A Formal Framework for Concrete Reputation-Systems

Karl Krukow

BRICS, University of Århus, Denmark

(joint work (in progress!) with Mogens and Vladimiro Sassone, Uni. of Sussex)
Motivation & Background:
Formal Guarantees and ‘trust’?

✓ Background for this work . . .

★ Access-control (decision making) in GC systems.
  → Traditional mechanisms fail.

★ Reputation systems.
  → Dynamic: ‘trust’ changes as behaviour is observed.
Motivation & Background:
Formal Guarantees and ‘trust’?

Background for this work...

- Access-control (decision making) in GC systems.
  → Traditional mechanisms fail.

- Reputation systems.
  → Dynamic: ‘trust’ changes as behaviour is observed.

Can one give any ‘security guarantees’ in trust/reputation-based systems?

- e.g. “if principal \( p \) gains access to resource \( r \) at time \( t \), then the past behaviour of \( p \) up until time \( t \), satisfies requirement \( \psi_r \).”
Expected Plan

✓ Formalising ‘behavioural information’:
  ★ Event Structures

✓ A declarative language for writing ‘interaction policies’.

✓ Implementation: Dynamic Algorithms.
  ★ A dynamic algorithm for model-checking of pure-past linear temporal logic.
  ★ A dynamic algorithm for pure-past-linear-temporal-temporal-logic model-checking ;-) 

✓ Language extensions and comments . . .
Model of behaviour
By example: E-Bay

✓ E-Bay: Online Auction-House
  ★ Seller and Bidders.
  ★ A feedback mechanism: a simple reputation system.

✓ Post-auction protocol:
  ★ Buyer \((B)\) sends payment of amount due.
  ★ Seller \((S)\) sends a receipt to confirm payment, then ships item.
  ★ Optionally, both \(B\) and \(S\) may leave feedback:
    → positive, neutral, negative + possibly a comment.
E-Bay – buyer observations

☑ From the buyers point-of-view, various events may occur:

★ \textit{B} may choose to pay for item (\texttt{pay}):
  
  → \textit{S} may send a receipt (\texttt{confirm}).
  
  → \textit{S} may not send a receipt within a certain time-limit (\texttt{time-out}).
  
  → ... 

★ \textit{B} may choose not to pay (\texttt{ignore}).

★ At any time, \textit{S} may leave feedback about transaction.
  
  → \texttt{Positive} (\texttt{positive}).
  
  → \texttt{Neutral} (\texttt{neutral}).
  
  → \texttt{Negative} (\texttt{negative}).

★ ...
Structure of observations

✓ The information obtained as a result of running a protocol can be described by a set of observable events, $E$. 
Structure of observations

✓ The information obtained as a result of running a protocol can be described by a set of observable events, $E$.

✓ These events have structure.

★ Conflict: both cannot occur.
   $\rightarrow$ e.g. positive vs. negative.

★ Dependency: a pre-condition for an event to occur.
   $\rightarrow$ e.g. pay before confirm.

★ Independence: none of the above.
   $\rightarrow$ e.g. negative and ignore.
Modelling observations.

- Model: Event structures.
  - \( ES = (E, \leq, \#) \).
  - \( E \) models the set of ‘observable events’.
  - \( \leq \subseteq E \times E \): dependency relation.
  - \( \# \subseteq E \times E \): conflict relation.

- Example:

  ![Diagram showing event structures]

  - \( S: \text{confirm} \)
  - \( S: \text{time-out} \)
  - \( B: \text{pay} \)
  - \( B: \text{ignore} \)
  - \( S: \text{positive} \)
  - \( S: \text{neutral} \)
  - \( S: \text{negative} \)
Modelling behavioural information.

✓ Information about one session constitutes a configuration – a set of (observed) events satisfying:
   ★ Conflict Free.
   ★ Causally Closed.

✓ Examples and non-examples:
   ★ Ex.: $\emptyset$, \{pay, positive\} and \{pay, confirm, positive\}
   ★ n-Ex.: \{pay, confirm, positive, negative\} and \{confirm\}

✓ Maximality: a configuration is maximal if no event can be added to obtain a new configuration.
For simplicity assume one ‘observer’/‘server’ and one ‘subject’/‘client’.

