

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

Karl Krukow<sup>\* †</sup>  
BRICS  
University of Aarhus, Denmark  
krukow@brics.dk

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

Vladimiro Sassone<sup>‡</sup>  
Department of Informatics  
University of Sussex, UK  
vs@sussex.ac.uk

## ABSTRACT

In a reputation-based trust-management system, agents maintain information about the past behaviour of other agents. This information is used to guide future trust-based decisions about interaction. However, while trust management is a component in security decision-making, few existing reputation-based trust-management systems aim to provide any formal security-guarantees. We provide a mathematical framework for a class of simple reputation-based systems. In these systems, decisions about interaction are taken based on policies that are exact requirements on agents' past histories. We present a basic declarative language, based on pure-past linear temporal logic, intended for writing simple policies. While the basic language is reasonably expressive, we extend it to encompass more practical policies, including several known from the literature. A naturally occurring problem becomes how to efficiently re-evaluate a policy when new behavioural information is available. Efficient algorithms for the basic language are presented and analyzed, and we outline algorithms for the extended languages as well.

## 1. INTRODUCTION

In global-scale distributed systems, traditional authorization mechanisms easily become either overly restrictive, or very complex [2]. In part, this is due to the vast numbers of principals they must encompass, and the open nature of the sys-

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

<sup>†</sup>BRICS: Basic Research in Computer Science ([www.brics.dk](http://www.brics.dk)), funded by the Danish National Research Foundation.

<sup>‡</sup>Supported by MyThS: Models and Types for Security in Mobile Distributed Systems, EU FET-GC IST-2001-32617.

tems. In *dynamic* and *reputation-based* trust-management systems, the problems of scale and openness are countered by taking a less static approach to authorization and, more generally, decision making. In these systems, principals keep track of the history of interactions with other principals. The recorded behavioural information is used to guide future decisions about interaction (see references [15, 21, 27, 30] on reputation, and [18] on dynamic trust management). This dynamic approach is being investigated as a means of overcoming the above mentioned *security* problems of global-scale systems. Yet, in contrast with traditional (cryptographic) security research, within the area of dynamic trust and reputation, no widely accepted security-models exist, and, consequently, few systems provide provable security guarantees (see, however, references [6, 19, 23] on general formal modelling of trust).

Many reputation systems have been proposed in the literature, but in most of these, the recorded behavioural information is heavily abstracted. For example, in the EigenTrust system [16], behavioural information is obtained by counting the number of 'satisfactory' and 'unsatisfactory' interactions with a principal. Besides lacking a precise semantics, this information has abstracted away any notion of time, and is further reduced (by normalization) to a number in the interval  $[0, 1]$ . In the Beta reputation system [14], similar abstractions are performed, obtaining a numerical value in  $[-1, 1]$  (with a statistical interpretation). It is not hard to find other examples of such information-abstraction in the reputation-system literature [15], and the only non-example we are aware of is the framework of Shmatikov and Talcott [30] which we discuss further in the concluding section.

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

In this paper, the concept of 'reputation system' is to be understood very broadly, simply meaning *any system in which principals record and use information about past behaviour of principals, when assessing the risk of future interaction*. We present a formal framework for a class of simple repu-

tation 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 verify precise formal 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 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  $\psi_r$ .”

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:

$$\psi^{browser} \equiv \neg F^{-1}(\text{modify}) \wedge \neg F^{-1}(\text{create-subprocess}) \wedge G^{-1}(\forall x. [\text{open}(x) \rightarrow F^{-1}(\text{create}(x))])$$

Informally, the atoms `modify`, `create-subprocess`, `open(x)` and `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, respectively, conjunction and negation. Thus, clauses  $\neg F^{-1}(\text{modify})$  and  $\neg F^{-1}(\text{create-subprocess})$  respectively require that the application has never modified a file, and has never created a sub-process. The quantified clause  $G^{-1}(\forall x. [\text{open}(x) \rightarrow F^{-1}(\text{create}(x))])$  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 associated with 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 Technical Contributions and Outline

We present a formal model of the behavioural information that principals obtain in our class of reputation systems. This model is based on previous work using event structures for modelling observations [24], 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) [25]. Indeed, our basic policy language is based on a (pure-past) variant of LTL. We give the formal syntax and semantics of our language, and provide several examples illustrating its naturality and expressiveness. We are able to encode several existing approaches to history-based access control, e.g. the Chinese Wall security policy [3] and a restricted version of so-called ‘one-out-of- $k$ ’ access control [9]. The formal description of our language, as well as the examples and encodings, is presented in Section 3.

We proceed by showing how one can effectively check whether an interaction history satisfies a policy. An interesting new problem is how to efficiently (dynamically) re-evaluate policies 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]. We present also an alternative algorithm using finite automata to obtain a lower dynamic complexity by doing a pre-computation on the policy. Interestingly, although one is verifying properties of an *entire* interaction history, one needs not store this complete history in order to verify a policy: old interaction can be efficiently summarized relative to the policy. Descriptions of both algorithms, and analysis of their time- and space-requirements is given in Section 4.

Finally, we illustrate how 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, 32], and within the traditional domain of reputation systems: peer-to-peer- and online feedback-systems [16, 27]). 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. 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.* [21], 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.

Related and future 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* [7]. Here we are not interested in modelling interaction protocols in such detail, but merely assume some system responsible for generating events.

We will use the event-structure framework of Nielsen and Krukow [24] 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  $i$ th 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 [24], we use an example complementing its formal definitions. We will use a scenario inspired by the eBay online auction-house [8] as our example.

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

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

The basis of the event-structure framework is the fact that the observations about protocol runs, such as an eBay transaction, have *structure*. Observations may be in *conflict* in the sense that one observation may exclude the occurrence of others, e.g. if the seller leaves positive feedback about the transaction, he can not leave negative or neutral feedback. An observation may *depend* on another in the sense that the first may only occur if the second has already occurred, e.g. the buyer cannot receive a confirmation of received payment if he has not made a payment. Finally, if two observations are neither in conflict nor dependent, they are said to be *independent*, and both may occur (in any order), e.g. feedback-events and receiving confirmation are independent. Note that ‘independent’ just means that the events are not in conflict nor dependent (e.g., it does *not* mean that the events are independent in any statistical sense). These relations between observations are directly reflected in the definition of an event structure. (For a general account of event structures, traditionally used in semantics of concurrent languages, consult the handbook chapter of Winskel and Nielsen [34]).

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

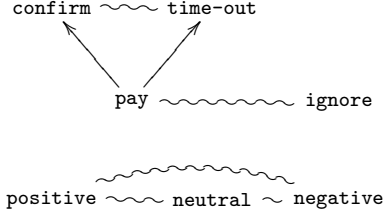
$$\text{the set } \lceil e \rceil \stackrel{\text{(def)}}{=} \{e' \in E \mid e' \leq e\} \text{ is finite.}$$

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

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

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.



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

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* iff 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  $e \in E$

$$(C.F.) d \# d'; \text{ and } (C.C.) e \leq d \Rightarrow e \in x$$

$\mathcal{C}_{ES}$  denotes the set of configurations of  $ES$ , and  $\mathcal{C}_{ES}^0 \subseteq \mathcal{C}_{ES}$  the set of *finite* configurations.

**Notation 2.1.** 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  $\rightarrow$  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  $x \not\xrightarrow{e} y$  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  $\emptyset, \{\text{pay}, \text{positive}\}$  and  $\{\text{pay}, \text{confirm}, \text{positive}\}$  are examples of configurations (the last configuration being maximal), whereas

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

and  $\{\text{confirm}\}$  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,  $h = x_1 x_2 \cdots x_n \in \mathcal{C}_{ES}^0$ . 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:

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

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

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

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

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

A possible variation of the model would be to also record this temporal information. We have chosen not to do so because it would complicate the model, and, further, we have found no example policies where this is necessary.

