A Logical Framework for Reputation Systems

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

4th January 2006

Abstract

Reputation systems are meta systems that record, aggregate and distribute information about the past behaviour of principals in an application. Typically, these applications are large-scale open distributed systems where principals are virtually anonymous, and (a priori) have no knowledge about the trustworthiness of each other. Reputation systems serve two primary purposes: helping principals decide whom to trust, and providing an incentive for principals to well-behave.

A logical policy-based framework for reputation systems is presented. In the framework, principals specify policies which state precise requirements on the past behaviour of other principals that must be fulfilled in order for interaction to take place. The framework consists of a formal model of behaviour, based on event structures; a declarative logical language for specifying properties of past behaviour; and efficient dynamic algorithms for checking whether a particular behaviour satisfies a property from the language. It is shown how the framework can be extended in several ways, most notably to encompass parameterized events and quantification over parameters. In an extended application, it is illustrated how the framework can be applied for dynamic history-based access control for safe execution of unknown and untrusted programs.

Keywords. Reputation systems, history-based access control, trust management, dynamic model checking, event structures, linear temporal logic.

*Corresponding author. Department of Computer Science, University of Aarhus, IT-Parken, Aabogade 32, DK-8200 Aarhus N., Denmark. Tel. +45-89425778, Fax. +45-89425601.

1 Introduction

Rich opportunities for fraud exist on the Internet. Still, risky interactions like electronic commerce, involving disclosure of private informations to semi-trusted parties, are every-day activities in our Internet lives. It seems that in practice, for most people, the utility of the Internet outweighs its risks. When one tries to understand better these facts, mathematical models from economic theory are very appealing. Online interaction can often be seen as a 'repeated game' played between selfish (semi) rational principals. Such interaction may result in utility gains for the involved principals, but often, with interaction comes also an associated inherent risk; a potential utility-loss. For risk-adverse principals, the fear of loss may outweigh the expectation of gain, leading to an unwillingness to participate. For example, one might have expected that an online auctioning system such as eBay, "a market ripe with the possibility of large-scale fraud and deceit" [19], would never have reached the more than one million transactions per day that are presently processed. The liveness on eBay is often attributed to its so-called Feedback Forum, a simple example of a reputation system. When principals have transacted, each party may leave feedback on the eBay website, consisting of a rating of 'positive', 'neutral' or 'negative'. A principal's aggregated rating is then visible to potential buyers or sellers before deciding whether to interact or not. In general, reputation systems record, aggregate and (sometimes) distribute information about the past behaviour of principals. Hence reputation systems may serve as a trust-enabling, or perhaps, more generally, trust-*informing* technology. Resnick et al. argue that reputation systems foster an incentive for principals to well-behave because of "the expectation of reciprocity or retaliation in future interactions" [28], and reputation itself has previously been formalized and analyzed by economists in simple game-theoretic models, leading to similar conclusions (e.g., [6,7,20,36]); it seems that reputation systems are well etablished, and their is usefulness generally accepted.

Many reputation systems have been proposed in the literature [15], and often the recorded behavioural information is heavily abstracted. This has the effect that several quite different concrete behaviours are collapsed in to the same "equivalence class" of recorded behaviours. For example, the eBay-rating of 'negative' may be the (subjective) result of several distinct seller behaviours: the seller may never ship the auctioned item, the item may be in a poor condition, a certain timeliness is expected, credit cards may be overcharged because of, say, shipping fees, etc. Different users will be interested in the actual meaning of the rating 'negative'; the *concrete behaviour* of the seller. There are other examples: In the EigenTrust system [17], 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). The only non-example of such crude information abstraction (that we are aware of) is the framework of Shmatikov and Talcott [31] 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" [17]. 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.* A *principal* is simply an identity; it may be the identity of a human users, a public key, a software program (e.g., an identifiable instance), etc. 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:

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

Informally, the atoms `modify-file`, `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 $\neg\mathsf{F}^{-1}(\texttt{modify-file})$ 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 [37] for modelling observations [25], 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) [26]. 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 [2] 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. In Section 4, two dynamic algorithms for policy checking is described, analysed and compared.

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, 33], and within the traditional domain of reputation systems: peer-to-peer- and online feedback systems [17, 28]). More specifically, we present two extensions. The first is quantification (as is used in the example policy in the introductory section). We extend the basic language, allowing parameterized events and quantification over the parameters. An algorithm for checking the extended language along with complexity analyses is provided. The second extension covers the two aspects of *information sharing*, and *quantitative properties.* We introduce constructs that allow principals to state properties, not only of their personally-observed behaviour, but also of the behaviour observed by others (in the terminology of Mui *et al.* [22], the first is *direct* and *encounter driven*, and the latter, *indirect* and *propagated*). Such information sharing is characteristic of most existing reputation systems. Another common characteristic is focus on conveying quantitative information. In contrast, standard temporal logic is qualitative: it deals with concepts such as *before*, *after*, *always* and *eventually*. We show that we can extend our language to include a range of quantitative aspects, intuitively, operators like 'almost always,' 'more than N,' etc. Section 5 illustrates these two extensions, and briefly discusses policy-checking for the extended languages.

Throughout the paper, we have small examples illustrating the applicability of our framework within the area of history-based access control. We have taken this one step further by developing a prototype security manager for Java, based on our logical framework. The security manager is parameterized by a policy in our language, and monitors a Java program with respect to this policy, throwing an exception if a violation is about to happen. In Section 6, we describe this application of our framework to history-based access control for Java programs. Related work is discussed in the concluding section.

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 [5]. 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 [25] 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, 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 [37], traditionally used in semantics of concurrent languages, consult the handbook chapter of Winskel and Nielsen [38]).

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 $e \in E$:

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

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 $e \in E$

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

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

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

Remarks. The notion of *time* in the model is based on when sessions are *started*. More precisely, in our local interaction histories, $h = x_1 x_2 \cdots x_n$ where $x_i \in \mathcal{C}_{ES}$, 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.

Note, 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

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

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 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.” Notice, by the way, that users would have a hard time implementing such a policy using the current eBay feedback forum.

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 [26]. 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 for each $e \in E$ (ranged over by meta-variables $e, e', e_i, \ldots$). The syntax of our language, which we denote $\mathcal{L}(ES)$, is given by the following BNF.

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

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, $h \in {\mathcal{C}_{ES}^0}^+$. 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 $h = x_1 x_2 \cdots x_N \in {\mathcal{C}_{ES}^0}^*$, and $i \in \mathbb{N}$. Define $(h, i) \models \psi$ by structural induction in ψ.

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

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 [21]. 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 [32], 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 the logic cannot distinguish the empty structure $\epsilon \in \mathcal{C}_{ES}^*$ from a structure consisting of any number of empty configurations, e.g., $\emptyset\emptyset\emptyset$. More generally, one way of looking at our structures is as *infinite* sequences $x_1 x_2 \cdots x_n \emptyset\emptyset \cdots$, having only finitely many non-empty configurations.

We define standard abbreviations using syntatic equality: $\mathsf{false} \equiv e \ \wedge \neg e$ for some fixed $e \in E$, $\mathsf{true} \equiv \neg\mathsf{false}$, $\psi_0 \vee \psi_1 \equiv \neg(\neg\psi_0 \wedge \neg\psi_1)$, $\psi_0 \rightarrow \psi_1 \equiv \neg\psi_0 \vee \psi_1$, $\mathsf{F}^{-1}(\psi) \equiv \mathsf{true}\ \mathsf{S}\ \psi$, $\mathsf{G}^{-1}(\psi) \equiv \neg\mathsf{F}^{-1}(\neg\psi)$. Note that, $\mathsf{F}^{-1}(\psi)$ means "formula ψ is true at *some time* in the past," whereas $\mathsf{G}^{-1}(\psi)$ means "ψ is true at *all times* in the past." We also define a 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 [2], 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.* [2], 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." [2] 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 $\mathit{Img}(L)$). We take $\leq$ to be discrete. Then a maximal configuration is a set $\{c, d\}$ so that the pair $(c, d) \in \mathit{Img}(L)$, corresponding 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

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

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 $j \leq i$. If the set C_j is co-finite (i.e., its complement $E \setminus C_j$ is finite), the following formula ψ_j^a encodes the requirement that $S \cup \{a\} \subseteq C_j$.