An interaction history is a sequence $h = x_1 x_2 \cdots x_N$, where $x_i \in C_{ES}$.

Our observation model will support two operations: `new()` and `update(e, i)`, performed on its history:

- $h$.new() = $h \cdot \emptyset$
- $h$.update(e, i) = $x_1 x_2 \cdots x_{i-1} \cdot (x_i \cup \{e\}) \cdot x_{i+1} \cdots x_N$

This will be the interface used by any system generating events.
Policies for Interaction

✓ Whether the ‘server’ will interact with the ‘client’ or not, is a policy decision.

✓ Policies will state exact criteria on the past behaviour of an entity, required for interaction.

✓ Decisions are binary (‘yes’/‘no’).

✓ Ideally, policies are written in a declarative language.
Policy examples

✓ Consider E-Bay again.
  ★ Policy: “only bid on auctions run by a seller which has never failed to ship items from won and paid auctions in the past”.

✓ Log-in server:
  ★ Policy: “attempting login is not allowed for 30 seconds if there have been three consecutive failed login-attempts”.

Policy examples

✓ Consider E-Bay again.
  ★ Policy: “only bid on auctions run by a seller which has never failed to ship items from won and paid auctions in the past”.

✓ Log-in server:
  ★ Policy: “attempting login is not allowed for 30 seconds if there have been three consecutive failed login-attempts”.

✓ Logic is declarative and natural for policies.

✓ It seems natural to phrase policies in a “past tense” form.
Language for Policies

✓ In fact, we already know what models are.
   ★ Finite linear Kripke structures, $h = x_1 \cdots x_N$.

✓ Pure-past linear temporal logic.
   ★ $\psi ::= e \mid \Diamond e \mid \psi_0 \land \psi_1 \mid \neg \psi \mid X^{-1}\psi \mid \psi_0 \mathcal{S} \psi_1$

✓ Given a history $h$, checking if $h$ satisfies the requirements of policy $\psi$ is the model checking problem: $h \models \psi$.
   ★ Interpreted from last session, “towards” first, that is,
     $x_1x_2\cdots x_N \models \psi \iff (x_1x_2\cdots x_N, N) \models \psi$
Examples Revisited

✓ E-Bay policy: “only bid on auctions run by a seller which has never failed to ship items from won and paid auctions in the past”.
  \[ \psi^{\text{buy}} \equiv \neg F^{-1}(\text{time-out}) \]

✓ furthermore, “seller has never provided negative feedback in auctions where payment was made”.
  \[ \psi^{\text{buy}} \equiv \ldots \land G^{-1}(\text{negative } \rightarrow \text{ ignore}) \]

✓ Log-in server: “attempt-login is not allowed for 30 seconds if there have been three consecutive failed login-attempts”
  \[ \psi^{\text{attempt-login}} \equiv \neg (X^{-1}\text{fail} \land X^{-1}X^{-1}\text{fail} \land X^{-1}X^{-1}X^{-1}\text{fail}) \]
Let $h = x_1 x_2 \cdots x_N$, and $\psi$ be a policy. Define the usual satisfaction relation $(h, i) \models \psi$ in an unusual way.

- if $i = 1$ then \ldots
- if $1 < i \leq N$, assume that $(h, i - 1) \models \psi$ is defined. Define $(h, i) \models \psi$ by structural induction:

  \hspace{1cm} \rightarrow \psi = e : (h, i) \models e \iff e \in x_i

  \hspace{1cm} \rightarrow \psi = \Diamond e : (h, i) \models e \iff \text{it is not the case that } e \not\in x_i

  \hspace{1cm} \rightarrow \psi = \psi_0 \land \psi_1, \neg \psi' : \ldots

  \hspace{1cm} \rightarrow \psi = X^{-1} \psi' : (h, i) \models X^{-1} \psi' \iff (h, i - 1) \models \psi'

  \hspace{1cm} \rightarrow \psi = \psi_0 S \psi_1 : (h, i) \models \psi_0 S \psi_1 \iff ((h, i) \models \psi_1) \text{ or } ((h, i - 1) \models \psi_0 S \psi_1 \text{ and } (h, i) \models \psi_0)$
Dynamic Model Checking

✓ The recursive semantics gives rise to an efficient algorithm for checking $h \models \psi$ (Havelund).