When applying our logic (described in the next section) to express policies for history-based access control (HBAC), we often use a special type of event structure in which the conflict relation is the maximal irreflexive relation on a set  $E$  of events. The reason is that *histories* in many frameworks for HBAC, are sequences of single events for a set  $E$ . When the conflict relation is maximal on  $E$ , the configurations of the corresponding event structure are exactly singleton event-sets, hence we obtain a useful specialization of our model, compatible with the tradition of HBAC. Note, the generalization from single events to configuration of an event structure makes the framework suitable for modelling interaction contexts in which the outcome of an interaction may not be known immediately after the interaction protocol has ended, e.g. in the eBay example, the buyer may observe its feedback at some point *after* the transaction has ended (and others might have started). A typical interaction history will have the form  $x_1 \cdots x_M x_{M+1} \cdots x_{M+K}$ , consisting of a prefix  $(x_i)_{i=1}^M$  of maximal sessions, followed by a suffix of possibly non-maximal sessions. The prefix of maximal sessions represent complete information about (typically old) protocol-runs, whereas the non-maximal sessions in the suffix, represent newer, ongoing protocol-runs for which the outcomes are not yet known.

### 3. A LANGUAGE FOR POLICIES

The reason for recording behavioural information is that it can be used to guide future decisions about interaction. We are interested in binary decisions, e.g., access-control and deciding whether to interact or not. In our proposed system, such decisions will be made according to interaction policies that specify exact requirements on local interaction histories. For example, in the eBay scenario from last section, the bidder may adopt a policy stating: “only bid on auctions

run by a seller which has never failed to send goods for won auctions in the past.”

In this section, we propose a declarative language which is suitable for specifying interaction policies. In fact, we shall use a pure-past variant of linear-time temporal logic, a logic introduced by Pnueli for reasoning about parallel programs [25]. 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  $\psi$ .

### 3.1 Formal Description

We describe the syntax and semantics of our variant of temporal logic.

#### 3.1.1 Syntax.

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

$$\psi ::= e \mid \diamond e \mid \psi_0 \text{ op } \psi_1 \mid \neg\psi \mid \mathbf{X}^{-1}\psi \mid \psi_0 \mathbf{S} \psi_1$$

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  $\mathbf{X}^{-1}$  (‘last time’) and  $\mathbf{S}$  (‘since’) are the usual past time operators, i.e.  $\mathbf{X}^{-1}\psi$  is true if  $\psi$  holds in the previous session, and  $\psi_0 \mathbf{S} \psi_1$  is true if  $\psi_1$  holds of some session in the past, and  $\psi_0$  holds of *every session since then*.

#### 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  $ES$ ,  $h \in \mathcal{C}_{ES}^0$ . The semantics is mostly standard, but will be presented in a non-standard (inductive) way. This facilitates the understanding of the algorithms, presented in the next section, for checking whether a history satisfies the requirements of a policy. 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  $\psi$ . 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  $(h, |h|) \models \psi$ . We define the semantics of  $(h, i) \models \psi$  inductively in  $i$ . Let  $h = x_1 x_2 \dots x_N \in \mathcal{C}_{ES}^0$  with  $N > 0$ . Define  $(h, 1) \models \psi$  by structural induction in  $\psi$ .

$$\begin{aligned} (h, 1) \models e & \quad \text{iff } e \in x_1 \\ (h, 1) \models \diamond e & \quad \text{iff } e \notin x_1 \\ (h, 1) \models \psi_0 \wedge \psi_1 & \quad \text{iff } (h, 1) \models \psi_0 \text{ and } (h, 1) \models \psi_1 \\ (h, 1) \models \psi_0 \vee \psi_1 & \quad \text{iff } (h, 1) \models \psi_0 \text{ or } (h, 1) \models \psi_1 \\ (h, 1) \models \neg\psi & \quad \text{iff } (h, 1) \not\models \psi \\ (h, 1) \models \mathbf{X}^{-1}\psi & \quad \text{iff } \mathbf{false} \\ (h, 1) \models \psi_0 \mathbf{S} \psi_1 & \quad \text{iff } (h, 1) \models \psi_1 \end{aligned}$$

Assume that  $1 < i \leq N$ , and (inductively) that  $(h, i-1) \models \psi$  is defined for all  $\psi$ .

Define  $(h, i) \models \psi$  by structural induction in  $\psi$ :

$$\begin{aligned} (h, i) \models e & \quad \text{iff } e \in x_i \\ (h, i) \models \diamond e & \quad \text{iff } e \notin x_i \\ (h, i) \models \psi_0 \wedge \psi_1 & \quad \text{iff } (h, i) \models \psi_0 \text{ and } (h, i) \models \psi_1 \\ (h, i) \models \psi_0 \vee \psi_1 & \quad \text{iff } (h, i) \models \psi_0 \text{ or } (h, i) \models \psi_1 \\ (h, i) \models \neg\psi & \quad \text{iff } (h, i) \not\models \psi \\ (h, i) \models \mathbf{X}^{-1}\psi & \quad \text{iff } (h, i-1) \models \psi \\ (h, i) \models \psi_0 \mathbf{S} \psi_1 & \quad \text{iff } (h, i) \models \psi_1 \text{ or} \\ & \quad \text{[} [(h, i-1) \models \psi_0 \mathbf{S} \psi_1] \text{ and} \\ & \quad \text{[} [(h, i) \models \psi_0] \text{]} \end{aligned}$$

Note that the inductive semantics is equivalent to the usual semantics, in particular, our inductive ‘since’ is equivalent to the usual definition:  $(h, i) \models \psi_0 \mathbf{S} \psi_1$  iff there exists  $1 \leq j \leq i$  so that  $(h, j) \models \psi_1$  and for all  $j'$  with  $j < j' \leq i$ ,  $(h, j') \models \psi_0$ .

**Remarks.** There are two main reasons for restricting ourselves to the *pure-past* fragment of temporal logic (PPLTL). First of all, while Sistla and Clarke proved that the model-checking problem for linear temporal logic with future- and past-operators (LTL) is PSPACE-complete [31], 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. Secondly, PPLTL is an expressive language for stating requirements over past behaviour. This claim is justified in two ways. Firstly, we can encode a number of existing ad-hoc approaches for checking requirements of past behaviour (c.f. Example 3.3 and 3.4 in the next section). Secondly, although one could add future operators to obtain a seemingly more expressive language, a recent result of Laroussinie *et al.* quantifies exactly what is lost by this restriction [20]. 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.

Note that we have defined the semantics of the logic only for *non-empty* structures,  $h \in \mathcal{C}_{ES}^0$ . This means that policies cannot be interpreted if there has been no previous interaction. In practice it is up to each agent to decide if interaction should take place in the case of no past history. For the remainder of this paper we shall define  $\epsilon \models \psi$  iff  $\emptyset \models \psi$ , that is we (arbitrarily) identify the empty sequence ( $\epsilon$ ) with the singleton sequence consisting of only the empty configuration.

Finally, we define standard abbreviations:  $\mathbf{false} \equiv e \wedge \neg e$  for some fixed  $e \in E$ ,  $\mathbf{true} \equiv \neg\mathbf{false}$ ,  $\psi_0 \rightarrow \psi_1 \equiv \neg\psi_0 \vee \psi_1$ ,  $\mathbf{F}^{-1}(\psi) \equiv \mathbf{true} \mathbf{S} \psi$ ,  $\mathbf{G}^{-1}(\psi) \equiv \neg\mathbf{F}^{-1}(\neg\psi)$ . We also define non-standard abbreviation  $\sim e \equiv \neg\diamond e$  (pronounced ‘conflict  $e$ ’ or ‘ $e$  is impossible’).

### 3.2 Example Policies

To illustrate the expressiveness 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.”