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

If instead C_j is itself finite, we encode

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

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

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 this section, 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 very efficient algorithm of Havelund and Roşu for model checking past-time

LTL [13]. Their idea is essentially this: because of the recursive 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 $(h, m) \models \psi$. We shall generalize this idea of Havelund and Roşu to obtain an algorithm for the dynamic problem.

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 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.

In the following, the number M will always refer to the size of the maximal prefix, and K to the size of the suffix.

4.1 An Array-based Implementation

We describe an implementation of the *DMC* interface based on a data structure *DS* maintaining the active sessions and a collection of boolean arrays. Understanding the data structure is understanding the invariant it maintains, and we will describe this in the following.

The data structure *DS* has a vector, accessed by variable $DS.h$, storing configurations of ES, which we denote $DS.h = (y_1, y_2, \ldots, y_K)$. Part of the invariant is that $DS.h$ stores only the suffix of active configurations, i.e. the *actual history* h can be written $h = x_1 \cdot x_2 \cdots x_M \cdot (DS.h)$, where the x_i are all maximal.

Initialization. The data structure is initialized with (a representation of) an event structure $ES = (E, \leq, \#)$ and a policy ψ. We assume that the representation of the configurations of ES, $x \in \mathcal{C}_{ES}$, is so that the membership $e \in x$, conflict $e \;\#\; x$, singleton union $x \cup \{e\}$ and maximality (i.e. is $x \in \mathcal{C}_{ES}$ maximal?) can be computed in constant time. Initialization starts by enumerating the sub-formulas of ψ, denoted $Sub(\psi)$, such that the following property holds.

Let there be $n+1$ sub-formulas of ψ, and let $\psi_0 = \psi$. The sub-formula enumeration $\psi_0, \psi_1, \psi_2, \ldots, \psi_n$ satisfies that if ψ_i is a proper sub-formula of ψ_j then $i > j$.

Invariance. As mentioned, part of the invariant is that $DS.h$ stores exactly the active configurations of the actual history h. In particular, this means that $DS.h_1$ is non-maximal, since otherwise there was a larger longest prefix of h.[2] In addition to $DS.h$, the data structure maintains a boolean array $DS.B_j$ for each entry y_j in the vector $DS.h$. The boolean arrays are indexed by the sub-formulas of ψ (more precisely, by the integers $0, 1, \ldots, n$, corresponding to the sub-formula enumeration). The following invariant will be maintained: $DS.B_k[j]$ is true iff $(h, M+k) \models \psi_j$, that is, if-and-only-if the *actual history* $h = x_1 \cdots x_M \cdot DS.h$ is a model of sub-formula ψ_j at time $M+k$. Additionally, once the longest prefix of maximal configurations becomes non-empty, we allocate a special array B_0, which maintains a "summary" of the entire maximal prefix of h with respect to ψ, meaning that it will satisfy the invariant: $B_0[j]$ is true iff $(h, M) \models \psi_j$.

Operations. The invariants above imply that the model-checking problem $h \models \psi$ can be computed simply by looking at entry 0 of array $DS.B_K$, i.e. $DS.B_K[0]$ is true iff $(h, M+K) \models \psi_0$ iff $h \models \psi$. This means that operation $DS.$**check**$()$ can be implemented in constant time $O(1)$. Operation $DS.$**new** is also easy: the vector $DS.h$ is extended by adding a new entry consisting of the empty configuration. We must also allocate a new boolean array $DS.B_{K+1}$, which is initialized using the recursive semantics, consulting the array $DS.B_K$, and the current state $\emptyset$. This can be done in linear time in the number of sub-formulas of ψ, $O(|\psi|)$.

The final and most interesting operation, is $DS.$**update**(e, i). It is assumed as a pre-condition, that $1 \leq i \leq K$, and that e is not in conflict with $DS.h_i$. First we must add event e to configuration $DS.h_i$, i.e. $DS.h_i$ becomes $DS.h_i \cup \{e\}$. This is simple, but it may break the invariant. In particular, arrays $DS.B_k$ (for $k \geq i$) may no longer satisfy $(h, M+k) \models \psi_j \iff DS.B_k[j] = \texttt{true}$. Note, however, that for any $0 \leq k < i$, the array $DS.B_k$ still maintains its invariant. This is due to the fact that all (sub) formulas are pure-past, and so their truth in h at time k does not depend on configurations *later than* k. In particular, since $i \geq 1$, the special array $DS.B_0$ always maintains its invariant. This means that we can always assume that $DS.B_{i-1}[j]$ is true iff $(h, M+i-1) \models \psi_j$. This information can be used to correctly fill-in array i, in time linear in $|\psi|$, using the recursive semantics. In turn, this can be used to update array $i+1$, and so forth until we have correctly updated array K, and the invariants are restored. Finally, in the case that $i = 1$ and the updated session $DS.h_1$ has become maximal, the updated actual history h now has a larger longest prefix of maximal configurations. We must now find the largest $k \leq K$ so that for all $1 \leq k' \leq k$, $DS.h_{k'}$ is maximal. All arrays $DS.B_{k'}$ and configurations $DS.h_{k'}$ for $k' < k$ may then be deallocated (configuration $DS.h_k$ may also be deallocated),

[2] We do not consider, here, the case where $DS.h$ is empty.

and the new "summarizing" array $DS.B_0$ becomes $DS.B_k$. We summarize the result of this section as a theorem.

Theorem 4.1 (Array-based *DMC*). *The array-based data structure (DS) implements 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 configurations in h (h is the current actual history).*

Furthermore, the space requirement of DS is $O(K + |E| \cdot |\mathcal{C}_{ES}|)$.

4.2 An Automata-based Implementation

In this section, we describe an alternative implementation of the DMC interface. The implementation uses a finite automaton to improve the *dynamic* complexity of the algorithm at the cost of a one-time computation, constructing the automaton.

We consider the problem of model-checking ψ in a history $h = x_1 x_2 \cdots x_{M+K}$ as the string-acceptance problem for an automaton, A_ψ, reading symbols from an alphabet consisting of the finite configurations of ES. The language $\{h \in \mathcal{C}_{ES}^* \mid h \models \psi\}$ turns out to be regular for all ψ in our policy language.

The states of the automaton A_ψ will be boolean arrays of size $|\psi|$, i.e. indexed by the sub-formulas of ψ. Thinking slightly more abstractly about the Havelund-Roşu algorithm, filling the array B_{cur} using B_{last} and the current configuration $x \in \mathcal{C}_{ES}$ can be seen as an automaton transition from state B_{last} to state B_{cur} performed when reading symbol x. We need some preliminary notation.

Let us identify a boolean array B indexed by the sub-formulas of ψ with a set $s \in \mathbf{2}^{Sub(\psi)}$, i.e. $B[j] = \texttt{true}$ iff $\psi_j \in s$. The recursive semantics for a fixed formula ψ, can be seen as an algorithm, denoted *RecSem*, taking as input the array $B_{last} \in \mathbf{2}^{Sub(\psi)}$ and the current configuration $x \in \mathcal{C}_{ES}$, and giving as output $B_{cur} \in \mathbf{2}^{Sub(\psi)}$. Furthermore, the base-case of the recursive semantics can be seen as an algorithm taking only a configuration as input and giving a subset $s \in \mathbf{2}^{Sub(\psi)}$ as output. The input-output behaviour of the recursive-semantics algorithm is exactly the transition function of our automaton.

