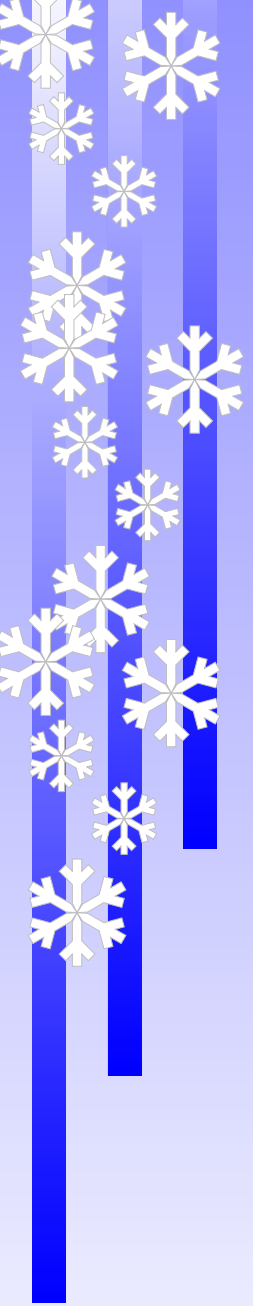


A Formal Framework for Concrete Reputation-Systems

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 ψ_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-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:
  - ★ B may choose to pay for item (`pay`):
    - → S may send a receipt (`confirm`).
    - → S may not send a receipt within a certain time-limit (`time-out`).
    - → ...
  - ★ B may choose not to pay (`ignore`).
  - ★ At any time, S may leave feedback about transaction.
    - → Positive (`positive`).
    - → Neutral (`neutral`).
    - → Negative (`negative`).
  - ★ ...

Structure of observations

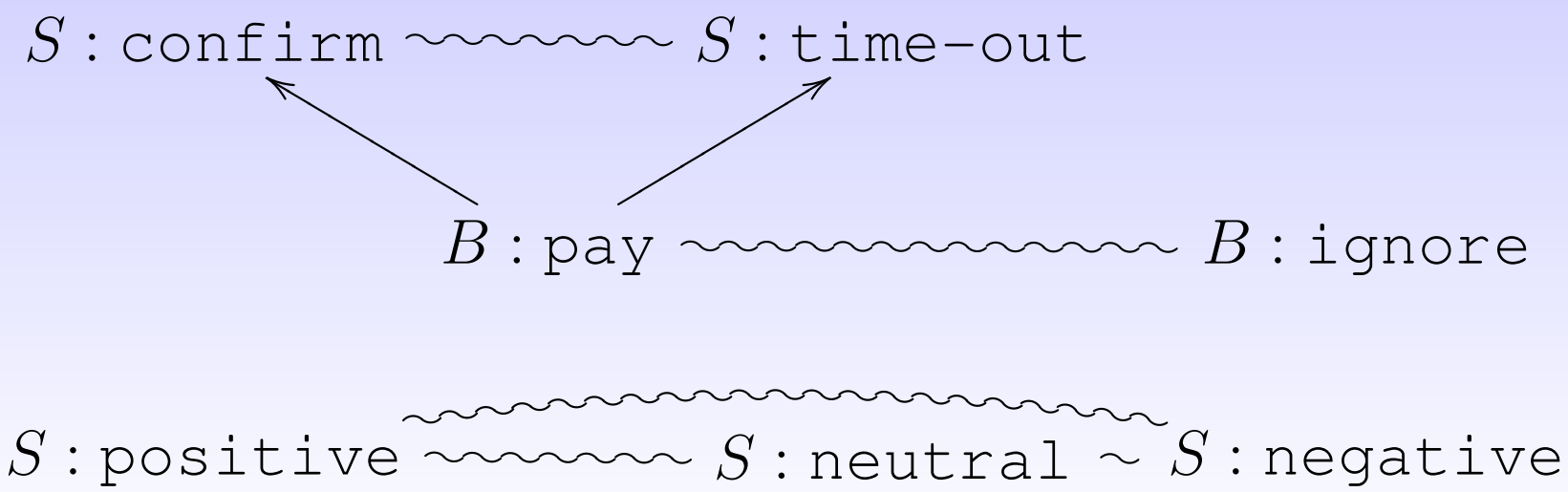
- ✓ The information obtained as a result of running a protocol can be described by a *set of observable events*, E.

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.
    - → e.g. `positive` vs. `negative`.
  - ★ Dependency: a pre-condition for an event to occur.
    - → e.g. `pay` before `confirm`.
  - ★ *In*dependence: none of the above.
    - → 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:



Modelling behavioural information.

- Information about one session consitutes a *configuration* – a set of (observed) events satisfying:
  - Conflict Free.
  - Causally Closed.
- Examples and non-examples:
  - Ex.: $\emptyset$, $\{\texttt{pay}, \texttt{positive}\}$ and $\{\texttt{pay}, \texttt{confirm}, \texttt{positive}\}$
  - n-Ex.:$\{\texttt{pay}, \texttt{confirm}, \texttt{positive}, \texttt{negative}\}$ and $\{\texttt{confirm}\}$
- Maximality: a configuration is *maximal* if no event can be added to obtain a new configuration.