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

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

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

*Example 3.2 (Log-in terminal).* Consider a scenario where a log-in server offers the service of letting a user attempt to log into a system by providing a user-name and a password. To counter certain brute-force password-attacks, the server wants to use a policy where a client is only allowed to attempt login if at least 30 seconds have passed since an occurrence of three consecutive failed logins. That is, if a client supplies a wrong (*username, password*)-pair three times in a row, then client must wait at least 30 seconds before being allowed to attempt login again. Consider the following event structure  $ES = (E, \leq, \#)$ . There are three events  $E = \{\text{fail}, \text{success}, \text{timeout}\}$ ,  $\leq$  is the discrete partial order, and  $\#$  is the maximal irreflexive relation,  $\# \stackrel{\text{def}}{=} (E \times E) \setminus \Delta(E)$ , where  $\Delta(E)$  is the diagonal  $\{(e, e) \mid e \in E\}$ . We express the server’s policy as

$$\psi^{\text{attempt-login}} \equiv \neg (\text{X}^{-1}\text{fail} \wedge \text{X}^{-1}\text{X}^{-1}\text{fail} \wedge \text{X}^{-1}\text{X}^{-1}\text{X}^{-1}\text{fail})$$

We take advantage of the abstractness of the event-structure model by assuming that the program responsible for generating events will keep track of time and issue event `timeout` every 30 seconds if no login has been attempted. A similar policy to the above can also express the ‘slowing down hogs’ policy of Edjlali *et al.* [9] which is used to complicate denial-of-service attacks by limiting the rate at which a program connects to a remote site.

*Example 3.3 (Chinese Wall).* The Chinese Wall policy is an important commercial security-policy [3], but has also found applications within computer science. In particular, Edjlali *et al.* [9] use an instance of the Chinese Wall policy to restrict program accesses to database relations. The Chinese Wall security-policy deals with subjects (e.g. users) and objects (e.g. resources). The objects are organized into *datasets* which, in turn, are organized in so-called *conflict-of-interest classes*. There is a hierarchical structure on objects, datasets and classes, so that each object has a unique dataset which, in turn, has a unique class. In the Chinese-Wall policy, any subject initially has freedom to access any object. After accessing an object, the set of future accessible objects is restricted: the subject can no longer access an object in the same conflict-of-interest class unless it is in a dataset already accessed. Non-conflicting classes may still be accessed.

We now show how our logic can encode any instance of the Chinese Wall policy. Following the model of Brewer *et al.* [3], we let  $S$  denote a set of *subjects*,  $O$  a set of *objects*, and  $L$  a labeling function  $L : O \rightarrow C \times D$ , where  $C$  is a set of *conflict-of-interest classes* and  $D$  a set of *datasets*. The interpretation is that if  $L(o) = (c_o, d_o)$  for an object

$o \in O$ , then  $o$  is in dataset  $d_o$ , and this dataset belongs to the conflict-of-interest class  $c_o$ . The hierarchical structure on objects, datasets and classes amounts to requiring that for any  $o, o' \in O$  if  $L(o) = (c, d)$  and  $L(o') = (c', d)$  then  $c = c'$ . The following ‘simple security rule’ defines when access is granted to an object  $o$ : “either it has the same dataset as an object already accessed by that subject, or, the object belongs to a different conflict-of-interest class.” [3] We can encode this rule in our logic. Consider an event structure  $ES = (E, \leq, \#)$  where the events are  $C \cup D$ , with  $(c, c') \in \#$  for  $c \neq c' \in C$ ,  $(d, d') \in \#$  for  $d \neq d' \in D$ , and  $(c, d) \in \#$  if  $(c, d)$  is not in the image of  $L$  (denoted  $\text{Img}(L)$ ). We take  $\leq$  to be discrete. Then a maximal configuration is a set  $\{c, d\}$  so that the pair  $(c, d) \in \text{Img}(L)$ , 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:

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

In this encoding we have one policy per object  $o$ . One may argue that the policy  $\psi^o$  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  $\psi^o$  according to the scheme above; it then checks  $\psi^o$  with respect to the current history of  $s$ .<sup>1</sup> If  $o$  has been queried before it simply checks  $\psi^o$  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 3.4 (One-Out-of-k).* The ‘one-out-of- $k$ ’ (OOok) access-control policy was introduced informally by Edjlali *et al.* [9]. Set in the area of access control for mobile code, the OOok scheme dynamically classifies programs into equivalence classes, e.g. “browser-like applications,” depending on their past behaviour. Edjlali *et al.* give, among others, the following example of an OOok policy classifying “browser-like” applications: “allow a program to connect to a remote site if and only if it has neither tried to open a local file that it has not created, nor tried to modify a file it has created, nor tried to create a sub-process.” Since this example implicitly quantifies over all possible files (for *any* file  $f$ , if the application tries to open  $f$  then it must have previously have created  $f$ ), it cannot be expressed directly in our basic language. In practice, one could use a “parametric” mechanism like that for Chinese Wall in the previous example. However, in Section 5 we propose a more elegant solution: we extend our basic language to include parameterized events, and quantification over parameters. In the following we instead show that, if one takes the *set-based* formalization of

<sup>1</sup>This check can be done in time linear in the history of subject  $s$ , e.g., using the array based-algorithm from the next section

OOok by Fong [11], we can encode all OOok policies. Since our model is sequence-based, it is richer than Fong’s shallow histories which are sets. An encoding of Fong’s OOok-model thus provides a good sanity-check as well as a *declarative* means of specifying OOok policies (as opposed to the more implementation-oriented security automata).

In Fong’s model of OOok, a finite number of application classes are considered, say,  $1, 2, \dots, k$ . Fong identifies an application class,  $i$ , with a *set of allowed actions*  $C_i$ .<sup>2</sup> To encode OOok policies, we consider an event structure  $ES = (E, \leq, \#)$  with events  $E$  being the set of all access-controlled actions. Fong allows  $E$  to be a countably infinite set, but we shall restrict to finite sets. As in the last example, we take  $\leq$  to be discrete, and the conflict relation to be the maximal irreflexive relation, i.e. a local interaction history in  $ES$  is simply a sequence of single events. Initially, a monitored entity (originally, a piece of mobile code [9]) has taken no actions, and its history (which is a set in Fong’s formalization) is  $\emptyset$ . If  $S$  is the current history, then action  $a \in E$  is allowed if there exists  $1 \leq i \leq k$  so that  $S \cup \{a\} \subseteq C_i$ , and the history is updated to  $S \cup \{a\}$ . For each action  $a \in E$  we define a policy  $\psi^a$  for  $a$ , expressing Fong’s requirement. Assume, without loss of generality, that the sets  $C_j$  that contain  $a$  are named  $1, 2, \dots, i$  for some  $i \leq k$ . The following formula  $\psi_j^a$  encodes the requirement that  $S \cup \{a\} \subseteq C_j$  for a *fixed*  $j \leq i$ .

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

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  $\psi$ , 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  $\psi$  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

<sup>2</sup>In fact, we do not believe that a set-based formalization captures all aspects of the original informal description of OOok. A property, such as “may only modify a file it has (previously) created” clearly has a temporal aspect that is not captured by a set-based semantics.

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 Rosu for model checking past-time LTL [13]. Their idea is essentially this: because of the recursive semantics, model-checking  $\psi$  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  $\psi_j$  of  $\psi$ , and (2) the truth of  $(h, m) \models \psi_i$  for all proper sub-formulas  $\psi_i$  of  $\psi$  (a sub-formula of  $\psi$  is proper if it is not  $\psi$  itself). The truth of the atomic sub-formulas of  $\psi$  in  $(h, m)$  can be computed directly from the state  $h_m$ , where  $h_m$  is the  $m$ th configuration in sequence  $h$ . For example, if  $\psi_3 = X^{-1}\psi_4 \wedge e$ , 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  $\psi$ , 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 Rosu 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 of DMC

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. We provide, in the appendix, (a fragment of) a Java implementation of the data structure, and the interested reader can consult this appendix for details regarding the description below.

The data structure  $DS$  has a vector, accessed by variable  $DS.h$ , storing configurations of  $ES$ , which we denote  $DS.h = (y_1, y_2, \dots, y_K)$ . 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  $\psi$ . 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  $\psi$ , denoted  $Sub(\psi)$ , such that the following property holds. Let there be  $n + 1$  sub-formulas of  $\psi$ , and let  $\psi_0 = \psi$ . The sub-formula enumeration  $\psi_0, \psi_1, \psi_2, \dots, \psi_n$  satisfies that if  $\psi_i$  is a proper sub-formula of  $\psi_j$  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  $h$ .<sup>3</sup> 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  $\psi$  (more precisely, by the integers  $0, 1, \dots, n$ , 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*  $h = x_1 \dots x_M \cdot DS.h$  is a model of sub-formula  $\psi_j$  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  $\psi$ , 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 initialised 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  $\psi$ ,  $O(|\psi|)$ .