Definition 4.1 (Automaton A_ψ). Let ψ be a formula in the pure-past policy language $\mathcal{L}(ES)$, where ES is an event structure. Define a deterministic finite automaton $A_\psi = (S, \Sigma, s_0, F, \delta_\psi)$, where $S = \mathbf{2}^{Sub(\psi)} \cup \{s_0\}$ is the set of states, $s_0 \notin \mathbf{2}^{Sub(\psi)}$ being a special initial state, and $\Sigma = \mathcal{C}_{ES}$ is the alphabet. The final states F consist of the set $\{s \in S \mid \psi \in s\}$, and if $\epsilon \models \psi$ then the initial state is also final, i.e. $s_0 \in F$ iff $\emptyset \models \psi$. The transition function restricted to the non-initial states, $\delta_\psi : \mathbf{2}^\psi \times \mathcal{C}_{ES} \rightarrow \mathbf{2}^\psi$, is given by the recursive semantics, i.e. $\delta_\psi(s, x) = \mathit{RecSem}(s, x)$ for all $s \in \mathbf{2}^{Sub(\psi)}, x \in \Sigma$. The transition function on the initial state, $\delta_\psi(s_0, \_)$, is given by the base-case of the recursive semantics.

Since we have identified the empty structure $\epsilon \in \mathcal{C}_{ES}^*$ with the singleton sequence $\emptyset$, we take the initial state to be a final state if-and-only-if $\emptyset \models \psi$. The additional accepting states are those that contain formula ψ.

Let $\hat{\delta}_\psi$ denote the canonical extension of function δ_ψ to strings $h \in \mathcal{C}_{ES}^*$.

Lemma 4.1 (Automaton Invariant). *Let $h \in \mathcal{C}_{ES}^+$ be any non-empty history, and ψ_j be any sub-formula of ψ. Then $\hat{\delta}_\psi(s_0, h) \neq s_0$ and furthermore, $\psi_j \in \hat{\delta}_\psi(s_0, h)$ if-and-only-if $h \models \psi_j$.*

Proof. Simple induction in h. □

Theorem 4.2. $\mathcal{L}(A_\psi) = \{h \in \mathcal{C}_{ES}^* \mid h \models \psi\}$

Proof. Immediate from Lemma 4.1 and the definition of s_0 and F. □

In the abstract setting of automaton A_ψ, we can now give a very simple and concise description of an alternative data structure DS' for implementing the interface for dynamic model checking, DMC. The basic idea is to pre-construct the automaton during initialization, and basically replacing the dynamic filling of the arrays $DS.B_j$ of DS with automaton-transitions.

Initialization. Just as with DS, the data structure DS' is initialized with an event structure ES and formula ψ. Initialization now simply consists of constructing the automaton A_ψ. More specifically, we construct the transition-matrix of δ_ψ so that $\delta_\psi(s, x)$ can be computed in time $O(1)$ by a matrix-lookup.[3] DS' maintains a variable $DS'.s_{\mathsf{summ}}$ of type S (the automaton states) which is initialized to s_0. In addition to s_{summ}, DS' will store a vector of pairs $DS'.h = [(y_1, s_1), (y_2, s_2), \ldots, (y_K, s_K)]$, where the y_i's are configurations representing active sessions, and the s_i's are corresponding automaton-states where s_i is the state that A_ψ is in after reading y_i. Initially this vector is empty.

Invariance. Let $h = x_1 x_2 \cdots x_M \cdot y_{M+1} \cdots y_{M+K}$ be the actual interaction history, i.e. $(x_i)_{i=1}^M$ is the longest prefix of maximal configurations. The data-structure invariant of DS' is that, if $DS'.h = [(y_1, s_1), (y_2, s_2), \ldots, (y_K, s_K)]$ then $(y_1, \ldots, y_K)$ are the active configurations of h, and s_i is the state of the automaton after reading the string $x_1 x_2 \cdots x_M \cdot y_1 \cdots y_i$, when started in state s_0. The invariant regarding the special variable $DS'.s_{\mathsf{summ}}$ is simply that $DS'.s_{\mathsf{summ}} = \hat{\delta}_\psi(s_0, x_1 x_2 \cdots x_M)$, i.e. $DS'.s_{\mathsf{summ}}$ "summarizes" the history up to time M with respect to formula ψ. Notice that the invariant is satisfied after initialization.

[3] We choose a transition-matrix representation of δ_ψ for simplicity. In practice, any representation allowing efficient computations of $\delta_\psi(s, x)$ could be used.

Operations. Let $DS'.h = [(y_1, s_1), (y_2, s_2), \dots, (y_K, s_K)]$. Then operation DMC.**check**() returns true iff $s_K \in F$. By the invariant and Lemma 4.1 this is equivalent to $h \models \psi$. For operation DMC.**new**(), extend $DS'.h$ with the pair $(\emptyset, \delta_\psi(s_K, \emptyset))$. Finally, for operation DMC.**update**(e, i), add e to configuration y_i of $DS'.h$, then update the table $DS'.h$ by starting the automaton in state s_{i-1} (or s_{summ} if $i = 1$), and setting $s_i := \delta_\psi(s_{i-1}, y_i)$, and then $s_{i+1} := \delta_\psi(s_i, y_{i+1})$, and so on until the entire table $DS'.h$ satisfies the invariant. If $i = 1$ and $y_1 \cup \{e\}$ is maximal, we must, as in DS, recompute the largest longest prefix, and we may deallocate the corresponding part of the table $DS'.h$ (taking care to update $DS'.s_{\mathsf{summ}}$ appropriately).

Since δ_ψ can be evaluated in time $O(1)$, we get the following theorem.

Theorem 4.3 (Automata-based *DMC*). *The automata-based data structure (DS') implements 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, the space requirement of DS' is $O(K + |E| \cdot |\mathcal{C}_{ES}| + 2^{|\psi|} \cdot |\mathcal{C}_{ES}|)$.

4.3 Remarks

The array- and automata-based implementations are very similar. The automata-based implementation simply precomputes a matrix of transitions $B \xrightarrow{x} B'$ instead of recomputing from scratch the array B' from B and x, every time it is needed. This reduces the complexity of operations DMC.**update**(e, i) and DMC.**new**() by a factor of $|\psi|$. The cost of this is in terms of storage and time for pre-computation, where, in the worst case, the transition matrix is exponential in ψ (of size $2^{|\psi|} \times |\mathcal{C}_{ES}|$). One important advantage with the automata-based implementation (besides being conceptually simpler) is that we can apply the standard technique for constructing the minimal finite automaton equivalent to A_ψ. We believe that, in practice, this minimization will give significant time and space reductions. Note that minimization can be run several times, and not just during initialization. In particular, one could run minimization each time state s_{summ} is updated in order to obtain optimizations, e.g. removing states that are unreachable in the future.

Recall, that one might be interested in different notions of "time" in our temporal logic. Consider the following. Redefine **update**(): for $h = x_1 \cdots x_n$, $1 \leq i \leq N$, $e \in E$, define

$$\mathbf{update}(h, e, i) = x_1 x_2 \cdots x_{i-1} x_{i+1} x_{i+2} \cdots x_N (x_i \cup \{e\})$$

This definition implements the idea that x_i precedes x_j in sequence h if x_j was updated more recently than x_i. Notice that our algorithms (as well as complexity analyses) can be easily adapted to this time concept: the update operation simply swaps the indexes of configurations i and N in the vector of configurations before updating the boolean arrays (or automata-states in case of the automata-based algorithm).

5 Language Extensions

In this section, we consider two extensions of the basic policy language to include more realistic and practical policies. The first is parameters and quantification. For example, consider the OOk 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. The first language extension allows this sort of quantification, and considers an accompanying notion of parameterized events.

