Cassandra

References:

Becker, Moritz; Sewell, Peter. *Cassandra: Flexible Trust Management, Applied to Electronic Health Records*. 2004.

Li, Ninghui; Mitchell, John. *Datalog with Constraints: A Foundation for Trust Management Languages*. Jan. 2003.

Cassandra

- Cassandra is designed so that the expressiveness of the language can be tuned by selecting an appropriate *constraint domain*
- Research grounded in a substantial real-world example for a national Electronic Health Record (EHR) system based on the UK National Health Service procurement exercise
- Case study includes 310 rules with 58 roles
- Paper is a precursor to PhD thesis on the topic

Electronic Health Record (EHR)

- EHR schemes are now being developed in Europe, Canada and Australia to provide “cradle-to-grave” summaries of patients’ records.
- In England the National Health Service is developing a nationwide Integrated Care Records Service to provide patients with 24-hour on-line access to EHRs on a central data-spine.
- Deployment is scheduled between 2005 and 2010.
- Needless to say, the proposed system has proved highly controversial.

EHR Policy Concerns

- Patients will refuse to share their data if they do not trust the system or do not have sufficient control of use of their data.
- Physicians will not want to use the system if it is cumbersome to use, if the access restrictions are too strict, or the response times too high.
- Any policy system has to comply with relevant legislation and regulations such as the Data Protection Act, Mental Health Act, Human Fertilization and Embryology Act, the Abortion Regulations and the Venereal Diseases Regulations.

EHR Policy Concerns (cont.)

- It is evident that a policy must be adaptable enough not just to function when deployed, but to be adapted to incorporate emerging restrictions and extensions to reflect the evolution of the public debate
- Health organizations will likely have customized policies that are compatible but different from the nationally accepted one
- Other EHR countries may require radically different policy specifics and/or adopt separate policy systems

Trust Management

- Traditional *access control* relies on a notion of identity
- In decentralized, open, distributed systems the resource owner and requester are often unknown to one another
- Instead of identity access control decisions are based on *policy statements* made by multiple principles
- Some statements are digitally signed – these are called *credentials*
- Some statements are stored in local trusted storage – these are called *access rules*

Trust Management (cont.)

Previous work on trust management suggests it is desirable to:

- factor out the policy from the application code
- express policy in terms of roles instead of individuals
- support distributed access control with policies that express automatic credential retrieval over the network and strategies to establish mutual trust between strangers
- be scalable to large numbers of sites and entities
- be adaptable to different administrative domains with independent policies or local adaptations of a default policy

Trust Management Scenario

- A *requester* submits a request, possibly supported by a set of *credentials* issued by other parties, to an *authorizer*, which specifies access rules governing access to the requested resource(s).
- The *authorizer* may have to contact the parties that issued the *credentials* to import additional policy considerations.

Tensions in the Design of a Real-World Policy Language

- It should:
  - should be expressive so intended policies can be written naturally
  - small and elegant
  - avoid ad hoc features

- BUT, it should be efficiently computable in practical examples

Cassandra

- is a language and system for expressing access control policies
- supports credential-based access control and rules can refer to remote policies
- policy language is small and it has a formal semantics for query evaluation and for the access control engine
- policies are expressed in a language based on Datalog^C (Datalog with constraints)

Cassandra Syntax

Predicate names p ::= canActivate | hasActivated | permits | canDeactivate | isDeactivated | canReqCred, and user-defined predicate names

Policy rule: $E_{loc}@E_{iss}.p_0(\vec{e}_0) \leftarrow loc_1@iss_1.p_1(\vec{e}_1),\ \ldots,\ loc_n@iss_n.p_n(\vec{e}_n),\ c$

Credential (rule): $E_{loc}@E_{iss}.p_0(\vec{e}_0) \leftarrow c$

Aggregation rule: $E_{loc}@E_{loc}.p(\text{agg-op}\langle x\rangle, \vec{y}) \leftarrow E_{loc}@iss.q(\vec{x}), c$
where agg-op is group or count

$\mathcal{C}_{eq}$ expressions: $e ::= x \mid E$

$\mathcal{C}_{eq}$ constraints: $c ::= \mathsf{true} \mid \mathsf{false} \mid e = e' \mid c \wedge c' \mid c \vee c'$.

