conflict-of-interest class c_o. The hierarchical structure on objects, datasets and classes amounts to requiring that for any $o, o' \in O$ if $L(o) = (c, d)$ and $L(o') = (c', d)$ then $c = c'$. The following 'simple security rule' defines when access is granted to an object o: "either it has the same dataset as an object already accessed by that subject, or, the object belongs to a different conflict-of-interest class." [3] We can encode this rule in our logic. Consider an event structure $ES = (E, \leq, \#)$ where the events are $C \cup D$, with $(c, c') \in \#$ for $c \neq c' \in C$, $(d, d') \in \#$ for $d \neq d' \in D$, and $(c, d) \in \#$ if (c, d) is not in the image of L (denoted $Img(L)$). We take $\leq$ to be discrete. Then a maximal configuration is a set $\{c, d\}$ so that the pair (c, d) is in $Img(L)$, i.e., corresponds

to an object-access. A history is then a sequence of object-accesses. Now stating the simple security rule as a policy is easy: to access object o with $L(o) = (c_o, d_o)$, the history must satisfy the following policy:

$$\psi^o \equiv \mathsf{F}^{-1} d_o \vee \mathsf{G}^{-1} \neg c_o$$

In this encoding we have one policy per object o. One may argue that the policy ψ^o only captures Chinese Wall for a single object (o), whereas the "real" Chinese Wall policy is *a single policy* stating that "*for every object* o, the simple security rule applies." However, in practical terms this is inessential. Even if there are infinitely many objects, a system implementing Chinese Wall one could easily be obtained using our policies as follows. Say that our proposed security mechanism (intended to implement "real" Chinese Wall) gets as input the object o and the subject s for which it has to decide access. Assuming that our mechanism knows function L, it does the following. If object o has never been queried before in the run of our system, the mechanism generates "on-the-fly" a new policy ψ^o according to the scheme above; it then checks ψ^o with respect to the current history of s.[1] If o has been queried before it simply checks ψ^o with respect to the history of s. Since only finitely many objects can be accessed in any finite run, only finitely many different policies are generated. Hence, the described mechanism is operationally equivalent to Chinese Wall.

Example 3.3 (Shallow One-Out-of-k). The 'one-out-of-k' (OOk) access-control policy was introduced informally by Edjlali *et al.* [9]. Set in the area of access control for mobile code, the OOk scheme dynamically classifies programs into equivalence classes, e.g. "browser-like applications," depending on their past behaviour. In the following we show that, if one takes the *set-based* formalization of OOk by Fong [11], we can encode all OOk policies. Since our model is sequence-based, it is richer than Fong's shallow histories which are sets. An encoding of Fong's OOk-model thus provides a good sanity-check as well as a *declarative* means of specifying OOk policies (as opposed to the more implementation-oriented security automata).

In Fong's model of OOk, a finite number of application classes are considered, say, $1, 2, \ldots, k$. Fong identifies an application class, i, with a *set of allowed actions* C_i. To encode OOk policies, we consider an event structure $ES = (E, \leq, \#)$ with events E being the set of all access-controlled actions. As in the last example, we take $\leq$ to be discrete, and the conflict relation to be the maximal irreflexive relation, i.e. a local interaction history in ES is simply a sequence of single events. Initially, a monitored entity (originally, a piece of mobile code [9]) has taken no actions, and its history (which is a set in Fong's formalization) is $\emptyset$. If S is the current history, then action $a \in E$ is allowed if there exists $1 \leq i \leq k$ so that $S \cup \{a\} \subseteq C_i$, and the history is updated to $S \cup \{a\}$. For each action $a \in E$ we define a policy ψ^a for a, expressing Fong's requirement. Assume, without loss of generality, that the sets C_j that contain a are named $1, 2, \ldots, i$ for some $i \leq k$. We will assume that each set C_j is either finite or co-finite.

Fix a $j \leq i$. The following formula ψ^a_j encodes the requirement that $S \cup \{a\} \subseteq C_j$. There are two cases. If the set C_j is co-finite (i.e., its complement $E \setminus C_j$ is finite),

$$\psi^a_j \equiv \neg \mathsf{F}^{-1}(\bigvee_{e \in E \setminus C_j} e)$$

If instead C_j is itself finite, we encode

$$\psi^a_j \equiv \mathsf{G}^{-1}(\bigvee_{e \in C_j} e)$$

Now we can encode the policy for allowing action a as $\psi^a \equiv \bigvee_{j=1}^{i} \psi^a_j$.