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  $(h, M + k) \models \psi_j \iff DS.B_k[j] = \mathbf{true}$ . 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  $(h, M + i - 1) \models \psi_j$ .<sup>4</sup> This information can be used to correctly fill-in array  $i$ , in time linear in  $|\psi|$ , using the re-

<sup>3</sup>We do not consider, here, the case where  $DS.h$  is empty. For these details, consult the appendix.

<sup>4</sup>In the special case where  $DS.B_0$  has not yet been allocated (i.e. when the longest prefix of maximal configurations is empty), one can construct a correct array  $DS.B_1$  in time  $O(|\psi|)$  using the base case of the recursive semantics. Details are provided in the appendix, and, here, we do not consider this special case further.

cursive 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), 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 initialised with a policy  $\psi$  and an event structure ES, then initialisation 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, if the configurations of  $ES$  are represented with event-set bit-vectors, the space complexity of  $DS$  is  $O(K \cdot (|\psi| + |E|))$ .

## 4.2 Using Finite Automata

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  $\psi$  in a history  $h = x_1 x_2 \dots x_{M+K}$  as the string-acceptance problem for an automaton,  $A_\psi$ , 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  $\psi$  in our policy language.

The states of the automaton  $A_\psi$  will be boolean arrays of size  $|\psi|$ , i.e. indexed by the sub-formulas of  $\psi$ . 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  $\psi$  with a set  $s \in \mathbf{2}^{Sub(\psi)}$ , i.e.  $B[j] = \mathbf{true}$  iff  $\psi_j \in s$ . The recursive semantics for a fixed formula  $\psi$ , can be seen as an algorithm, denoted  $RecSem$  (corresponding to class  $RecSem.java$  of the appendix), 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  $A_\psi$ ).** Let  $\psi$  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.  $\delta_\psi(s, x) = RecSem(s, x)$  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  $\psi$ .

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

**Lemma 4.1 (Automaton Invariant).** *Let  $h \in \mathcal{C}_{ES}^+$  be any non-empty history, and  $\psi_j$  be any sub-formula of  $\psi$ . 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.* This can be proved by induction in  $h$ , essentially since  $\delta_\psi$  is defined by the recursive semantics.  $\square$

**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  $F$ .  $\square$

In the abstract setting of automaton  $A_\psi$ , 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  $\psi$ . Initialization now simply consists of constructing the automaton  $A_\psi$ . More specifically, we construct the transition-matrix of  $\delta_\psi$  so that  $\delta_\psi(s, x)$  can be computed in time  $O(1)$  by a matrix-lookup.<sup>5</sup>  $DS'$  maintains a variable  $DS'.s_{\text{summ}}$  of type  $S$  (the automaton states) which is initialized to  $s_0$ . In addition to  $s_{\text{summ}}$ ,  $DS'$  will store a vector of pairs  $DS'.h = [(y_1, s_1), (y_2, s_2), \dots, (y_K, s_K)]$ , 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  $A_\psi$  is in after reading  $y_i$ . Initially this vector is empty.

<sup>5</sup>We choose a transition-matrix representation of  $\delta_\psi$  for simplicity. In practice, any representation allowing efficient computations of  $\delta_\psi(s, x)$  could be used.

**Invariance.** Let  $h = x_1x_2 \dots x_M \cdot y_{M+1} \dots y_{M+K}$  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  $DS'.h = [(y_1, s_1), (y_2, s_2), \dots, (y_K, s_K)]$  then  $(y_1, \dots, y_K)$  are the active configurations of  $h$ , and  $s_i$  is the state of the automaton after reading the string  $x_1x_2 \dots x_M \cdot y_1 \dots y_i$ , when started in state  $s_0$ . The invariant regarding the special variable  $DS'.s_{\text{summ}}$  is simply that  $DS'.s_{\text{summ}} = \hat{\delta}_\psi(s_0, x_1x_2 \dots x_M)$ , i.e.  $DS'.s_{\text{summ}}$  “summarizes” the history up to time  $M$  with respect to formula  $\psi$ . Notice that the invariant is satisfied after initialization.

**Operations.** All operations are now very simple. 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  $h \models \psi$ .<sup>6</sup> 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  $s_{\text{summ}}$  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  $DS'.s_{\text{summ}}$  appropriately).

Since  $\delta_\psi$  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  $\psi$  and an event structure  $ES = (E, \leq, \#)$ , then initialization of  $DS'$  is  $O(2^{|\psi|} \cdot |\mathcal{C}_{ES}| \cdot |\psi|)$ . At any time during execution, the complexity of the interface operations is:*

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

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

### 4.3 Remarks

The array- and automata-based implementations are very similar. The automata-based implementation simply pre-computes a matrix of transitions  $B \xrightarrow{x} B'$  instead of re-computing 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

<sup>6</sup>In the case where  $DS'.h$  is empty,  $DS'.s_{\text{summ}} \in F$  is returned. We consider this case no further.

is exponential in  $\psi$  (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  $A_\psi$ . We conjecture 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  $s_{\text{summ}}$  is updated in order to obtain optimizations, e.g. removing states that are unreachable in the future.

## 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  $G^{-1}(\text{open-f} \rightarrow F^{-1}\text{create-f})$  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

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

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  $G^{-1}(\text{negative} \rightarrow \text{ignore})$ . 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 each event  $e$  can occur with different parameters (e.g.

**open**(“/etc/passwd”) or **open**(“./tmp.txt”). In a configuration, the occurred events occur with exactly one concrete parameter.

#### 5.1.1 Parameterized Event Structures

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

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

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

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

$$\forall e \in \text{dom}(x). x(e) \in \rho(e)$$

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

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

$$\text{update}(h, e, p, i) = x_1 x_2 \dots ([e \mapsto p]/x_i) \dots x_n$$

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  $x, y, \dots$ ). Let the meta-variables  $v, u$  range over  $Val \stackrel{(def)}{=} Var \cup \bigcup_{i=1}^{\infty} P_i$ , and metavariable  $p$  range over  $\bigcup_{i=1}^{\infty} P_i$ .

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

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

We need some terminology.

- Write  $fv(\psi)$  for the set of free variables in  $\psi$  (defined in the usual way).
- A *policy* of the quantified language is a closed formula.
- Let  $\psi$  be any formula. Say that a variable  $x$  has type  $P_i$  in  $\psi$  if it occurs in a sub-formula  $e(x)$  of  $\psi$  and  $\rho(e) = P_i$ .

We impose the following static well-formedness requirement on formulas  $\psi$ . All free variables have unique type, and, if  $x$  is a bound variable of type  $P_i$  in  $\psi$ , then  $x$  is bound by a quantifier of the correct type (e.g., by  $\forall x : P_i. \psi$ ). Further, for each occurrence of  $e(p)$ ,  $p$  is of the correct type:  $p \in \rho(e)$ .