The second language extension covers two aspects: quantitative properties and referencing. 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. For this reason, our first extension to the usual past-time temporal-logic is the ability to express also *quantitative* properties, e.g. "in at least 98% of the previous interactions, seller has not provided negative feedback in auctions where payment was made." The second extension is the ability, to not only refer to the locally observed behaviour, but also to require properties of the behaviour observed by others. As a simple example of this, suppose that b_1 and b_2 are two branches of the same network of banks. When a client c wants to obtain a loan in b_1, the policy of b_1 might require not only that c's history in b_1 satisfy some appropriate criteria, but also that c has always payed his mortgage on time in his previous loans with b_2. Thus we allow local policies, like that of b_1, to refer to the *global* behaviour of an entity.

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") or `open`("./tmp.txt")).

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 $\rho : E \to \mathcal{P}$ 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 $x : E \to \bigcup_{e\in E} \rho(e)$ 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 $(x/y)(e) = y(e)$. Finally we write $\emptyset$ for the totally undefined configuration (when the meaning is clear from the context).

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).

Definition 5.3 (Histories). A local (interaction) history h in a parameterized event structure ρES is a finite sequence $h \in {\mathcal{C}^0_{\rho ES}}^*$.

Definition 5.4 (Extended Interface). Overload operation **new** : ${\mathcal{C}^0_{\rho ES}}^* \to {\mathcal{C}^0_{\rho ES}}^*$ by $\mathbf{new}(h) = h\emptyset$. Overload also partial operation **update** : ${\mathcal{C}^0_{\rho ES}}^* \times E \times (\bigcup_{e\in E} \rho(e)) \times \mathbb{N} \to {\mathcal{C}^0_{\rho ES}}^*$ as follows. For any $h = x_1 x_2 \cdots x_i \cdots x_n \in {\mathcal{C}^0_{\rho ES}}^*$, $e \in E$, $p \in \bigcup_{e\in E} \rho(e)$, and $i \in \mathbb{N}$, $\mathbf{update}(h, e, p, i)$ is undefined if $i \notin \{1, 2, \ldots, n\}$, $dom(x_i) \overset{e}{\not\to} dom(x_i) \cup \{e\}$ or $p \notin \rho(e)$. Otherwise

$$\mathbf{update}(h, e, p, i) = x_1 x_2 \cdots ((e \mapsto p)/x_i) \cdots x_n$$

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-variables v, u range over $\mathit{Val} \overset{(def)}{=} \mathit{Var} \cup \bigcup_{i=1}^{\infty} P_i$, and metavariable p range over $\bigcup_{i=1}^{\infty} P_i$.

The quantified policy language is given by the following BNF.

$$\begin{array}{rcl} \psi & ::= & e(v) \mid \Diamond e(v) \mid \psi_0 \wedge \psi_1 \mid \neg\psi \mid \\ & & \mathsf{X}^{-1}\psi \mid \psi_0 \;\mathsf{S}\; \psi_1 \mid \forall x : P_i.\psi \end{array}$$

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 use the syntactic abbreviations of Section 3, and additionally the existential quantification $\exists x : P_i.\psi \equiv \neg\forall x : P_i.\neg\psi$.

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 $\sigma : \mathit{Val} \to \bigcup_{i=1}^{\infty} P_i$ 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 history, and $i \in \mathbb{N}$. We define a satisfaction relation $(h, i) \models^{\sigma} \psi$ by structural induction on ψ.

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

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 OOk 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:

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

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 $B_k[j]$, 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 $(h,k) \models^{\sigma} \psi_j$.

Constraints. Fix a quantified formula ψ and a history $h = x_1 x_2 \cdots x_n \in {\mathcal{C}^0_{\rho ES}}^*$. We assume for simplicity that all m variables of ψ, say $\mathit{vars}(\psi) = \{y_1, y_2, \ldots, y_m\}$, have the same type P (this restriction is inessential). Let $P_h \subset P$ denote the set of distinct parameter occurrences in h (i.e., $P_h = \{q \in P \mid \exists e \in E \exists i \leq |h|.e \in \mathit{dom}(x_i) \text{ and } x_i(e) = q\}$). For a finite set V of variables, let Σ_V denote the set of substitutions for the variables V, i.e., $\Sigma_V = V \rightarrow P$. Let us define an equivalence $\equiv_{P_h}$ on substitutions Σ_V, by

$$\sigma \equiv_{P_h} \sigma' \text{ iff } \forall x \in V. \begin{cases} \sigma(x) = \sigma'(x) & \text{if } \sigma(x) \in P_h \\ \sigma'(x) \notin P_h & \text{if } \sigma(x) \notin P_h \end{cases}$$

Let $\Sigma_V^{P_h} = \Sigma_V / \equiv_{P_h}$ be the set of equivalence classes for $\equiv_{P_h}$. Let $\star \notin P$ be arbitrary but fixed. Note that an equivalence class $[\sigma]$ can be uniquely represented as a function $s : V \to P_h \cup \{\star\}$, i.e., by $s(x) = \sigma(x)$ if $\sigma(x) \in P_h$ and $s(x) = \star$ otherwise. This is clearly independent of the class representative σ. For the rest of this paper we shall identify $\Sigma_V^{P_h}$ with $V \to P_h \cup \{\star\}$. The following lemma establishes that with respect to model checking, substitutions are only distinguished up to $\equiv_{P_h}$-equivalence.

Lemma 5.1. *For all quantified formulas ψ, all histories h, and all substitutions $\sigma, \sigma' \in \Sigma_{fv(\psi)}$*

$$\textit{if } \sigma \equiv_{P_h} \sigma' \textit{ then } h \models^{\sigma} \psi \iff h \models^{\sigma'} \psi$$

Proof. Let $h = x_1 \cdots x_n$ be fixed, and recall $P_h = \{q \in P \mid \exists e \in E \exists i \leq |h|.e \in dom(x_i) \text{ and } x_i(e) = q\}$. Let $\sigma \equiv_{P_h} \sigma'$. Our proof is by structural induction in ψ. For the base case we need only consider the atomic formulas of form $e(x)$ or $\Diamond e(x)$ (if ψ doesnt have a free variable then its truth is independent of the substitution). If $\sigma(x) \in P_h$ then since $\sigma \equiv_{P_h} \sigma'$, we have $\sigma'(x) = \sigma(x)$ and the result is obvious. If $\sigma(x) \notin P_h$ then since $\sigma \equiv_{P_h} \sigma'$, we also have $\sigma'(x) \notin P_h$. Hence $h \not\models^{\sigma} e(x)$ and $h \not\models^{\sigma'} e(x)$. If e is in conflict with x_n or $e \in x_n$ then $h \not\models^{\sigma} \Diamond e(x)$ and $h \not\models^{\sigma'} \Diamond e(x)$, otherwise $h \models^{\sigma} \Diamond e(x)$ and $h \models^{\sigma'} \Diamond e(x)$.

For the inductive step, all cases follow trivially from the inductive hypothesis. For example, for $\psi = \forall x : P_j.\psi$ then since $h \models^{\sigma} \psi \iff h \models^{\sigma'} \psi$, clearly $h \models^{\sigma} \forall x : P_j.\psi$ iff for all $p \in P_j.h \models^{(x \mapsto p)/\sigma} \psi_j$ iff $p \in P_j.h \models^{(x \mapsto p)/\sigma'} \psi_j$ (because for any fixed $p \in P_j$ we have $(x \mapsto p)/\sigma \equiv_{P_h} (x \mapsto p)/\sigma'$). □

A function $c : \Sigma_V^{P_h} \to \{\top, \bot\}$ is called a (V-) *constraint* (in h). A substitution $\sigma \in \Sigma_V$ satisfies constraint c if $c([\sigma]) = \top$. In this case we write $\sigma \models c$. We write $Constraint_V$ for the set of V-constraints (in some fixed history h which is clear from the context), and if $c : \Sigma_V^{P_h} \to \{\top, \bot\}$ is a constraint, then $vars(c) \overset{(\text{def})}{=} V$. Notice that when $fv(\psi) = \emptyset$ then $\Sigma_{fv(\psi)} \simeq \mathbf{1}$ (i.e., a singleton set), hence a constraint is simply a boolean. In this sense, constraints generalize booleans.