4. DYNAMIC MODEL CHECKING

The problem of verifying a policy with respect to a given observed history is the model-checking problem: given $h \in \mathcal{C}^+_{ES}$ and ψ, does $h \models \psi$ hold? However, our intended scenario requires a more dynamic view. Each entity will make many decisions, and each decision requires a model check. Furthermore, since the model h changes as new observations are made, it is not sufficient simply to cache the answers. This leads us to consider the following *dynamic* problem. Devise an implementation of the following interface, '*DMC*'. *DMC* is initially given an event structure $ES = (E, \leq, \#)$ and a policy ψ written in the basic policy language. Interface *DMC* supports three *operations*: *DMC*.**new**(), *DMC*.**update**(e, i), and *DMC*.**check**(). A sequence of non-'check' operations gives rise to a local interaction history h, and we shall call this the *actual history*. Internally, an implementation of *DMC* must maintain information about the actual history h, and operations **new** and **update** are those of Section 2, performed on h. At any time, operation *DMC*.**check**() must return the truth of $h \models \psi$.

In the full paper [17], we describe two implementations of interface *DMC*. The first has a cheap precomputation, but higher complexity of operations **update** and **new**, whereas the second implementation has a higher time- and space-complexity for its precomputation, but gains in the long run with a better complexity of the interface operations. Both implementations are inspired by the efficient algorithm of Havelund and Roşu for model checking past-time LTL [13]. Their idea is essentially this: because of the semantics, model-checking ψ in (h, m), i.e. deciding $(h, m) \models \psi$, can be done easily if one knows (1) the truth of $(h, m-1) \models \psi_j$ for all sub-formulas ψ_j of ψ, and (2) the truth of $(h, m) \models \psi_i$ for all proper sub-formulas ψ_i of ψ (a sub-formula of ψ is proper if it is not ψ itself). The truth of the atomic sub-formulas of ψ in (h, m) can be computed directly from the state h_m, where h_m is the mth configuration in sequence h. For example, if $\psi_3 = \mathsf{X}^{-1}\psi_4 \wedge e$, then $(h, m) \models \psi_3$ iff $(h, m-1) \models \psi_4$, and $e \in h_m$. This information needed to decide $(h, m) \models \psi$ can be stored efficiently as two boolean arrays B_{last} and B_{cur}, indexed by the sub-formulas of ψ, so that $B_{last}[j]$ is true iff $(h, m-1) \models \psi_j$, and $B_{cur}[i]$ is true iff $(h, m) \models \psi_i$. Given array B_{last} and the current state h_m, one then constructs array B_{cur} starting from the atomic formulas (which have the largest indices), and working in a 'bottom-up' manner towards index 0, for which entry $B_{cur}[0]$ represents $(h, m) \models \psi$.

In this section we summarize our results regarding dynamic model checking. We need some preliminary terminology. Initially, the actual interaction history h is empty, but after some time, as observations are made, the history can be written $h = x_1 \cdot x_2 \cdots x_M \cdot y_{M+1} \cdots y_{M+K}$, consisting

[1]This check can be done in time linear in the history of subject s.

of a *longest prefix* $x_1 \cdots x_M$ of *maximal* configurations, followed by a suffix of K possibly non-maximal configurations $y_{M+1} \cdots y_{M+K}$, called the *active* sessions (since we consider the longest prefix, y_{M+1} must be non-maximal). A maximal configuration represents complete information about a protocol-run, and has the property that it will never change in the future, i.e. cannot be changed by operation **update**. This property will be essential to our dynamic algorithms as it implies that the maximal prefix needs not be stored to check $h \models \psi$ dynamically.

Theorem 4.1 (Array-based *DMC*). *One can construct an array-based data structure (DS) implementing the DMC interface correctly. More specifically, assume that DS is initialized with a policy ψ and an event structure ES, then initialization of DS is $O(|\psi|)$. At any time during execution, the complexity of the interface operations is:*

- *DMC*.**check**() *is* $O(1)$.
- *DMC*.**new**() *is* $O(|\psi|)$.
- *DMC*.**update**(e, i) *is* $O((K - i + 1) \cdot |\psi|)$ *where K is the current number of active sessions in h (h is the current actual history).*

Furthermore, if the configurations of ES are represented with event-set bit-vectors, the space complexity of DS is $O(K \cdot (|\psi| + |E|))$.