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

$$\begin{aligned} (h, i) \models^\sigma e(v) & \text{ iff } e \in \text{dom}(x_i) \text{ and } x_i(e) = \sigma(v) \\ (h, i) \models^\sigma \diamond e(v) & \text{ iff } e \notin \text{dom}(x_i) \text{ and } (e \in \text{dom}(x_i) \Rightarrow x_i(e) = \sigma(v)) \\ (h, i) \models^\sigma \psi_0 \wedge \psi_1 & \text{ iff } (h, i) \models^\sigma \psi_0 \text{ and } (h, i) \models^\sigma \psi_1 \\ (h, i) \models^\sigma \psi_0 \vee \psi_1 & \text{ iff } (h, i) \models^\sigma \psi_0 \text{ or } (h, i) \models^\sigma \psi_1 \\ (h, i) \models^\sigma \neg \psi & \text{ iff } (h, i) \not\models^\sigma \psi \\ (h, i) \models^\sigma \mathbf{X}^{-1} \psi & \text{ iff } i > 1 \text{ and } (h, i-1) \models^\sigma \psi \\ (h, i) \models^\sigma \psi_0 \mathbf{S} \psi_1 & \text{ iff } \exists j \leq i. ((h, j) \models^\sigma \psi_1) \text{ and } (\forall j < j' \leq i. (h, j') \models^\sigma \psi_0) \\ (h, i) \models^\sigma \forall x : P_j. \psi & \text{ iff } \forall p \in P_j. (h, i) \models^{((x \mapsto p)/\sigma)} \psi \\ (h, i) \models^\sigma \exists x : P_j. \psi & \text{ iff } \exists p \in P_j. (h, i) \models^{((x \mapsto p)/\sigma)} \psi \end{aligned}$$

*Example 5.1 (OOok Revisited).* Now we can model the OOok-example properly. Let  $String$  be the set of strings over some appropriate alphabet. Consider a parameterized event structure with two conflicting events **create** and **open**, each of type  $String$  (representing file-names). Consider the following quantified policy:

$$\psi^{q\text{-browser}} \equiv \mathbf{G}^{-1}(\forall x : String. (\text{open}(x) \rightarrow \mathbf{F}^{-1} \text{create}(x)))$$

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

Note also that this policy cannot be expressed in Fong’s set-based model. This follows since the above policy essentially depends on the *order* in which events occur (i.e. **create** before **open**).

### 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  $\psi_j$  of a formula  $\psi$ , a *constraint*  $C_k[j]$  on the free variables of  $\psi_j$ . The invariant will be that the sub-formula  $\psi_j$  is true for a substitution  $\sigma$  at time  $(h, k)$  if-and-only-if  $\sigma$  “satisfies” the constraint  $C_k[j]$ .

**Constraints.** Fix a quantified formula  $\psi$  and a history  $h = x_1 x_2 \cdots x_n \in \mathcal{C}_{\rho ES}^0$ . We assume for simplicity that all variables in  $\psi$  have the same type  $P$  (this restriction is inessential). Let  $P_h$  denote the set of distinct parameter occurrences in  $h$  (i.e.,  $P_h = \{q \in P \mid \exists e \in E \exists i \leq |h|. e \in \text{dom}(x_i) \text{ and } x_i(e) = q\}$ ). Let  $\Sigma_\psi$  denote the set of substitutions for the free variables of  $\psi$ , i.e.,  $\Sigma_\psi = fv(\psi) \rightarrow P_h$ . Note that since histories are finite objects, the set  $P_h$  is finite for any history  $h$ .

A function  $c : \Sigma_\psi \rightarrow \{\top, \perp\}$  is called a ( $\psi$ -) constraint (in  $h$ ). A substitution  $\sigma \in \Sigma_\psi$  satisfies constraint  $c$  if  $c(\sigma) = \top$ . In this case we write  $\sigma \models c$ . We write  $Constraint_\psi$  for the set of  $\psi$ -constraints (over some fixed history  $h$ ).<sup>7</sup>

Notice that when  $fv(\psi) = \emptyset$  then  $\Sigma_\psi \simeq \mathbf{1}$  (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  $\psi_j$  will be associated with a  $\psi_j$ -constraint  $C_k[j]$  in  $h$ , i.e., on the free variables of  $\psi_j$  (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

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

to

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

Notice that for closed  $\psi_j$ , the invariants are equivalent. It is also important to notice that constraints can be viewed as functions taking as input an  $n$ -ary vector of  $P_h$ -values (where  $n$  corresponds to the number of free 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  $c \wedge c'$  of two constraints  $c$  and  $c'$  (even if they are not on the same variables). 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  $\perp$  denote respectively the two constant constraints.

<sup>7</sup>If  $fv(\psi') \subseteq fv(\psi)$  then any  $\psi'$ -constraint can be thought of as a  $\psi$ -constraint (by imposing no additional requirements on the extra free variables).

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

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

We define function  $\llbracket \cdot \rrbracket_h^k$  as follows.

$$\llbracket e(v) \rrbracket_h^k = \begin{cases} x \in \{p\} & \text{if } v = x \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p \\ \top & \text{if } v = p \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p \\ \perp & \text{otherwise} \end{cases}$$

$$\llbracket \diamond e(v) \rrbracket_h^k = \begin{cases} x \in \{p\} & \text{if } v = x \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p \\ \top & \text{if } (v = p \text{ and } e \in \text{dom}(x_k) \text{ and} \\ & x_k(e) = p) \text{ or if} \\ & e \notin \text{dom}(x_k) \text{ and } e \# \text{dom}(x_k) \\ \perp & \text{otherwise} \end{cases}$$

We proceed inductively in  $\psi$ .

$$\begin{aligned} \llbracket \psi_0 \text{ op } \psi_1 \rrbracket_h^k &= \llbracket \psi_0 \rrbracket_h^k \text{ op } \llbracket \psi_1 \rrbracket_h^k \\ \llbracket \neg \psi \rrbracket_h^k &= \neg \llbracket \psi \rrbracket_h^k \\ \llbracket \mathbf{X}^{-1} \psi \rrbracket_h^k &= \llbracket \psi \rrbracket_h^{k-1} \\ \llbracket \psi_0 \mathbf{S} \psi_1 \rrbracket_h^k &= \llbracket \psi_1 \rrbracket_h^k \vee (\llbracket \psi_0 \rrbracket_h^k \wedge \llbracket \psi_1 \rrbracket_h^{k-1}) \\ \llbracket \forall x : P_i. \psi \rrbracket_h^k &= \text{elim}_x(\llbracket \psi \rrbracket_h^k) \\ \llbracket \exists x : P_i. \psi \rrbracket_h^k &= \text{elim}'_x(\llbracket \psi \rrbracket_h^k) \end{aligned}$$

All the clauses are straightforward except for  $\forall x : P_i. \psi$  and  $\exists x : P_i. \psi$ , which are handled by auxiliary functions  $\text{elim}_x$  and  $\text{elim}'_x$ . We define these functions now. Consider first  $\forall x : P_i. \psi$ . Assuming we have access to  $c = \llbracket \psi \rrbracket_h^k$  so that  $\sigma \models c \iff (h, k) \models^\sigma \psi$ , we must produce a new constraint  $c'$  which doesn't mention variable  $x$ , so that

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

The function  $\text{elim}_x$  does this; it transforms a constraint  $c$  into a constraint  $\text{elim}_x(c)$  with  $x \notin \text{fv}(\text{elim}_x(c))$ , satisfying the above equivalence. Function  $\text{elim}_x$  is the composition of two functions,  $\text{elim}_x = \text{conjoin} \circ \text{split}_x$ , where  $\text{split}_x(c)$  is a partial function of type  $P_i \cup \{\star\} \rightarrow \text{Constraint}_\psi$  (where  $\star$  is a special element not in  $P_i$ ). Function  $\text{conjoin}$  maps its input partial-function into a single constraint by taking the conjunction of all the formulas in the image of the partial function. We explain in detail now.