Interface

- 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 $x_i \in \mathcal{C}_{ES}$.
- Our observation model will support two operations:
  $\mathbf{new}()$ and $\mathbf{update}(e, i)$, performed on its history:
  - $h.\mathbf{new}() = h \cdot \emptyset$
  - $h.\mathbf{update}(e, i) = x_1 x_2 \cdots x_{i-1} \cdot (x_i \cup \{e\}) \cdot x_{i+1} \cdots x_N$
- 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.
  - ★ $\psi ::= e \mid \Diamond e \mid \psi_0 \wedge \psi_1 \mid \neg\psi \mid \mathsf{X}^{-1}\psi \mid \psi_0 \,\mathsf{S}\, \psi_1$
- ✓ Given a history h, checking if h satisfies the requirements of policy ψ is the model checking problem: $h \models \psi$.
  - ★ Interpreted from last session, "towards" first, that is,
    $x_1 x_2 \cdots x_N \models \psi \iff (x_1 x_2 \cdots x_N, N) \models \psi$

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”.
  - ★ $\psi^{\mathsf{buy}} \equiv \neg \mathsf{F}^{-1}(\texttt{time-out})$
- ✓ furthermore, “seller has never provided negative feedback in auctions where payment was made”.
  - ★ $\psi^{\mathsf{buy}} \equiv \ldots \wedge \mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore})$
- ✓ Log-in server: “attempt-login is not allowed for 30 seconds if there have been three consecutive failed login-attempts”
  - ★ $\psi^{\mathsf{attempt\text{-}login}} \equiv \neg(\mathsf{X}^{-1}\texttt{fail} \wedge \mathsf{X}^{-1}\mathsf{X}^{-1}\texttt{fail} \wedge \mathsf{X}^{-1}\mathsf{X}^{-1}\mathsf{X}^{-1}\texttt{fail})$

Recursive Semantics

- ✓ Let $h = x_1 x_2 \cdots x_N$, and ψ be a policy. Define the usual satisfaction relation $(h, i) \models \psi$ in an unusual way.
  - ★ if $i = 1$ then ...
  - ★ if $1 < i \leq N$, assume that $(h, i-1) \models \psi$ is defined. Define $(h, i) \models \psi$ by structural induction:
    - → $\psi = e : (h, i) \models e \iff e \in x_i$
    - → $\psi = \Diamond e : (h, i) \models e \iff$ it is not the case that $e \# x_i$
    - → $\psi = \psi_0 \wedge \psi_1, \neg\psi' : \ldots$
    - → $\psi = \mathsf{X}^{-1}\psi' : (h, i) \models \mathsf{X}^{-1}\psi' \iff (h, i-1) \models \psi'$
    - → $\psi = \psi_0 \,\mathsf{S}\, \psi_1 : (h, i) \models \psi_0 \,\mathsf{S}\, \psi_1 \iff ((h, i) \models \psi_1)$ or $((h, i-1) \models \psi_0 \,\mathsf{S}\, \psi_1$ 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 $h.\mathbf{new}(), h.\mathbf{update}(e, i), h.\mathbf{check}()$.
- ✓ 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 ψ, i.e. $B_0[j] = \texttt{true} \iff x_1 \cdots x_i \models \psi_j$.
  - ★ Similarly, $B_l[j] = \texttt{true} \iff x_1 \cdots x_{i+l} \models \psi_j$.
  - ★ When $x_i \longrightarrow x_i \cup \{e\}$, start at $i - 1$, and update arrays.

Summary

- ✓ 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 –
    - → $\mathbf{check}() : O(1)$.
    - → $\mathbf{new}() : O(|\psi|)$.
    - → $\mathbf{update}(e, i) : O(k \cdot |\psi|)$.
    - → Space required: $O(k \cdot |\psi|)$ + model.
- ✓ What now?

A slightly different view

- ✓ Consider $x_1 x_2 \cdots x_N$ as a string over the alphabet $\mathcal{C}_{ES}$.
- ✓ For any pure past ψ, the language $\{h \in \mathcal{C}_{ES}^* \mid h \models \psi\}$ is regular.
- ✓ Consider a deterministic finite automaton $A_\psi = (S, \Sigma, s_0, F, \delta_\psi)$
  - ★ $S = \mathbf{2}^\psi \cup \{s_0\}, \Sigma = \mathcal{C}_{ES}, F = \{s \in \mathbf{2}^\psi \mid \psi \in s\}$
  - ★ $\delta_\psi : S \times \mathcal{C}_{ES} \to S$ (given by the recursive semantics)
- ✓ Of course the theorem is: $\mathcal{L}(A_\psi) = \{h \in \mathcal{C}_{ES}^* \mid h \models \psi\}$.

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_ψ.
  - ★ Store $k+1$ references to A_ψ states,$s_0, s_1, \ldots, s_k$, where k is the number of active sessions, i.e. $s_i = \hat{\delta}(s_{init}, x_1 x_2 \ldots x_m x_{m+1} \ldots x_{m+i})$.
  - ★ **new**(): set $s_{k+1} = \delta(s_k, \emptyset)$.
  - ★ **check**(): iff $s_k \in F$.
  - ★ **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.