Regularity of policies. In fact, it turns out that the set of behaviours satisfying a policy is a regular language (over the alphabet of configurations of a finite event structure). This observation leads to an implementation of the *DMC* interface which uses a finite automaton, essentially storing the state of the array-based data structure. We have the following.

Theorem 4.2. *For any policy ψ in the basic language, the set of behaviours satisfying ψ is regular. That is for any ψ there is a finite automaton A_ψ with $\mathcal{L}(A_\psi) = \{h \in \mathcal{C}_{ES}^* \mid h \models \psi\}$. Further, there exists an automata-based data structure (DS′) implementing the DMC interface correctly. More specifically, assume that DS′ is initialized with a policy ψ and an event structure $ES = (E, \leq, \#)$, then initialization of DS′ is $O(2^{|\psi|} \cdot |\mathcal{C}_{ES}| \cdot |\psi|)$. At any time during execution, the complexity of the interface operations is:*

- *DMC*.**check**() *is* $O(1)$.
- *DMC*.**new**() *is* $O(1)$.
- *DMC*.**update**(e, i) *is* $O(K - i + 1)$ *where K is the current number of active configurations in h (h is the current actual history).*

Furthermore, if the configurations of ES are represented with event-set bit-vectors, the space complexity of DS′ is $O(K \cdot |E| + 2^{|\psi|} \cdot |\mathcal{C}_{ES}|)$.

A further important advantage of the automata-based approach is that one can use minimization to obtain the most efficient automata for a given policy.

5. LANGUAGE EXTENSIONS

In this section, we consider an extension of the basic policy language to include more realistic and practical policies. For example, consider the OO*k* policy for classifying "browser-like" applications (Section 3). We could use a clause like $\mathsf{G}^{-1}(\texttt{open-f} \rightarrow \mathsf{F}^{-1}\texttt{create-f})$ for two events `open-f` and `create-f`, representing respectively the opening and creation of a file with name f. However, this only encodes the requirement that for a *fixed* f, file f must be created before it is opened. Ideally, one would want to encode that for *any* file, this property holds, i.e., a formula similar to

$$\mathsf{G}^{-1}\left(\forall x.\left[\texttt{open}(x) \rightarrow \mathsf{F}^{-1}(\texttt{create}(x))\right]\right)$$

where x is a variable, and the universal quantification ranges over all possible file-names. Further language extensions are discussed in the full paper [17]. These includes a notion of *policy referencing*, where policies may depend on other agents' policies and histories with entities. Another useful extension for reputation systems is *quantitative policies*. Pure-past temporal logic is very useful for specifying qualitative properties. For instance, in the eBay example, "the seller has never provided negative feedback in auctions where payment was made," is directly expressible as $\mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore})$. However, sometimes such qualitative properties are too strict to be useful in practice. For example, in the policy above, a single erroneous negative feedback provided by the seller will lead to the property being irrevocably unsatisfiable. We have an extension of the basic language which allows a type of *quantitative* properties, e.g. "in at least 98% of the previous interactions, seller has not provided negative feedback in auctions where payment was made."

5.1 Quantification

We introduce a notion of parameterized event structure, and proceed with an extension of the basic policy language to include quantification over parameters. A parameterized event structure is like an ordinary event structure, but where events occur with certain parameters (e.g. `open`("/etc/passwd")).

5.1.1 Parameterized Event Structures

We define parameterized event structures and an appropriate notion of configuration.

Definition 5.1 (Parameterized Event Structure). A *parameterized event structure* is a tuple $\rho ES = (E, \leq, \#, \mathcal{P}, \rho)$ where $(E, \leq, \#)$ is an (ordinary) event structure, component $\mathcal{P}$, called the *parameters*, is a set of countable *parameter sets*, $\mathcal{P} = \{P_e \mid e \in E\}$, and $\rho : E \rightarrow \mathcal{P}$ is a function, called the *parameter-set assignment*.

Definition 5.2 (Configuration). Let $\rho ES = (E, \leq, \#, \mathcal{P}, \rho)$ be a parameterized event structure. A *configuration* of ρES is a partial function $x : E \rightarrow \bigcup_{e \in E} \rho(e)$ satisfying the following two properties. Let $dom(x) \subseteq E$ be the set of events on which x is defined. Then

$$dom(x) \in \mathcal{C}_{ES}$$

$$\forall e \in dom(x).x(e) \in \rho(e)$$

When x is a configuration, and $e \in dom(x)$, then we say that e has occurred in x. Further, when $x(e) = p \in \rho(e)$,

we say that e has occurred with parameter p in x. So a configuration is a set of event occurrences, each occurred event having exactly one parameter.