Two important properties hold of  $\text{split}_x(c)$ . Property (i):

$$\text{dom}(\text{split}_x(c)) \text{ is finite with } \star \in \text{dom}(\text{split}_x(c))$$

and Property (ii): for all substitutions  $\sigma$

$$\begin{cases} \sigma(x) \in \text{dom}(\text{split}_x(c)) \Rightarrow [\sigma \models c \iff \sigma \models \text{split}_x(c)(\sigma(x))] \\ \sigma(x) \notin \text{dom}(\text{split}_x(c)) \Rightarrow [\sigma \models c \iff \sigma \models \text{split}_x(c)(\star)] \end{cases}$$

We can think of  $\text{split}_x(c)$  as splitting the set  $\Sigma_\psi$  of substitutions into finitely many equivalence classes depending on where in  $P_i$  they map variable  $x$ . Each class is represented either by a singleton point  $p \in P_i$  or the special ‘‘class’’  $\star$  representing ‘‘the rest’’ of  $P_i$ . We associate with each class  $q$ , a constraint formula  $\text{split}_x(c)(q)$  (not containing variable  $x$ ). This splitting of  $\Sigma_\psi$  is so that for any substitution  $\sigma \in \Sigma_\psi$ , if  $\sigma$  maps  $x$  to a class  $q$  in  $\text{dom}(\text{split}_x(c))$  then the formula associated with  $q$  (i.e.  $\text{split}_x(c)(q)$ ) is equivalent to the original formula  $c$  under  $\sigma$ . If  $\sigma$  doesn't map  $x$  to  $\text{dom}(\text{split}_x(c))$  then  $c$  is equivalent to  $\text{split}_x(c)(\star)$  under  $\sigma$ . It is not hard to see that this property (i.e. Property (ii)) implies that for all substitutions  $\sigma$ ,

$$[\sigma \models \bigwedge_{\substack{q \in \text{dom}(\text{split}_x(c))}} \text{split}_x(c)(q)] \iff \forall p \in P_i. ([x \mapsto p]/\sigma) \models c$$

[Split must be explained – it does however depend on the representation of constraints. Question: is ‘‘split’’ really the (generalized) Shannon Decomposition when constraints are MDD's (generalized BDDs)? ]

We summarize the essence of  $\text{split}_x$  and  $\text{conjoin}$  as a lemma.

**Lemma 5.1.** *Let  $\rho ES$  be a parameterized event structure, and  $h = x_1 \cdots x_n$  be a history in  $\rho ES$ . Let  $1 \leq k \leq n$ , and  $\phi = \forall x : P_i. \psi$ . Then for all substitutions  $\sigma$ ,*

$$\sigma \models \text{conjoin} \circ \text{split}_x(\llbracket \psi \rrbracket_h^k) \iff \forall p \in P_i. h, k \models^{([x \mapsto p]/\sigma)} \psi$$

*Proof.* By construction.  $\square$

Notice that we obtain the function  $\text{elim}'_x$  for existential quantification by  $\text{elim}'_x = \text{disjoin} \circ \text{split}_x$ , where  $\text{disjoin}$  is like  $\text{conjoin}$  but takes the disjunction instead of conjunction.

**Array-based Model Checking.** In the light of function  $\llbracket \cdot \rrbracket$  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  $n$  constraint-arrays  $DS^\forall.C_k[j]$  (for  $1 \leq k \leq n$ ), each array indexed by the sub-formulas of  $\psi$ . The constraint in  $C_k[j]$  will be  $C_k[j] = \llbracket \psi_j \rrbracket_h^k$ . The invariant implies that for any closed  $\psi$ ,

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

(we write  $\models c$ , and say that  $c$  is *valid*, if  $\forall \sigma. \sigma \models c$ ). Hence operation **check** is a validity check, which is easy since  $\text{vars}(DS^\forall.C_n[0]) = \emptyset$  when  $\psi$  is closed.<sup>8</sup> Operations **new** and **update** are essentially as in  $DS$  (with the generalization from booleans to constraints), and we omit further details here.

**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.

<sup>8</sup>In fact, by applying a set of rewrite rules as computation proceeds, we can ensure that  $DS^\forall.C_n[0]$  is *syntactically* either  $\perp$  or  $\top$ , and hence **check**() is a constant time operation.

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

*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  $\psi$  of the quantified language (over  $ES$ ). Say that a QMC instance  $(h, \psi)$  is in QMC if  $h \models \psi$ . A single element model is a model,  $h \in \mathcal{C}_{\rho ES}^0$ , with  $h = x$ , where  $x \in \mathcal{C}_{\rho ES}^0$ .

The quantified boolean formula (QBF) problem is the problem of deciding the truth of quantified formulas of the form  $Q_1x_1Q_2x_2 \cdots Q_nx_n.\phi(x_1, \dots, x_n)$ , where each  $Q_i$  is a quantifier ( $\forall$  or  $\exists$ ), and  $\phi$  is a quantifier-free boolean formula (i.e., a propositional formula) with  $fv(\phi) \subseteq \{x_1, \dots, x_n\}$ . The QBF problem is known to be PSPACE complete [33]. Given a QBF  $f = Q_1x_1Q_2x_2 \cdots Q_nx_n.\phi$ , construct an MC-instance as follows. Use a parameterized event structure with a single event  $\star$  having two possible parameters  $\perp$  and  $\top$ . Let  $h = [\star \mapsto \top]$  be a single element history. Construct formula  $\psi$  as  $\psi \equiv Q_1x_1Q_2x_2 \cdots Q_nx_n.\psi'$ , where  $x_1, \dots, x_n$  are the variables of  $f$ , and  $\psi'$  is  $\phi$  with each variable  $x_j$  replaced by  $\star(x_j)$ . Then  $f$  is satisfiable if-and-only-if  $(h, |h|) \models \psi$ .  $\square$

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  $\psi' \equiv Q_1x_1Q_2x_2 \cdots Q_nx_n.\psi$ , 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.

**Proposition 5.2 (Complexity Bound).** *[To be changed] Let formula  $\psi' \equiv Q_1x_1Q_2x_2 \cdots Q_nx_n.\psi$  where the  $Q_i$  are quantifiers,  $x_i$  variables, and  $\psi$  is a quantifier-free formula from the quantified language with  $fv(\psi) \subseteq \{x_1, \dots, x_n\}$ . Let  $h \in \mathcal{C}_{\rho ES}^0$ , and for each parameter set  $P_i$ , let  $p_i$  be the number of parameter occurrences from  $P_i$  in history  $h$ . The constraint based algorithm for model checking  $h \models \psi'$  can be made to run in time  $O(?)$  ( $? = |h||P|2^n$  something like that).*

*Proof.* Omitted due to space limitations.  $\square$

## 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  $\psi$  is a basic policy. The construct is intended to mean that principal  $p$ 's observations (about a subject) must satisfy past-time  $\psi$ . For quantitative properties, we introduce a counting operator  $\#$ , used e.g. in formula  $p : \#\psi$  which counts the number of  $p$ -observed sessions satisfying  $\psi$  (we use  $\#$  to avoid confusion with the conflict relation, often denoted by  $\#$ ).

To express referencing, we extend the basic syntax to include a new syntactic category  $\pi$  (for policy). Let  $Prin$  be

a collection of principal identities.

$$\pi ::= p : \psi \mid \pi_0 \wedge \pi_1 \mid \neg \pi \quad p \in Prin$$

The policy  $p : \psi$  means that the observations that  $p$  has made should satisfy  $\psi$ . 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  $Prin \rightarrow \mathcal{C}_{ES}^*$ ).

The quantitative extension is given by extending the category  $\psi$ . 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  $\llbracket \mathcal{R}_j \rrbracket \subseteq \mathbb{N}^k$ .

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

The denotation of the construct  $\#\psi$  is the number of sessions in the past which satisfy formula  $\psi$ , e.g.,  $\#\text{negative}$  counts the number of states in the past satisfying **negative**. So the denotation of  $\#\psi$  is a number, and the semantics of  $\mathcal{R}_j(\#\psi_1, \#\psi_2, \dots, \#\psi_k)$  is **true** iff  $(n_1, n_2, \dots, n_k) \in \llbracket \mathcal{R}_j \rrbracket$ , where  $n_i$  is the denotation of  $\#\psi_i$ . Finally, we extend also category  $\pi$ :

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

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

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,  $p, p_1, p_2, \dots, p_n \in Prin$  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:

$$\pi_p^{\text{bid}} \equiv p : G^{-1}(\text{negative} \rightarrow \text{ignore}) \wedge \bigwedge_{q \in \{p, p_1, \dots, p_n\}} q : \neg F^{-1}(\text{time-out})$$

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 [30]. Consider a scenario where a P2P file-server 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.”