In the array-based algorithm, sub-formula ψ_j will be associated with a ψ_j-constraint $C_k[j]$ in h, i.e., on the free variables of ψ_j (where C_k will correspond to time k in a history h). Notice that replacing the boolean arrays $B_k[j]$ with constraint arrays $C_k[j]$ can be seen as a proper generalization of the array-based algorithm. We generalize the (main) invariant of the algorithm from

$$h, k \models \psi_j \iff B_k[j] = \texttt{true}$$

to

$$\forall \sigma \in \Sigma_{fv(\psi_j)}.\big[h, k \models^{\sigma} \psi_j \iff \sigma \models C_k[j]\big]$$

Notice that for closed ψ_j, the invariants are equivalent. It is also important to notice that constraints can be viewed as functions taking as input an m'ary vector of $(P_h \cup \{\star\})$-values (where m is the number of variables) and giving a boolean value as output. Hence constraints are finite objects. Notice also that since constraints are boolean valued, it makes sense to consider logical operators on constraints, e.g., the conjunction $(c \wedge c')([\sigma]) = c([\sigma]) \wedge c'([\sigma])$ of two constraints c and c' (even if they are not on the same variables).[4] For a variable x and a parameter $p \in P_h$ we will use notation $x \in \{p\}$ to denote the constraint given by $(x \in \{p\})([\sigma]) = \top \iff \sigma(x) = p$. Further $\top$ and $\bot$ denote respectively the two constant constraints.

Constructing constraints. Let $h = x_1 \cdots x_n$ be a history and $1 < k \leq n$. Define a translation $[\![\cdot]\!]_h^k$ from the quantified language to constraints, associating with each formula in the quantified language ψ, a constraint $[\![\psi]\!]_h^k$ on the free variables of ψ. The function $[\![\cdot]\!]_h^k$ is defined relative to index k and history h, and we assume (inductively) that when defining $[\![\psi]\!]_h^k$, we have access to $[\![\psi']\!]_h^k$ for all proper sub-formulas ψ' of ψ, and also $[\![\psi']\!]_h^{k-1}$ for all sub-formulas ψ' of ψ. In the model-checking algorithm, the constraint $[\![\psi_j]\!]_h^k$ will correspond to entry j in array C_k. Recall that the invariant we aim to maintain is the following.

$$\forall \sigma \in \Sigma_{fv(\psi_j)}.\big[h, k \models^{\sigma} \psi_j \iff \sigma \models C_k[j]\big]$$

We define function $[\![\cdot]\!]_h^k$ as follows.

$$[\![e(v)]\!]_h^k \quad = \quad \begin{cases} y \in \{p\} & \text{if } v = y \text{ and } e \in \mathit{dom}(x_k) \text{ and} \\ & \quad x_k(e) = p \\ \top & \text{if } v = p \text{ and } e \in \mathit{dom}(x_k) \text{ and} \\ & \quad x_k(e) = p \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\Diamond e(v)]\!]_h^k \quad = \quad \begin{cases} y \in \{p\} & \text{if } v = y \text{ and } e \in \mathit{dom}(x_k) \text{ and} \\ & \quad x_k(e) = p \\ \top & \text{if } (v = p \text{ and } e \in \mathit{dom}(x_k) \text{ and} \\ & \quad x_k(e) = p) \text{ or if} \\ & e \notin \mathit{dom}(x_k) \text{ and } e \ \# \ \mathit{dom}(x_k) \\ \bot & \text{otherwise} \end{cases}$$

We proceed inductively in ψ.

$$\begin{array}{lcl} [\![\psi_0 \wedge \psi_1]\!]_h^k & = & [\![\psi_0]\!]_h^k \wedge [\![\psi_1]\!]_h^k \\ [\![\neg\psi]\!]_h^k & = & \neg[\![\psi]\!]_h^k \\ [\![\mathsf{X}^{-1}\psi]\!]_h^k & = & [\![\psi]\!]_h^{k-1} \\ [\![\psi_0 \ \mathsf{S} \ \psi_1]\!]_h^k & = & [\![\psi_1]\!]_h^k \vee ([\![\psi_0 \ \mathsf{S} \ \psi_1]\!]_h^{k-1} \wedge [\![\psi_0]\!]_h^k) \\ [\![\forall x : P.\psi]\!]_h^k & = & \mathit{elim}_x([\![\psi]\!]_h^k) \end{array}$$

[4] If $A \subseteq B$ then an A-constraint can be seen as a B-constraint by imposing no additional requirements on the extra variables.

All the clauses are straightforward except for $\forall x : P.\psi$, which is handled by auxiliary function $elim_x$. We define this function now. Assuming we have access to $c = [\![\psi]\!]_h^k$ so that $\sigma \models c \iff (h,k) \models^\sigma \psi$, we must produce a new constraint c' of type $Constraint_{fv(\psi)\setminus\{x\}}$, so that

$$\sigma \models c' \iff \big[\forall p \in P.((x \mapsto p)/\sigma) \models c\big] \qquad \text{(for all } \sigma\text{)}$$

The function $elim_x$ does this; it transforms a constraint c into a constraint $c' = elim_x(c)$ with $vars(c') = vars(c) \setminus \{x\}$, satisfying the above equivalence. Since P_h is finite we can build c' as one large conjunction: for all $\sigma \in \Sigma_{fv(\psi)\setminus\{x\}}$

$$c'([\sigma]) = (\bigwedge_{q \in P_h} c([(x \mapsto q)/\sigma])) \wedge c((x \mapsto \star)/[\sigma])$$

Notice that we would obtain a function for existential quantification by taking a disjunction instead of a conjunction.

Array-based Model Checking. In the light of function $[\![\cdot]\!]$ there is a straightforward extension of data-structure DS into a similar data-structure $DS^\forall$ for array-based dynamic model-checking of the quantified language. Structure $DS^\forall$ will maintain a history $DS^\forall.h = x_1 x_2 \cdots x_n$, and a collection of $n+1$ constraint-arrays $DS^\forall.C_k[j]$ (for $0 \leq k \leq n$), each array indexed by the subformulas of ψ. The constraint in $C_k[j]$ will be $C_k[j] = [\![\psi_j]\!]_k^h$ for $k > 0$ (C_0 is the special summary constraint). The invariant implies that for any closed ψ,

$$(h,n) \models \psi \iff \models DS^\forall.C_n[0]$$

(we write $\models c$, and say that c is *valid*, if $c = \top$). Hence operation **check** is a validity check, which is easy since $vars(DS^\forall.C_n[0]) = \emptyset$ when ψ is closed. Operation **new** is essentially as in DS (with the generalization from booleans to constraints).

For operation **update**(e,p,i) there are two cases. In the first case $p \in P_h$, and update works as usual (again generalizing to constraints). In the case where $p \notin P_h$, we update history h to h' appropriately, and thus obtain a new, larger $P_{h'} = P_h \cup \{p\}$. Notice that constraints in h can be easily extended to constraints in h': if $c : \Sigma_V^{P_h} \to \{\top, \bot\}$ then we can think of c as a constraint in h' by the following. For all $\sigma \in \Sigma_V$, let $[\sigma]_{h'}$ be the $\equiv_{P_{h'}}$-equivalence class for σ, and let $[\sigma]_h$ be the $\equiv_{P_h}$-equivalence class for σ, then

$$c([\sigma]_{P_h'}) = c([\sigma]_{P_h})$$

This means that we can use the logical operators on constraints c in the history h and constraints c' in the history h', by first extending c to a constraint in h', and then performing the logical operation. Hence **update**(e,p,i) can be implemented as usual, except that we may need to dynamically extend some constraints in h to constraints in h'.

Complexity. The above paragraphs show that dynamic model-checking for the quantified language is decidable in spite of the fact that we allow quantification over infinite parameter sets. This is essentially due to the fact that in any history, only a finite portion of the parameters can actually occur. 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.