✓ The algorithm extends to the dynamic problem, i.e. supporting operations $h.\text{new}(), h.\text{update}(e, i), h.\text{check}()$.

✓ Assume that $h = x_1 x_2 \cdots x_i \underbrace{x_{i+1} \cdots x_N}_{k}$, and there is a suffix $k$ active sessions.

✓ Store $k + 1$ arrays $B_0, B_1, \ldots, B_k$.

★ $B_0$ summarises $x_1 \cdots x_i$ with respect to $\psi$, i.e. $B_0[j] = \text{true} \iff x_1 \cdots x_i \models \psi_j$.

★ Similarly, $B_l[j] = \text{true} \iff x_1 \cdots x_{i+l} \models \psi_j$.

★ When $x_i \to x_i \cup \{e\}$, start at $i - 1$, and update arrays.
We have

- A model of behavioural information – event structures.
- The core of a declarative language for specifying requirement of an interaction history – Pure Past LTL.
- A dynamic algorithm for policy checking –
  - \(\text{check}() : O(1)\).
  - \(\text{new}() : O(|\psi|)\).
  - \(\text{update}(e, i) : O(k \cdot |\psi|)\).
  - Space required: \(O(k \cdot |\psi|) + \text{model}\).

What now?
A slightly different view

✓ Consider \( x_1x_2 \cdots x_N \) as a string over the alphabet \( C_{ES} \).

✓ For any pure past \( \psi \), the language \( \{ h \in C_{ES}^* \mid h \models \psi \} \) is regular.

✓ Consider a deterministic finite automaton

\[
A_\psi = (S, \Sigma, s_0, F, \delta_\psi)
\]

\[ S = 2^\psi \cup \{ s_0 \}, \Sigma = C_{ES}, F = \{ s \in 2^\psi \mid \psi \in s \} \]

\[ \delta_\psi : S \times C_{ES} \to S \) (given by the recursive semantics)

✓ Of course the theorem is:

\[
\mathcal{L}(A_\psi) = \{ h \in C_{ES}^* \mid h \models \psi \}.
\]
Model checking with automata

✔ Consider a new algorithm for the dynamic model checking problem: let
\[ h = x_1 x_2 \cdots x_m x_{m+1} \cdots x_{m+k} = n \]

★ Pre-computation: construct \( A_\psi \).
★ Store \( k + 1 \) references to \( A_\psi \) states, \( s_0, s_1, \ldots, s_k \), where \( k \) is the number of active sessions, i.e.
\[ s_i = \hat{\delta}(s_{\text{init}}, x_1 x_2 \cdots x_m x_{m+1} \cdots x_{m+i}) \]
★ \textbf{new}(): set \( s_{k+1} = \delta(s_k, \emptyset) \).
★ \textbf{check}(): iff \( s_k \in F \).
★ \textbf{update}(e, i): start automata in state \( s_{i-1} \), run on
\[ (x_i \cup \{e\}) x_{i+1} \cdots x_N \], recording the states.

✔ Cost?
Tailoring the logic
(a.k.a. ‘ad-hoc’ extensions)

✓ (At least) Two aspects of usual ‘reputation systems’ are not expressible by our policies.

★ Referencing.

\[ \psi^{\text{buy}} \equiv \ldots \land G^{-1}(\text{negative } \rightarrow \text{ ignore}) \]

– “and the same for all my friends \( p_1, p_2, \ldots, p_n \).”

★ Quantitative statements.

\[ \psi^{\text{buy}} \equiv \neg F^{-1}(\text{time-out}) \]

– “well, most of the time, anyway…”.
Tailoring the logic
(a.k.a. ‘ad-hoc’ extensions)

✓ (At least) Two aspects of usual ‘reputation systems’ are not expressible by our policies.

★ Referencing.

\[ \psi^{\text{buy}} \equiv \ldots \land G^{-1}(\text{negative} \rightarrow \text{ignore}) \]

– “and the same for all my friends \( p_1, p_2, \ldots, p_n \).”