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

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).

$$\pi_p^{\text{client-dl}} \equiv (p : \overline{\#}\text{dl}) + (\sum_{i=1}^n p_i : \overline{\#}\text{dl}) \leq 3 \cdot (p : \overline{\#}\text{ul} + (\sum_{i=1}^n p_i : \overline{\#}\text{ul}))$$

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

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

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

**Implementation remarks.** Dynamic model checking for the extended policy language can be 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. The construct  $p : \psi$ , where  $p$  is a principal identity, requires that  $p$ ’s interaction history (with the subject in question) satisfies  $\psi$ . This is handled simply by “sending formula  $\psi$ ” to  $p$ . Principal  $p$  maintains the truth of  $\psi$  with respect to its interaction history using the algorithms of last section, and sends the required value to the requesting principal when needed.<sup>9</sup> 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 [30]. We have the following result, assuming that the relations can be evaluated in constant time.

**Proposition 5.3.** *The basic policy language extended quantitative properties can be model checked efficiently.  $O(|h||\psi|)$ ?*

It does not make sense to consider the algorithmic complexity of referencing. The message complexity of referencing is constant however (one query and one reply). Note finally, that the automata-based algorithm does not easily extend: the (semantics of the) extended language is no-longer regular, e.g. illustrated by formula  $p : (\overline{\#}\text{dl} \leq \overline{\#}\text{ul})$ .

## 6. CONCLUSION

We have presented a mathematical framework for what we have named ‘concrete’ reputation-based trust-management-systems. Our approach differs from most existing systems in that reputation information has an exact semantics, and

<sup>9</sup>One might argue that this leads to problems of timing: at what point in time is  $\psi$  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  $\psi$  thus knows that received valuation (true or false) reflects an evaluation of  $\psi$  with respect to some *recent view* of  $p$ ’s history.

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, existing approaches to “history based” access control, e.g., Chinese Wall and policies for controlling mobile code.

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

The notion of time in our temporal logic is based on when sessions are *started*. More precisely, our models are local interaction histories,  $h = x_1x_2 \cdots x_n$  where  $x_i \in \mathcal{C}_{ES}$ , and the order of the sessions reflects *the order in which the corresponding interaction-protocols are initiated*, i.e.  $x_i$  refers to the observed events in the  $i$ th-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  $x_i$  (the algorithms considered in Section 4 can be straightforwardly be adapted to this notion of time).

**Related Work.** Many reputation-based systems have been proposed in the literature (Jøsang *et al.* [15] provide many references), so we choose to mention only a few typical examples and closely related systems. Kamvar *et al.* present EigenTrust [16], Shmatikov and Talcott propose a license-based framework [30], and the EU project ‘SECURE’ [4, 5] (which also uses event structures for modelling observations [18,24]) 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” [30]). 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 [26]. 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 [28] 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  $\exists x : P_i.\psi$  (we can have  $\exists x$  either as a primitive or we can encode it as  $\exists x : P_i.\psi \equiv \neg\forall x : P_i.\neg\psi$ ), while their language is implicitly universally quantified.

The notion of security automata, introduced by Schneider [29], 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, 29, 32]). 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, 29]; 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, 32]. 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.

**Future Work.** For simplicity we have considered a simple first order temporal logic without function or relation symbols. There is no reason to expect additional difficulty

of model checking when adding these to our logic. Supposing we have (interpreted) relation symbols, then the “real” Chinese Wall policy could be encoded as

$$\psi^{Chinese} \equiv \forall x : O \exists c : C \exists d : D. \mathcal{L}(x, c, d) \wedge (\mathbf{F}^{-1}d \vee \mathbf{G}^{-1}\neg c)$$

Where  $\mathcal{L}$  is a relation symbol representing function  $L : O \rightarrow C \times D$ , i.e.,  $\mathcal{L}(o, c, d)$  holds iff  $L(o) = (c, d)$ . We plan to formalize this extension in future work.

The PSPACE-hardness of the quantified model-checking problem seems discouraging at first sight. However, the complexity bound in Proposition 5.2 leaves room for some optimism. It seems that, as long as policies do not have “too many” quantifiers, the problem is still tractable. Further, the bound in Proposition 5.2 is worst-case, and even in worst-case, the bound likely to be too loose. Future work includes tightening the bound, as well as experimenting with implementations to explore how well the algorithm performs in “average” case.

Finally, we are planning on implementing the algorithms described in this paper, in an experimental policy-tool. We conjecture that using the automata-based algorithm for the basic language will work very well in practice, in spite of the worst-case exponential-space requirement. The main reason for our conjecture is automata minimization. For example, it is not hard to see that many of the example policies in this paper have very small minimal automata. A concrete implementation would be helpful to test the hypothesis further. The MONA tool [17, 22], developed at BRICS, contains an advanced package for finite automata, which may prove itself useful in this respect.

## 7. REFERENCES

- [1] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005, 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] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In J. Vitek and C. D. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 1999.
- [3] D. F. Brewer and M. J. Nash. The chinese wall security policy. In *Proceedings from the 1989 IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, 1989.
- [4] V. Cahill and E. Gray *et al.* Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, 2003.
- [5] V. Cahill and J.-M. Seigneur. The SECURE website. <http://secure.dsg.cs.tcd.ie>, 2004.
- [6] M. Carbone, M. Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *Proceedings*

- from *Software Engineering and Formal Methods (SEFM'03)*. IEEE Computer Society Press, 2003.
- [7] M. Carbone, M. Nielsen, and V. Sassone. A calculus for trust management. In *Proceedings from Foundations of Software Technology and Theoretical Computer Science: 24th International Conference (FSTTCS'04)*, pages 161–173. Springer, December 2004.
- [8] eBay Inc. The eBay website. <http://www.ebay.com>, 2004.
- [9] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings from the 5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 38–48. ACM Press, 1998.
- [10] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings from the 2000 DARPA Information Survivability Conference and Exposition*, pages 1287–1295. IEEE Computer Society Press, 2000.
- [11] P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings from the 2004 IEEE Symposium on Security and Privacy*, pages 43–55. IEEE Computer Society Press, 2004.
- [12] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.
- [13] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2002.
- [14] A. Jøsang and R. Ismail. The beta reputation system. In *Proceedings from the 15th Bled Conference on Electronic Commerce, Bled*, 2002.
- [15] A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation for online service provision. *Decision Support Systems*, (to appear, preprint available online: <http://security.dstc.edu.au/staff/ajosang>), 2004.
- [16] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proceedings from the twelfth international conference on World Wide Web, Budapest, Hungary*, pages 640–651. ACM Press, 2003.
- [17] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, 2001. BRICS Notes Series, <http://www.brics.dk>.
- [18] K. Krukow. On foundations for dynamic trust management. Unpublished PhD Progress Report, available online: <http://www.brics.dk/~krukow>, 2004.
- [19] K. Krukow and A. Twigg. Distributed approximation of fixed-points in trust structures. To appear in *Proceedings from the 25th International Conference on Distributed Computing Systems (ICDCS'05)*. Preprint available online: <http://www.brics.dk/~krukow>, 2005.
- [20] 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.
- [21] 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.
- [22] A. Møller. The MONA project website. <http://www.brics.dk/mona>, 2004.
- [23] M. Nielsen and K. Krukow. Towards a formal notion of trust. In *Proceedings from the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 4–7. ACM Press, 2003.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara. Reputation systems. *Communications of the ACM*, 43(12):45–48, Dec. 2000.
- [28] 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.
- [29] F. B. Schneider. Enforceable security policies. *Journal of the ACM*, 3(1):30–50, 2000.
- [30] V. Shmatikov and C. Talcott. Reputation-based trust management. *Journal of Computer Security*, 13(1):167–190, 2005.
- [31] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

- [32] 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.
- [33] 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.
- [34] 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.

## APPENDIX

### A. IMPLEMENTATION-FRAGMENT OF THE ARRAY-BASED ALGORITHM

This appendix provides a fragment of a Java implementation of the array-based data-structure, *DS*, implementing interface *DMC*. We provide an implementation of two Java classes: *ArrayBasedDMC* and *RecSem*. The *ArrayBasedDMC* class implements the data structure described in Section 4.1, and uses *RecSem* which implements the recursive semantics. These classes are probably unintelligible for someone who has not read Section 4; the idea being to supplement the curious reader interested in more specific implementation-details.