Proof. Fix a parameterized event structure ES. A quantified model-checking (QMC) instance (for ES) consists of a history $h = x_1 \cdots x_n$ and a closed formula ψ of the quantified language (over ES). Say that a QMC instance (h, ψ) is in QMC if $h \models \psi$. A single element model is a model, $h \in {\mathcal{C}^0_{\rho ES}}^*$, with $h = x$, where $x \in \mathcal{C}^0_{\rho ES}$.

The quantified boolean formula (QBF) problem is the problem of deciding the truth of quantified formulas of the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n . \phi(x_1, \ldots, x_n)$$

where each Q_i is a quantifier ($\forall$ or $\exists$), and ϕ is a quantifier-free boolean formula (i.e., a propositional formula) with $\mathit{fv}(\phi) \subseteq \{x_1, \ldots, x_n\}$. The QBF problem is known to be *PSPACE* complete [34]. Given a QBF $f = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n . \phi$, construct an MC-instance as follows. Use a parameterized event structure with a single event $\star$ having two possible parameters $\bot$ and $\top$. Let $h = [\star \mapsto \top]$ be a single element history. Construct formula ψ as $\psi \equiv Q_1 x_1 Q_2 x_2 \cdots Q_n x_n . \psi'$, where $x_1, \ldots, x_n$ are the variables of f, and ψ' is ϕ with each variable x_j replaced by $\star(x_j)$. Then f is satisfiable if-and-only-if $(h, |h|) \models \psi$. □

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. Clearly the complexity depends on the representation of constraints $c : \Sigma_V^{P_h} \rightarrow \{\top, \bot\}$. One efficient representation of constraints is using multiple-valued decision diagrams [16]. With this representation, constraints c can be efficiently stored in space $O((|P_h| + 1)^n)$ and the logical operations can all be computed in linear time $O((|P_h| + 1)^n)$. Further a constraint in h can be extended to a constraint in $h' = \mathbf{update}(h, e, p, i)$ in linear time $O((|P_{h'}| + 1)^n)$.

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 $\mathit{fv}(\psi') \subseteq \{x_1, \ldots, x_n\}$. 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.

- $\mathit{DMC}.\mathbf{check}()$ is $O(1)$.
- $\mathit{DMC}.\mathbf{new}()$ is $O(|\psi| \cdot (|P_h| + 1)^n)$.

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

- $DMC.\mathbf{update}(e, p, i)$ 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, the space requirement of DS' is $O(K \cdot (|E| + |\psi| \cdot (|P_h| + 1)^n))$.

5.2 References and Quantitative Properties

In this section, we briefly illustrate another way to extend the core policy-language to a more practical one. As mentioned, we consider two aspects: referencing and quantitative properties. For referencing we introduce a construct $p : \psi$, where p is a principal-identity and ψ is a basic policy. The construct is intended to mean that principal p's observations (about a subject) must satisfy past-time ψ. For quantitative properties, we introduce a counting operator $\overline{\#}$, used e.g. in formula $p : \overline{\#}\psi$ which counts the number of p-observed sessions satisfying ψ (we use $\overline{\#}$ to avoid confusion with the conflict relation, often denoted by $\#$).

To express referencing, we extend the basic syntax to include a new syntactic category π (for policy). Let *Prin* be a collection of principal identities.

$$\pi \quad ::= \quad p : \psi \mid \pi_0 \wedge \pi_1 \mid \neg\pi \qquad p \in \mathit{Prin}$$

The policy $p : \psi$ means that the observations that p has made should satisfy ψ. Note that in this extended language, models are no longer local interaction histories, but, instead, global interaction histories, represented as a principal-indexed collection of local histories (i.e., functions of type $\mathit{Prin} \to \mathcal{C}_{ES}^*$).

The quantitative extension is given by extending the category ψ. Let $(\mathcal{R}_j)_{j=1}^{\infty}$ be a countable collection of k'ary relation-symbols for each $k \in \mathbb{N}$, representing computable relations $[\![\mathcal{R}_j]\!] \subseteq \mathbb{N}^k$.

$$\psi \quad ::= \quad \ldots \mid \mathcal{R}_j(\overline{\#}\psi_1, \overline{\#}\psi_2, \ldots, \overline{\#}\psi_k)$$

The denotation of the construct $\overline{\#}\psi$ is the number of sessions in the past which satisfy formula ψ, e.g., $\overline{\#}$`negative` counts the number of states in the past satisfying `negative`. So the denotation of $\overline{\#}\psi$ is a number, and the semantics of $\mathcal{R}_j(\overline{\#}\psi_1, \overline{\#}\psi_2, \ldots, \overline{\#}\psi_k)$ is `true` iff $(n_1, n_2, \ldots, n_k) \in [\![\mathcal{R}_j]\!]$, where n_i is the denotation of $\overline{\#}\psi_i$. Finally, we extend also category π:

$$\pi \quad ::= \quad \ldots \mid \mathcal{R}_j(p_1 : \overline{\#}\psi_1, \ldots, p_k : \overline{\#}\psi_k) \qquad p_i \in \mathit{Prin}$$

The construct $\mathcal{R}_j(p_1 : \overline{\#}\psi_1, \ldots, p_k : \overline{\#}\psi_k)$ means that, letting n_i denote the number of sessions observed by principal p_i satisfying ψ_i, then the relation $[\![\mathcal{R}_j]\!]$ on numbers must have $(n_1, \ldots, n_k) \in [\![\mathcal{R}_j]\!]$.

We do not provide a formal semantics as the meaning of our constructs should be intuitively clear, and our purpose is simply to illustrate how the core language can be extended to encompass more realistic policies. To further illustrate the constructs, we consider a number of example policies. In the following examples, $p, p_1, p_2, \ldots, p_n \in \mathit{Prin}$ are principal identities.

Example 5.2 (eBay revisited). Consider the eBay scenario again. The policy of Example 3.1 could be extended with referencing, e.g. principal p might use policy:

$$\begin{aligned} \pi_p^{\mathsf{bid}} \equiv \quad & p : \mathsf{G}^{-1}(\mathtt{negative} \rightarrow \mathtt{ignore}) \quad \wedge \\ & \textstyle\bigwedge_{q \in \{p, p_1, \ldots, p_n\}} q : \neg \mathsf{F}^{-1}(\mathtt{time\text{-}out}) \end{aligned}$$

Intuitively, this policy represents a requirement by principal p: "seller has never provided negative feedback about me, regarding auctions where I made payment, and, furthermore, seller has never cheated me or any of my friends."

Example 5.3 (P2P File-sharing). This example is inspired by the example used in the license-based system of Shmatikov and Talcott [31]. Consider a scenario where a P2P file-servent has two resources, dl (download), and ul (upload). Suppose this is modelled by an event structure with two independent events `dl` and `ul`, so that in each session, a peer-client either uploads, downloads or both. We express a policy used by server p for granting download, stating that "the number of uploads should be at least a third of the number of downloads."

$$\pi_p^{\mathsf{client\text{-}dl}} \equiv p : (\overline{\#}\mathtt{dl} \leq 3 \cdot \overline{\#}\mathtt{ul})$$

This refers only to the local history with p. Supposing we instead want to express a more "global" policy on the behaviour, stating that globally, p has uploaded at least a third of its downloads (e.g. locally this may be violated).

$$\begin{aligned} \pi_p^{\mathsf{client\text{-}dl}} \equiv \quad & (p : \overline{\#}\mathtt{dl}) + (\textstyle\sum_{i=1}^{n} p_i : \overline{\#}\mathtt{dl}) \quad \leq \\ & 3 \cdot (p : \overline{\#}\mathtt{ul} + (\textstyle\sum_{i=1}^{n} p_i : \overline{\#}\mathtt{ul})) \end{aligned}$$

Example 5.4 ("Probabilistic" policy). Consider an arbitrary event structure $ES = (E, \leq, \#)$. We express a policy ensuring that "statistically, event $\mathtt{ev} \in E$ occurs with frequency at least 75%."

$$\pi_p^{\mathsf{probab}} \equiv p : \frac{\overline{\#}\mathtt{ev}}{\overline{\#}\mathtt{ev} + \overline{\#}{\sim}\mathtt{ev} + 1} \geq \frac{3}{4}$$

Here $\overline{\#}\,{\sim}\mathtt{ev}$ counts the number of sessions in which `ev` has not occurred and cannot occur in the future.