Notation 5.1. We write $\mathcal{C}_{\rho ES}$ for the set of configurations of ρES, and $\mathcal{C}^0_{\rho ES}$ for the set of *finite* configurations of ρES (a configuration x is finite of $dom(x)$ is finite). If x, y are two partial functions $x : A \to B$ and $y : C \to D$ we write (x/y) (pronounced x over y) for the partial function $(x/y) : A \cup B \to C \cup D$ given by $dom(x/y) = dom(x) \cup dom(y)$, and for all $e \in dom(x/y)$ we have $(x/y)(e) = x(e)$ if $e \in dom(x)$ and otherwise $(x/y)(e) = y(e)$.

Here we are not interested in the theory of parameterized event structures, but mention only that they can be explained in terms of ordinary event structures by expanding a parameterized event e of type $\rho(e)$ in to a set of conflicting events $\{(e, p) \mid p \in \rho(e)\}$. However, the parameters give a convenient way of saying that the *same* event can occur with different parameters (in different runs). A local (interaction) history h in a parameterized event structure ρES is a finite sequence $h \in {\mathcal{C}^0_{\rho ES}}^*$. The **update**$(h, e, i)$ function is extended appropriately to include also the parameter p that e occurs with. Throughout the following sections, we let $\rho ES = (E, \leq, \#, \mathcal{P}, \rho)$ be a parameterized event structure, where $\mathcal{P} = \{P_i \mid i \in \mathbb{N}\}$.

5.1.2 Quantified Policies

We extend the basic language from Section 3 to parameterized event structures, allowing quantification over parameters.

Syntax. Let *Var* denote a countable set of variables (ranged over by $x, y, \ldots$). Let the meta-variable v range over $Val \stackrel{(def)}{=} Var \cup \bigcup_{i=1}^{\infty} P_i$, and metavariable p range over $\bigcup_{i=1}^{\infty} P_i$.

The quantified policy language is given by the following BNF. Again *op* ranges over $\{\wedge, \vee\}$.

$$\psi \;::=\; e(v) \mid \Diamond e(v) \mid \psi_0 \; op \; \psi_1 \mid \neg\psi \mid \mathsf{X}^{-1}\psi \mid \psi_0 \; \mathsf{S} \; \psi_1 \mid \forall x : P_i.\psi \mid \exists x : P_i.\psi$$

We need some terminology. Write $fv(\psi)$ for the set of free variables in ψ (defined in the usual way). A *policy* of the quantified language is a closed formula. Let ψ be any formula. Say that a variable x has type P_i in ψ if it occurs in a sub-formula $e(x)$ of ψ and $\rho(e) = P_i$. We impose the following static well-formedness requirement on formulas ψ. All free variables have unique type, and, if x is a bound variable of type P_i in ψ, then x is bound by a quantifier of the correct type (e.g., by $\forall x : P_i.\psi$). Further, for each occurrence of $e(p)$, p is of the correct type: $p \in \rho(e)$.

Semantics. A (generalized) substitution is a function $\sigma : Val \to \bigcup_{i=1}^{\infty} P_i$ so that σ is the identity on each of the parameter sets P_i. Let $h = x_1 \cdots x_n \in {\mathcal{C}^0_{\rho ES}}^*$ be a non-empty history, σ a substitution, and $1 \leq i \leq n$.

We now define relation $(h, i) \models^\sigma \psi$.