★ Quantitative statements.

\[ \psi^{\text{buy}} \equiv \neg F^{-1}(\text{time-out}) \]

– “well, most of the time, anyway…”

✓ A policy \( \pi \) is given by \( \pi \equiv p : \psi \mid \pi_0 \land \pi_1 \mid \neg \pi \).

★ Referencing!
More ‘ad-hoc’ extensions

✓ Introduce a counting operator,

\[
\psi ::= \ldots | R_j^k(\#\psi_1, \#\psi_2, \ldots, \#\psi_k).
\]

★ meaning of \#\psi: count the number of states in the past (relative to current state) which satisfy \psi.

★ symbols \( R_j^k \) denoting efficiently computable \( k \)-ary relations \( [R_j^k] \), for each arity \( k \).
More ‘ad-hoc’ extensions

✓ Introduce a counting operator,

\[ \psi ::= \ldots | R^k_{kj}(\#\psi_1, \#\psi_2, \ldots, \#\psi_k). \]

★ meaning of \#\psi: count the number of states in the past (relative to current state) which satisfy \psi.

★ symbols \( R^k_{kj} \) denoting efficiently computable \( k \)-ary relations \( [R^k_{kj}] \), for each arrity \( k \).

✓ Examples:

\[ \pi^\text{buy}_p \equiv p : G^{-1}(\text{negative} \rightarrow \text{ignore}) \land \]
\[ \land_{q \in \{p, p_1, \ldots, p_n\}} q : \neg F^{-1}(\text{time-out}) \]
More ‘ad-hoc’ extensions

✓ Introduce a counting operator,

\[ \psi ::= \ldots | R_j^k(\#\psi_1, \#\psi_2, \ldots, \#\psi_k). \]

★ meaning of \(\#\psi\): count the number of states in the past (relative to current state) which satisfy \(\psi\).

★ symbols \(R_j^k\) denoting efficiently computable \(k\)-ary relations \([R_j^k]\), for each arity \(k\).

✓ Examples:

\[ \pi_{p}^{\text{buy}} \equiv p: G^{-1}(\text{negative} \to \text{ignore}) \land \]
\[ \land_{q\in\{p,p_1,\ldots,p_n\}} q: \neg F^{-1}(\text{time-out}) \]

★ \(\pi_{p}^{\text{client-dl}} \equiv p: (\#\text{dl} \leq 3 \cdot \#\text{ul}) \)
More ‘ad-hoc’ extensions

✓ Introduce a counting operator,
\[ \psi ::= \ldots \ | \ \mathcal{R}_j^k(\#\psi_1, \#\psi_2, \ldots, \#\psi_k). \]

★ meaning of \#\psi: count the number of states in the past (relative to current state) which satisfy \psi.
★ symbols \mathcal{R}_j^k denoting efficiently computable \( k \)-ary relations \( \llbracket \mathcal{R}_j^k \rrbracket \), for each arity \( k \).

✓ Examples:

\[ \pi^\text{buy}_p \equiv p : G^{-1}(\text{negative} \rightarrow \text{ignore}) \land \]
\[ \land_{q \in \{p,p_1,\ldots,p_n\}} q : \neg F^{-1}(\text{time-out}) \]
★ \[ \pi^\text{client-dl}_p \equiv p : (\#dl \leq 3 \cdot \#ul) \]
★ \[ \pi^\text{probab}_p \equiv p : \frac{\#ev}{\#ev + \#\sim ev + 1} \geq \frac{3}{4} \]
Extending the algorithm

Essentially the same algorithm for dynamic model-checking works:

- $\lceil \#\psi \rceil^{(h,i+1)}$ can be computed given
  - value of $\lceil \#\psi \rceil^{(h,i)}$ (last time); and
  - truth of $(h, i) \models \psi$ (in current state).

- $q : \psi$: e.g. register $\psi$ with $q$, and when required as $q$
  about the truth of $\psi$.

Finite Automata-based approach doesn’t work, i.e. language is non regular.

- e.g. $\psi \equiv p : \#\text{pay} > \#\text{access}$. 

Concluding

✓ A model of behavioural information.
   ★ Supports “partial information”.

✓ Pure-Past temporal logic for policies.
   ★ Extensions to include referencing, and quantitative policies.

✓ Efficient dynamic policy-checking.
   ★ ‘Plain’ and automata-based algorithms.