5.2.1 Implementation remarks.

Dynamic model checking for the extended policy language can done by extending the array-based algorithm from the previous section. Note that the value of $\overline{\#}\psi$ can easily be defined in the style of the recursive semantics. To handle the construct $\mathcal{R}(\overline{\#}\psi)$, one maintains a number of integer variables which denote the values of sub-formula $\overline{\#}\psi$ at each active session. The integers are then updated using the recursive semantics in a way similar to the array-updates in Section 4. We have the following result, assuming that the relations can be evaluated in constant time, and that numbers can be stored/manipulated in constant space/time.

Theorem 5.2. Let formula ψ be from the basic language extended with the quantitative constructs. Let $h \in {\mathcal{C}^0_{ES}}^*$ be a history. The dynamic model checking can be implemented with the following complexity.

- $DMC.$**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 configurations in h.

Note, that the automata-based algorithm does not easily extend: the (semantics of the) extended language is no-longer regular, e.g. illustrated by formula $\psi_p \equiv p : (\overline{\#}\texttt{dl} \leq \overline{\#}\texttt{ul})$.

The construct $p : \psi$, where p is a principal identity, requires that p's interaction history (with the subject in question) satisfies ψ. This is handled simply by "sending formula ψ" to p. Principal p maintains the truth of ψ with respect to its interaction history using the algorithms of last section, and sends the required value to the requesting principal when needed.[5] Another approach is for p to send its entire interaction history so that the verification can be performed locally, e.g., as is done with method **exportEvents** in the license-based framework of Shmatikov and Talcott [31]. It does not make sense to consider the algorithmic complexity of referencing. The message complexity of referencing, however, is linear in the number of principals to be contacted (one query and one reply).

6 A Java Security Manager

In this section we describe an application of our logical framework to the area of history-based access control for untrusted code. We have designed and implemented a prototype Java Security Manager which is able to monitor a Java program with respect to a "history-based" policy, written in our logic. If the security manager detects a violation of policy, a Java security exception is thrown and the violating action is aborted. We describe briefly the Java security model,

[5] One might argue that this leads to problems of timing: at what point in time is ψ then to be evaluated? But such timing-issues are inherent in distributed systems. Formula $p : \psi$ is a relative temporal specification that is interpreted by the sender as referring to the current history of p, *when p decides to evaluate it.* The sender of ψ thus knows that received valuation (true or false) reflects an evaluation of ψ with respect to some *recent view* of p's history.

and proceed with a more detail description of the design and implementation of our history-based security manager.

The Java Programming Language supports the concept of a *security manager*: an object that supervises another Java application with respect to security sensitive operations, e.g., file or network access.[6] Java programs that run other Java programs, e.g., a browser running a Java applet, can install a security manager that mediates the untrusted program's security sensitive operations. Operations, like connecting to a socket on a remote site, are performed by Java applications via the Java API, e.g., class `Socket` of the `java.net` library provides an appropriate abstraction for "sockets." API classes make calls to the security manager's `checkPermission(java.lang.Permission)` method whenever a security sensitive operation is requested, e.g., the `Socket` class calls `checkPermssion` with an appropriate instance of the `java.net.SocketPermission` (containing information about which remote site and port is being accessed). The security manager then inspects the `java.lang.Permission` object (possibly consulting a user-specified security policy) and throws a `java.lang.SecurityException` if access should not be granted.

The Java security architecture allows users to write their own security managers by extending the `java.lang.SecurityManager` class. We have written a security manager that decides access by checking conformance to policies from our history-based framework. The application is a simple prototype, used only for testing the validity of our approach to history-based access control, and consequently, the current version supports only a small subset of the security relevant operations available in Java.[7]

6.1 Design

We have designed two versions of the HBAC application, a basic and a parameterized version, corresponding to the basic and parameterized languages described previously. The basic events in our event structure correspond to the Java security events, e.g., `java.io.FilePermission` for representing file-access. For simplicity, the current version supports only the events corresponding to file and network access, corresponding to the java classes `java.io.FilePermission` and `java.net.SocketPermission`, however it would be simple to extend this to all the security relevant events. The basic event structure thus consists of conflicting events `FilePermission(read)`, `FilePermission(write)`, `FilePermission(delete)`, `FilePermission(execute)`; and `SocketPermission(connect)`, `SocketPermission(listen)`, `SocketPermission(accept)`, `SocketPermission(resolve)`. Note that since there are only four types of operations for each event-type (e.g. 'read' for the 'FilePermission') these "finitely parameterized" events can be represented in the basic model. In the parameterized model, the parameterized events include also information about filenames/hostnames, e.g., event `Socket-`

[6] More information about the Java security architecture, and security managers can be found at `http://java.sun.com/security/index.jsp`.

[7] The prototype source code is available as an open-source project, hosted at SourceForge, `https://sourceforge.net/projects/javahbac`.

Permission(connect) has further string-type parameters specifying a port and hostname, and FilePermission(read) has a parameter specifying the filename.

We have provided DSD2.0 [18, 24] descriptions of XML languages for both the basic and parameterized policies. A policy consists of a list of actions, e.g., java.net.SocketPermission(connect), followed by a formula from one of the two logics. An example policy is provided in Figure 7; it describes the policy requiring that for the application to perform the actions of connecting a socket or accepting a socket connection, the history must satisfy the property

$$\mathsf{G}^{-1}(\forall x : \mathit{String}.(\texttt{java.lang.FilePermission}(\texttt{read})(x) \quad \rightarrow \\ \mathsf{F}^{-1}\texttt{java.lang.FilePermission}(\texttt{write})(x)))$$

We have implemented a SAX parser which reads a policy file from the disk and generates an internal data-structure representing the policy. This parser can be used by an application that wishes to install a security manager implementing a policy.

We have defined two security managers: an automata-based security manager (SecMan.java) for the basic language, using the efficient Java package dk.brics.automaton [23]; and an array-based security manager for the quantified language (QSecMan.java), using the JavaBDD binary decision diagram package for implementing constraints [35]. The input for both security managers is an XML representation of a policy, and both override the method checkPermission of the SecurityManager class to check whether a specific action is allowed. The basic security manager uses the automata-based algorithm from Section 4, whereas the quantified security manager uses the array-based algorithm from the same section, but extending the booleans to the constraints of Section 5.

We illustrate, by means of example, how one might use our security managers. Consider the following Java program which tries to read the file ”secret.txt” and then send the contents to a location on the Internet.

```
import java.io.*;
import java.net.*;
public class Evil {
    public static void main(String[] args) throws Exception {
        System.out.println("begin");
        BufferedReader buf = new BufferedReader(new FileReader("secret.txt"));
        String line = null;
        StringBuffer sbuf = new StringBuffer();
        System.out.println("reading password");
        while ((line = buf.readLine())!=null) {
            System.out.println(line);
            sbuf.append(line);
        }
        System.out.println("opening connection");
        Socket s = new Socket("www.microsoft.com",80);
        Writer out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));
        out.write(sbuf.toString(),0,sbuf.toString().length());

        System.out.println("done!");
    }
}
```

Suppose we want to run this program under the above example security policy. An application class for installing a security manager and running the program could be the following.

```
import java.util.*;
import java.security.Permission;

public class TestQSecMan {
    public static void main(String[] args) throws Exception {
        QPolicyParser pp = new QPolicyParser();
        pp.parse(args[0]);

        QSecMan sec = setupSecurityManager(pp);
        System.out.println("Setting Security Manager");
        System.setSecurityManager(sec);
        System.out.println("Starting Program");
        Evil.main(null);
    }
    private static QSecMan setupSecurityManager(QPolicyParser pp) {
        // initialize security manager
        ...
    }
}
```

First, the policy is specified as an argument to the program. The program parses the policy using our SAX parser. Then it constructs a quantified security manager object, using the setup-method (the specifics are simple and not relevant here). Once the security manager is constructed, it uses the `System.setSecurityManager` Java method, to install the monitor. After this, the program can be run, and the security manager ensures that the policy is not violated.