**Caveat.** Our two implemented Java classes require additional implementations of the following classes: *Formula*, *EventFormula*, *PossibleFormula*, *Conjunction*, *Negation*, *LastTime* and *Since*, representing an implementation of the abstract syntax of interaction policies (c.f. Section 3). These classes should constitute a class-hierarchy, where class *Formula* is an abstract super-class of the others. Furthermore, implementations of classes *Event*, *EventStructure*, *Configurations* and *Configuration*, with the obvious meaning, is needed. The two classes *ArrayBasedDMC* and *RecSem* have been compiled in an environment providing interfaces for the above classes, but *no actual implementation has been tested*.

### Recursive Semantics

```
public class RecSem {
  /* Method baseCase implements the base case of the
  recursive semantics. The last parameter (cur) is a
  target array that will be filled according to the
  semantics.
  The base-case semantics takes a configuration x
  and a formula p, and outputs for each subformula
  f_i of p whether or not x |= f_i
  (this value is stored in cur[i], where i is the
  index for subformula f_i).
  More elegant software engineering would be to use
  the Visitor design-pattern to implement the
  recursive semantics, thus avoiding the clumsy
  'instanceof'.
  */
  public static void baseCase(Configuration x, Formula p,
    boolean[] cur) {
```

```
int i = p.getEnumerationSize()-1;
Formula f_i;
while (i >= 0) {
  f_i = p.getFormulaByIndex(i);
  if (f_i instanceof LastTime) { //f_i = X^{-1}(f)
    cur[i] = false;
  } else if (f_i instanceof Since) {
    Since s = ((Since) f_i);
    Formula f2 = s.getSecond();
    //f_i = -- Since f2
    cur[i] = cur[p.getIndexOf(f2)];
  } else //Case: Event, Possible,
    // Negation, Conjunction
    cur[i] = commonSemantics(x, f_i, cur, p);
  i--;
}
}
/* Similarly to method baseCase, this method
implements the inductive case of the recursive
semantics.
Again, the last parameter is a target array that will
be filled according to the recursive semantics.
*/
public static void inductiveCase(boolean[] last,
  Configuration x,
  Formula p,
  boolean[] cur) {
  int i = p.getEnumerationSize()-1;
  Formula f_i;
  while (i >= 0) {
    f_i = p.getFormulaByIndex(i);
    if (f_i instanceof LastTime) {
      //f_i = X^{-1}(f)
      Formula f = ((LastTime) f_i).getInner();
      cur[i] = last[p.getIndexOf(f)];
    } else if (f_i instanceof Since) {
      Since s = ((Since) f_i);
      Formula f1 = s.getFirst();
      Formula f2 = s.getSecond();
      //f_i = f1 Since f2
      cur[i] = cur[p.getIndexOf(f2)] ||
        (last[i] && cur[p.getIndexOf(f1)]);
    } else //Case: Event, Possible ...
      cur[i] = commonSemantics(x, f_i, cur, p);
    i--;
  }
}
/* Returns the semantics of x |= f_i (a boolean)
in the cases which are common of both the baseCase
and the inductiveCase semantics (in order to avoid
code duplication).
Precondition: !(f_i instanceof LastTime ||
  f_i instanceof Since)
*/
private static boolean commonSemantics(Configuration x,
  Formula f_i, boolean[] cur, Formula p) {
  if (f_i instanceof EventFormula) {
    Event e = ((EventFormula) f_i).getEvent();
    return x.contains(e);
  } else if (f_i instanceof Possible) {
    Event e = ((Possible) f_i).getEvent();
    return x.isPossible(e);
  } else if (f_i instanceof Conjunction) {
    Conjunction c = ((Conjunction) f_i);
    Formula f1 = c.getFirst();
```

```

        Formula f2 = c.getSecond();
        return cur[p.getIndexOf(f1)]
            && cur[p.getIndexOf(f2)];
    } else if (f_i instanceof Negation) {
        Formula f = ((Negation) f_i).getInner();
        return ! cur[p.getIndexOf(f)];
    } else
        throw new RuntimeException("Formula not in " 90
            + "basic language: "
            + f_i.toString());
    }
}

```

---

## DMC Interface

---

```

public interface DMC {
    public boolean check();
    public void New();
    public void update(Event e,int i);
}

```

---

## Array-Based DMC

---

```

import java.util.Vector;
public class ArrayBasedDMC implements DMC {
    private EventStructure es;
    private Formula p; // interaction policy
    private int policySize; // |Sub(p)|

    private Vector history_h; //DS.h
    private Vector arrays_B; //DS.B
    private boolean[] B0; //DS.B0 - summarising array

    private boolean empty_longest_prefix = true; 10
    //needed to deal with special-case of an empty
    //maximal prefix in the actual interaction history

    public ArrayBasedDMC(EventStructure es,Formula p) {
        p.enumerateSubformulas();
        policySize = p.getEnumerationSize();
        this.es = es;
        history_h = new Vector();
        arrays_B = new Vector(); 20

        //initialise B0 to contain values of  $\emptyset$  | = f_i
        //where f_i are the subformulas of p
        B0 = new boolean[policySize];
        RecSem.baseCase(es.configurations().getEmpty(),p,B0);
    }

    public boolean check() {
        if (arrays_B.isEmpty()) return B0[0];
        return ((boolean[]) arrays_B.lastElement())[0]; 30
    }

    public void New() {
        Configuration empty = es.configurations().getEmpty();
        boolean[] B_new = new boolean[policySize];
        if (!arrays_B.isEmpty()) {
            boolean[] B_last = (boolean[])
                arrays_B.lastElement();

```

```

            RecSem.inductiveCase(B_last,empty,p,B_new);
        } else if (!empty_longest_prefix) { 40
            RecSem.inductiveCase(B0,empty,p,B_new);
        } else {
            //special case: arrays_B.isEmpty() && empty_longest_prefix
            //(can only occur once: at beginning of first session)
            B_new = B0;//equivalent to RecSem.baseCase(empty,p,B_new)
            B0 = null;//not needed until empty_longest_prefix == false
        }
        history_h.add(empty);
        arrays_B.add(B_new);
    } 50
    /* Precondition: 1 <= i <= history_get.size() and
        not ((Configuration) history_h.get(i-1)).inConflict(e)
    */
    public void update(Event e,int i) {
        int vector_index = i - 1;
        Configuration x = (Configuration)
            history_h.get(vector_index);
        x.addEvent(e);
        boolean[] curB = (boolean[])
            arrays_B.get(vector_index); 60
        if (vector_index == 0) {
            if (empty_longest_prefix) {
                RecSem.baseCase(x,p,curB);
            } else { //non-empty maximal prefix
                RecSem.inductiveCase(B0,x,p,curB);
            }
        } else { //vector_index > 0
            RecSem.inductiveCase((boolean[])
                arrays_B.get(vector_index-1),
                x,p,curB); 70
        }
        while (++vector_index < history_h.size()) {
            boolean[] lastB = curB;
            x = (Configuration) history_h.get(vector_index);
            curB = (boolean[]) arrays_B.get(vector_index);
            RecSem.inductiveCase(lastB,x,p,curB);
        }
        if (i == 1 && es.isMaximal(x))
            B0 = longestPrefixCleanup(); 80
    }
    /* Deallocates unneeded arrays and configurations.
        Returns the new summarising array
    */
    //Precondition: called only after a call, update(e,1), where
    // es.isMaximal(history_h.get(0)) == true
    private boolean[] longestPrefixCleanup() {
        empty_longest_prefix = false;
        //now non-empty maximal prefix exists.

        history_h.removeElementAt(0); 90
        boolean[] summary =
            (boolean[]) arrays_B.remove(0);
        while (history_h.size() > 0 &&
            es.isMaximal((Configuration)
                history_h.firstElement())) {
            history_h.removeElementAt(0);
            summary = (boolean[]) arrays_B.remove(0);
        }
        return summary; 100
    }
}

```

---