$$\begin{array}{lll}
(h,i) \models^\sigma e(v) & \text{iff} & e \in dom(x_i) \text{ and } x_i(e) = \sigma(v) \\
(h,i) \models^\sigma \Diamond e(v) & \text{iff} & e \;\#\; dom(x_i) \text{ and} \\
 & & (e \in dom(x_i) \Rightarrow x_i(e) = \sigma(v)) \\
(h,i) \models^\sigma \psi_0 \wedge \psi_1 & \text{iff} & (h,i) \models^\sigma \psi_0 \text{ and } (h,i) \models^\sigma \psi_1 \\
(h,i) \models^\sigma \psi_0 \vee \psi_1 & \text{iff} & (h,i) \models^\sigma \psi_0 \text{ or } (h,i) \models^\sigma \psi_1 \\
(h,i) \models^\sigma \neg\psi & \text{iff} & (h,i) \not\models^\sigma \psi \\
(h,i) \models^\sigma \mathsf{X}^{-1}\psi & \text{iff} & i > 1 \text{ and } (h, i-1) \models^\sigma \psi \\
(h,i) \models^\sigma \psi_0 \;\mathsf{S}\; \psi_1 & \text{iff} & \exists j \leq i.\big((h,j) \models^\sigma \psi_1\big) \text{ and} \\
 & & [\forall j < j' \leq i.(h,j') \models^\sigma \psi_0]\big) \\
(h,i) \models^\sigma \forall x : P_j.\psi & \text{iff} & \forall p \in P_j.(h,i) \models^{((x \mapsto p)/\sigma)} \psi \\
(h,i) \models^\sigma \exists x : P_j.\psi & \text{iff} & \exists p \in P_j.(h,i) \models^{((x \mapsto p)/\sigma)} \psi
\end{array}$$

Example 5.1 (True OOok). Recall the 'one-out-of-k' policy (Example 3.3). Edjlali *et al.* give, among others, the following example of an OOok policy classifying "browser-like" applications: "allow a program to connect to a remote site if and only if it has neither tried to open a local file that it has not created, nor tried to modify a file it has created, nor tried to create a sub-process." Since this example implicitly quantifies over all possible files (for *any* file f, if the application tries to open f then it must have previously have created f), it cannot be expressed directly in our basic language. Note also that this policy cannot be expressed in Fong's set-based model [11]. This follows since the above policy essentially depends on the *order* in which events occur (i.e. `create` before `open`). Now, consider a parameterized event structure with two conflicting events: `create` and `open`, each of type *String* (representing file-names). Consider the following quantified policy:

$$\mathsf{G}^{-1}(\forall x : \mathit{String}.(\texttt{open}(x) \to \mathsf{F}^{-1}\texttt{create}(x)))$$

This faithfully expresses the idea of Edjlali *et al.* that the application "can only open files it has previously created."

5.1.3 Model Checking the Quantified Language

We can extend the array-based algorithm to handle the quantified language. The key idea is the following. Instead of having *boolean* arrays, we associate with each sub-formula ψ_j of a formula ψ, a *constraint* $C_k[j]$ on the free variables of ψ_j. The invariant will be that the sub-formula ψ_j is true *for a substitution* σ at time (h, k) if-and-only-if σ "satisfies" the constraint $C_k[j]$, i.e., $C_k[j]$ represents the set of substitutions σ so $(h, k) \models^\sigma \psi_j$. Once again we refer the reader to the full paper for details. The results regarding quantified dynamic model-checking are summarized below. However, we do have the following hardness result.

Proposition 5.1 (PSPACE Hardness). Even for single element models, the model-checking problem for the quantified policy language is PSPACE hard.

While the general problem is PSPACE hard, we are able to obtain the following quantitative result which bounds the complexity of our algorithm. Suppose we are to check a formula $\psi' \equiv Q_1 x_1 Q_2 x_2 \cdots Q_n x_n.\psi$, where the Q_i are quantifiers and x_i variables. We can obtain a bound on the running time of our proposed algorithm in terms of the number of quantifiers n. This is of practical relevance since many useful policies have few quantifiers. For any history h, P_h refers to the (finite) set of distinct parameters that have occurred in h. The requirement below that all variables be of same type is to simplify presentation, and not essential.

Theorem 5.1 (Complexity Bound). Let formula $\psi \equiv Q_1 x_1 Q_2 x_2 \cdots Q_n x_n.\psi'$ where the Q_i are quantifiers, x_i variables all of type P, and ψ' is a quantifier-free formula from the quantified language with $fv(\psi') \subseteq \{x_1, \ldots, x_n\}$. Let $h \in {\mathcal{C}^0_{\rho ES}}^*$ and $|P_h|$ be the number of parameter occurrences in history h. The constraint-based algorithm for dynamic model checking has the following complexity.

- DMC.**check**() is $O(1)$.
- DMC.**new**() is $O(|\psi| \cdot (|P_h| + 1)^n)$.
- DMC.**update**(e, p, i) when $p \in P_h$ and K is the current number of active configurations in h, is $O((K - i + 1) \cdot |\psi| \cdot (|P_h| + 1)^n)$
- DMC.**update**(e, p, i) when $p \notin P_h$ and K is the current number of active configurations in h, is $O((K - i + 1) \cdot |\psi| \cdot (|P_h| + 2)^n)$

Furthermore, if the configurations of ES are represented with event-set bit-vectors, the space complexity of DS' is $O(K \cdot (|E| + |\psi| \cdot (|P_h| + 1)^n))$.