Specifically, the output of running the above looks as follows.

```
[krukow@smeagol:...HBAC-Security/test]$ java TestQSecMan file-quant-example.xml
Setting Security Manager
...
Starting Program
...
check: (java.util.PropertyPermission user.dir read)
check: (java.io.FilePermission secret.txt read)
reading password
thesecretpassWord
opening connection
check: (java.util.PropertyPermission java.net.preferIPv6Addresses read)
check: (java.lang.RuntimePermission loadLibrary.net)
check: (java.io.FilePermission /home/java/Linux-jdk1.5.0_04/jre/lib/i386/libnet.so read)
check: (java.util.PropertyPermission java.net.preferIPv4Stack read)
check: (java.util.PropertyPermission impl.prefix read)
check: (java.lang.reflect.ReflectPermission suppressAccessChecks)
check: (java.util.PropertyPermission sun.net.spi.nameservice.provider.1 read)
check: (java.net.SocketPermission www.microsoft.com resolve)
Exception in thread "main" java.lang.SecurityException: Execution History Exception: Neg(QSince(QTrue, Neg(QF
orall x.(Neg(Conj(Event((java.io.FilePermission x read)(x)), Neg(QSince(QTrue, Event((java.io.FilePermission
x write)(x))))))))))
        at QSecMan.checkPermission(QSecMan.java:37)
        at java.lang.SecurityManager.checkConnect(SecurityManager.java:1031)
        at java.net.InetAddress.getAllByName0(InetAddress.java:1117)
        at java.net.InetAddress.getAllByName0(InetAddress.java:1098)
        at java.net.InetAddress.getAllByName(InetAddress.java:1061)
        at java.net.InetAddress.getByName(InetAddress.java:958)
        at java.net.InetSocketAddress.<init>(InetSocketAddress.java:124)
        at java.net.Socket.<init>(Socket.java:178)
        at Evil.main(Evil.java:15)
        at TestQSecMan.main(TestQSecMan.java:16)
```

We see that a security exception is thrown, not when the programs accesses the password, but when it tries to open a socket connection. We see also that there are a number of additional operations that are necessary for opening sockets, e.g., `(java.io.FilePermission,/home/java/Linux-jdk1.5.0_04/jre/lib/i386/libnet.so,read)`.

We have not yet done further experimentation with the framework, but our initial impression is good. Finally, we would like to compare our proposed framework to the similar system *Deeds*, of Edjlali et al. [9]. The Deeds system is similar to our prototype system in that Deeds also seeks to do history-based access control for Java (infact, *Deeds* was the main source of inspiration for this application). First, Deeds is more general than our system because "the set of security events is not fixed." In our system, the set of security-relevant events is restricted to what Java considers security events (this may change with future releases of Java). Secondly, Deeds is more *low-level* than our system: in Deeds, the programmer explicitly must maintain the event history (performing optimizations as he sees fit), and the programmer explicitly programs the security monitor (using full Java). This has the advantage that it is more flexible, but the disadvantage that such programming is error-prone, and highly security sensitive. In contrast, specifying an XML policy which is automatically monitored is less error-prone as the policy is declarative, and domain-specific. Furthermore, we're using standard algorithms that can efficiently handle all policies in the XML language, and which performs optimizations automatically, e.g., event history maintainance (and deallocation) and automata minimization. Finally, Deeds is much more fully developed while our approach is still at the prototype and evalutation level. We encourage interested readers to download the source code at `https://sourceforge.net/projects/javahbac`, and develop it further.

7 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.

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 [17], Shmatikov and Talcott propose a license-based framework [31], and the EU project 'SECURE' [3, 4] (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" [31]). 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 [27]. 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 [29] 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 [30], 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 [1,9,11,12,30,33]). 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, 30]; and *static* HBAC in which some preliminary static analysis of the program (written in a predetermined 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, 33]. 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.

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, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*, pages 316–332. Springer, 2005.

[2] 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.

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

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

[5] 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.

[6] C. Dellarocas. The digitization of word of mouth: Promise and challenges of online feedback mechanisms. *Management Science*, 49(10):1407–1424, October 2003.

[7] C. Dellarocas. Sanctioning reputation mechanisms in online trading environments with moral hazard. Working paper. Available online: `http://ccs.mit.edu/dell`, July 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] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Multivalued decision diagrams: Theory and applications. *International Journal on Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[17] 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.

[18] N. Klarlund, A. Møller, and M. I. Schwartzbach. The DSD schema language. *Automated Software Engineering*, 9(3):285–319, August 2002.

[19] P. Kollock. The production of trust in online markets. *Advances in Group Processes*, 16, 1999.

[20] D. Kreps, R. Milgrom, J. Roberts, and R. Wilson. Reputation and imperfect information. *Journal of Economic Theory*, 27(2):253–279, August 1982.

[21] 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.

[22] L. Mui, M. Mohtashemi, and A. Halberstadt. Notions of reputation in multi-agents systems: a review. In *Proceedings from The First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS'02)*, pages 280–287. ACM Press, 2002.

[23] A. Møller. The `dk.brics.automaton` project website. `http://www.brics.dk/automaton/`, 2005.

[24] A. Møller. The DSD2.0 project website. `http://www.brics.dk/DSD/`, 2005.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

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

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

[33] 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.

[34] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time(preliminary report). In *Proceedings of the fifth annual ACM symposium on Theory of computing (STOC'73)*, pages 1–9, New York, NY, USA, 1973. ACM Press.

[35] J. Whaley. The `javabdd` project website. `http://javabdd.sourceforge.net/`, 2005.

[36] R. Wilson. Reputations in games and markets. In A. Roth, editor, *Game-Theoretic Models of Bargaining*, pages 27–62. Cambridge University Press, 1985.

[37] G. Winskel. Event structure semantics for CCS and related languages. In *Proceedings from ICALP 1982*, volume 140 of *Lecture Notes in Computer Science*, pages 561–576. Springer Verlag, 1982.

[38] 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.

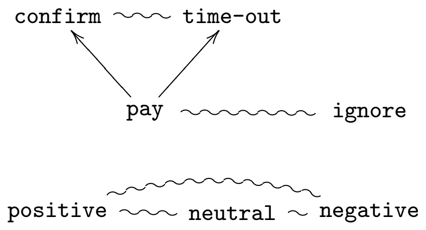

Figure 1: An event structure modelling the buyer's observations in the eBay scenario. (Immediate) Conflict is represented by $\sim$, and dependency by $\to$.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?dsd href="http://www.brics.dk/~krukow/dsd/quantifiedjavapolicies.dsd"?>

<policy  xmlns:xi="http://www.w3.org/2001/XInclude"
         xmlns="http://www.brics.dk/~krukow/dsd/quantifiedjavapolicies">

    <actions>
     <java.net.SocketPermission host="*">
       connect
     </java.net.SocketPermission>
     <java.net.SocketPermission host="*">
       accept
     </java.net.SocketPermission>
    </actions>
    <behaviour>
      <always>
       <forall var="x">
        <implication>
         <premise>
          <event>
           <java.io.FilePermission path="x">
            read
           </java.io.FilePermission>
          </event>
         </premise>
         <conclusion>
          <sometime>
            <event>
              <java.io.FilePermission path="x">
                write
              </java.io.FilePermission>
            </event>
          </sometime>
         </conclusion>
      </implication>
     </forall>
    </always>
  </behaviour>
</policy>
```

Figure 2: Example xml quantified HBAC policy.