→ $\psi^{\mathsf{buy}} \equiv \ldots \wedge \mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore})$

–"and the same for all my friends $p_1, p_2, \ldots, p_n$."

★ Quantitative statements.

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

– "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.
    - → $\psi^{\mathsf{buy}} \equiv \ldots \wedge \mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore})$

      –"and the same for all my friends $p_1, p_2, \ldots, p_n$."
  - ★ Quantitative statements.
    - → $\psi^{\mathsf{buy}} \equiv \neg\mathsf{F}^{-1}(\texttt{time-out})$

      – "well, most of the time, anyway…".
- ✓ A policy π is given by $\pi ::= p : \psi \mid \pi_0 \wedge \pi_1 \mid \neg\pi$.
  - ★ Referencing!

More ‘ad-hoc’ extensions

- ✓ Introduce a counting operator,
  $\psi ::= \ldots \mid \mathcal{R}^k_j(\#\psi_1, \#\psi_2, \ldots, \#\psi_k)$.
  - ★ meaning of $\#\psi$: count the number of states in the past (relative to current state) which satisfy ψ.
  - ★ symbols $\mathcal{R}^k_j$ denoting efficiently computable k-ary relations $[\![\mathcal{R}^k_j]\!]$, for each arrity k.

More ‘ad-hoc’ extensions

- ✓ Introduce a counting operator,
  $\psi ::= \ldots \mid \mathcal{R}^k_j(\#\psi_1, \#\psi_2, \ldots, \#\psi_k)$.
  - ★ meaning of $\#\psi$: count the number of states in the past (relative to current state) which satisfy ψ.
  - ★ symbols $\mathcal{R}^k_j$ denoting efficiently computable k-ary relations $[\![\mathcal{R}^k_j]\!]$, for each arrity k.
- ✓ Examples:
  - ★ $\pi_p^{\mathsf{buy}} \equiv \; p : \mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore}) \;\; \wedge$
    $\bigwedge_{q \in \{p, p_1, \ldots, p_n\}} q : \neg \mathsf{F}^{-1}(\texttt{time-out})$

More ‘ad-hoc’ extensions

- ✓ Introduce a counting operator,
  $\psi ::= \ldots \mid \mathcal{R}^k_j(\#\psi_1, \#\psi_2, \ldots, \#\psi_k)$.
  - ★ meaning of $\#\psi$: count the number of states in the past (relative to current state) which satisfy ψ.
  - ★ symbols $\mathcal{R}^k_j$ denoting efficiently computable k-ary relations $[\![\mathcal{R}^k_j]\!]$, for each arrity k.
- ✓ Examples:
  - ★ $\pi_p^{\mathsf{buy}} \equiv \; p : \mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore}) \;\; \wedge \bigwedge_{q \in \{p, p_1, \ldots, p_n\}} q : \neg \mathsf{F}^{-1}(\texttt{time-out})$
  - ★ $\pi_p^{\mathsf{client\text{-}dl}} \equiv p : (\#\texttt{dl} \leq 3 \cdot \#\texttt{ul})$

More ‘ad-hoc’ extensions

- ✓ Introduce a counting operator,
  $\psi ::= \ldots \mid \mathcal{R}^k_j(\#\psi_1, \#\psi_2, \ldots, \#\psi_k)$.
  - ★ meaning of $\#\psi$: count the number of states in the past (relative to current state) which satisfy ψ.
  - ★ symbols $\mathcal{R}^k_j$ denoting efficiently computable k-ary relations $[\![\mathcal{R}^k_j]\!]$, for each arrity k.
- ✓ Examples:
  - ★ $\pi_p^{\mathsf{buy}} \equiv \; p : \mathsf{G}^{-1}(\texttt{negative} \rightarrow \texttt{ignore}) \;\; \wedge \bigwedge_{q \in \{p, p_1, \ldots, p_n\}} q : \neg \mathsf{F}^{-1}(\texttt{time-out})$
  - ★ $\pi_p^{\mathsf{client\text{-}dl}} \equiv p : (\#\texttt{dl} \leq 3 \cdot \#\texttt{ul})$
  - ★ $\pi_p^{\mathsf{probab}} \equiv p : \frac{\#\texttt{ev}}{\#\texttt{ev} + \#\sim\texttt{ev} + 1} \geq \frac{3}{4}$

Extending the algorithm

- ✓ Essentially the same algorithm for dynamic model-checking works:
  - ★ $[\![\#\psi]\!]^{(h,i+1)}$ can be computed given
    - → value of $[\![\#\psi]\!]^{(h,i)}$ (last time); and
    - → truth of $(h, i) \models \psi$ (in current state).
  - ★ $q : \psi$: e.g. register ψ with q, and when required as q about the truth of ψ.
- ✓ Finite Automata-based approach doesn't work, i.e. language is non regular.
  - ★ e.g. $\psi \equiv p : \#\mathtt{pay} > \#\mathtt{access}$.

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.