6. CONCLUSION

Our approach to reputation-systems differs from most existing systems in that reputation information has an exact semantics, and is represented in a very concrete form. In our view, the novelty of our approach is that our instance systems can verifiably provide a form of exact *security guarantees*, albeit non-standard, that relate a *present authorization* to a precise property of *past behaviour*. We have presented a declarative language for specifying such security properties, and the applications of our technique extends beyond the traditional domain of reputations systems in that we can explain, formally, several existing approaches to "history based" access control.

We have given two efficient algorithms for the dynamic model-checking problem, supporting the feasibility of running implementations of our framework on devices of limited computational and storage capacity; a useful property in global computing environments. In particular, it is noteworthy that principals need not store their entire interaction histories, but only the so-called active sessions.

The notion of time in our temporal logic is based on when sessions are *started*. More precisely, our models are local interaction histories, $h = x_1 x_2 \cdots x_n$ where $x_i \in \mathcal{C}_{ES}$, and the order of the sessions reflects *the order in which the corresponding interaction-protocols are initiated*, i.e. x_i refers to the observed events in the ith-initiated session. Different notions of time could just as well be considered, e.g. if x_i precedes x_j in sequence h, then it means that x_j was updated more recently than x_i (our algorithms can be straightforwardly be adapted to this notion of time).

Related Work. Many reputation-based systems have been proposed in the literature (Jøsang *et al.* [15] provide many references), so we choose to mention only a few typical examples and closely related systems. Kamvar *et al.* present EigenTrust [16], Shmatikov and Talcott propose a license-based framework [27], and the EU project 'SECURE' [4, 5] (which also uses event structures for modelling observations) can be viewed as a reputation-based system, to name a notable few.

The framework of Shmatikov and Talcott is the most closely related in that they deploy also a very concrete representation of behavioural information ("evidence" [27]). This representation is not as sophisticated as in the event-structure framework (e.g., as histories are sets of time-stamped events there is no concept of a session, i.e., a logically connected set of events), and their notion of reputation is based on an entity's past ability to fulfill so-called licenses. A license is a contract between an issuer and a licensee. Licenses are more general than interaction policies since they are *mutual* contracts between issuer and licensee, which may *permit* the licensee to perform certain actions, but may also *require* that certain actions are performed. The framework does not have a domain-specific language for specifying licenses (i.e. for specifying license-methods **permits** and **violated**), and the *use* of reputation information is not part of their formal framework (i.e. it is up to each application programmer to write method **useOk** for protecting a resource). We do not see our framework as competing, but, rather, *compatible* with theirs. We imagine using a policy language, like ours, as a domain-specific language for specifying licenses as well as use-policies. We believe that because of the simplicity of our declarative policy language and its formal semantics, this would facilitate verification and other reasoning about instances of their framework.

Pucella and Weissman use a variant of pure-future linear temporal logic for reasoning about licenses [23]. They are not interested in the specific details of licenses, but merely require that licenses can be given a trace-based semantics; in particular, their logic is illustrated for licenses that are regular languages. As our basic policies can be seen (semantically) as regular languages (Theorem 4.2), and policies can be seen as a type of license, one could imagine using their logic to reason about our policies.

Roger and Goubault-Larreq [25] have used linear temporal logic and associated model-checking algorithms for log auditing. The work is related although their application is quite different. While their logic is first-order in the sense of having variables, they have no explicit quantification. Our quantified language differs (besides being pure-past instead of pure-future) in that we allow explicit quantification (over different parameter types) $\forall x : P_i.\psi$ and $\exists x : P_i.\psi$, while their language is implicitly universally quantified.

The notion of security automata, introduced by Schneider [26], is related to our policy language. A security automaton runs in parallel with a program, monitoring its execution with respect to a security policy. If the automata detects that the program is about to violate the policy, it terminates the program. A policy is given in terms of an automata, and a (non-declarative) domain-specific language for defining security automata (SAL) is supported but has been found awkward for policy specification [10]. One can view the finite automaton in our automata-based algorithm as a kind of security automaton, *declaratively* specified by a temporal-logic formula.

Security automata are also related, in a technical sense [11], to the notion of history-based access control (HBAC). HBAC has been the subject of a considerable amount of research (e.g., papers [1, 9, 11, 12, 26, 29]). There is a distinction between *dynamic* HBAC in which programs are monitored as they execute, and terminated if about to violate policy [9, 11, 12, 26]; and *static* HBAC in which some preliminary static analysis of the program (written in a pre-

determined language) extracts a safe approximation of the programs' runtime behaviour, and then (statically) checks that this approximation will always conform to policy (using, e.g., type systems or model checking) [1,29]. Clearly, our approach has applications to dynamic HBAC. It is noteworthy to mention that many ad-hoc optimizations in dynamic HBAC (e.g., history summaries relative to a policy in the system of Edjlali [9]) are captured in a *general* and optimal way by using the automata-based algorithm, and exploiting the finite-automata minimization-theorem. Thus in the automata based algorithm, one gets "for free," optimizations that would otherwise have to be discovered manually.

7. REFERENCES

[1] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005. Proceedings*, pages 316–332. Springer, 2005.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In J. Vitek and C. D. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 1999.

[3] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings from the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, 1989.

[4] V. Cahill and E. Gray *et al.* Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, 2003.

[5] V. Cahill and J.-M. Seigneur. The SECURE website. `http://secure.dsg.cs.tcd.ie`, 2004.

[6] M. Carbone, M. Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *Proceedings from Software Engineering and Formal Methods (SEFM'03)*. IEEE Computer Society Press, 2003.

[7] M. Carbone, M. Nielsen, and V. Sassone. A calculus for trust management. In *Proceedings from Foundations of Software Technology and Theoretical Computer Science: 24th International Conference (FSTTCS'04)*, pages 161–173. Springer, December 2004.

[8] eBay Inc. The eBay website. `http://www.ebay.com`.

[9] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings from the 5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 38–48. ACM Press, 1998.

[10] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings from the 2000 DARPA Information Survivability Conference and Exposition*, pages 1287–1295. IEEE Computer Society Press, 2000.

[11] P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings from the 2004 IEEE Symposium on Security and Privacy*, pages 43–55. IEEE Computer Society Press, 2004.

[12] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

[13] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2002.

[14] A. Jøsang and R. Ismail. The beta reputation system. In *Proceedings from the 15th Bled Conference on Electronic Commerce, Bled*, 2002.

[15] A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation for online service provision. Decision Support Systems, (to appear, preprint available online: `http://security.dstc.edu.au/staff/ajosang`), 2004.

[16] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proceedings from the twelfth international conference on World Wide Web, Budapest, Hungary*, pages 640–651. ACM Press, 2003.

[17] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems. Technical Report RS-05-23, BRICS, University of Aarhus, July 2005.

[18] K. Krukow and A. Twigg. Distributed approximation of fixed-points in trust structures. In *Proceedings from the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 805–814. IEEE, 2005.

[19] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *Proceedings from the 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 383–392. IEEE Computer Society Press, 2002.

[20] M. Nielsen and K. Krukow. Towards a formal notion of trust. In *Proceedings from the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 4–7. ACM Press, 2003.

[21] M. Nielsen and K. Krukow. On the formal modelling of trust in reputation-based systems. In J. Karhumäki, H. Maurer, G. Paun, and G. Rozenberg, editors, *Theory Is Forever: Essays Dedicated to Arto Salomaa*, volume 3113 of *Lecture Notes in Computer Science*, pages 192–204. Springer Verlag, 2004.

[22] A. Pnueli. The temporal logic of programs. In *Proceedings from the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE, New York, 1977.

[23] R. Pucella and V. Weissman. A logic for reasoning about digital rights. In *Proceedings from 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 282–294. IEEE Computer Society Press, 2002.

[24] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara. Reputation systems. *Communications of the ACM*, 43(12):45–48, Dec. 2000.

[25] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *Proceedings from the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236. IEEE Computer Society Press, 2001.

[26] F. B. Schneider. Enforceable security policies. *Journal of the ACM*, 3(1):30–50, 2000.

[27] V. Shmatikov and C. Talcott. Reputation-based trust management. *Journal of Computer Security*, 13(1):167–190, 2005.

[28] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[29] C. Skalka and S. Smith. History effects and verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, pages 107–128. Springer, 2005.

[30] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, 1995.