$\mathcal{C}_0$ expressions $e ::= x \mid E \mid N \mid C \mid () \mid (e_1, .., e_n) \mid \pi^n_i(e) \mid R(e_1, .., e_n) \mid A(e_1, .., e_n) \mid f(e_1, .., e_n) \mid \emptyset \mid \Omega \mid \{e_1, .., e_n\} \mid e - e' \mid e \cap e' \mid e \cup e'$

$\mathcal{C}_0$ constraints $c ::= \mathsf{true} \mid \mathsf{false} \mid e = e' \mid e \neq e' \mid e < e' \mid e \subseteq e' \mid c \wedge c' \mid c \vee c'$

and derivable constraints: $c ::= \ldots \mid e \in e' \mid e \notin e' \mid e \in [e_1, e_2] \mid [e_1, e_2] \subseteq [e'_1, e'_2]$

$\mathcal{C}_0$ types $\tau ::= \mathit{entity} \mid \mathit{int} \mid \mathit{const} \mid \mathit{unit} \mid \tau_1 \times \ldots \times \tau_n \mid \mathit{role}(\tau) \mid \mathit{action}(\tau) \mid \mathit{set}(\tau)$

Access-control operations:
doAction$(A(\vec{e}))$, activate$(R(\vec{e}))$, deactivate$(E_v, R(\vec{e}))$, reqCred$(E_s@E_{iss}.p(\vec{x}) \leftarrow c)$

DATALOG

- DATALOG is often associated with database systems with its origins found in that field in the late 1970s.
- More generally, DATALOG is a restricted form of logic programming with variables, predicates, and constants, but without function symbols.
- DATALOG $\subset$ Prolog (syntactically)

DATALOG Rule

$$R_0(t_{0,1},\ldots,t_{0,k0}) :- R_1(t_{1,1},\ldots,t_{1,k1}),\ldots, R_0(t_{n,1},\ldots,t_{n,kn})$$

- $R_0,\ldots,R_n$ are predicate relation symbols
- Each term $t_{i,j}$ is either a constant or a variable
- The formula $R_0(t_{0,1},\ldots,t_{0,k0})$ is called the *head* of the rule and the remainder is called the *body*
- If $n = 0$ then the rule is called a *fact*
- A DATALOG program is a finite set of such rules

DATALOG^C

- The notion of *constraint databases* grew out of research on DATALOG and Constraint Logic Programming.
- The notion of *constraint domains* generalizes the relational model of data by allowing infinite relations that are finitely representable using constraints.
- DATALOG^C is effectively DATALOG with constraints. The term refers to a wide range of specific languages since different constraint domains can be considered.

Constraint Domains

DEF: A *constraint domain* Φ is a 3-tuple $(\Sigma,\mathcal{D},\mathcal{L})$.

- Σ consists of a set of constants and a collection of *k*-ary predicate and function symbols
- $\mathcal{D}$ consists of a set *D* called the universe of the structure, a mapping from each constant to an element in *D*, a mapping from each *k*-ary predicate symbol in Σ to a *k*-ary relation over *D*, and a mapping from each *k*-ary function symbol in Σ to a function from D^k into *D*.
- $\mathcal{L}$ is a class of quantifier-free first-order formulas over Σ

Constraint Domains Examples

- **Equality:** The signature Σ consists of a set of constants and one predicate =. A primitive constraint has the form $x = y$ or $x = c$. DATALOG can be regarded as a specific instance of DATALOG^C with this constraint domain.
- **Order:** The signature Σ has two predicates: = and <. A primitive constraint has the form $x\ \theta\ y$, $x\ \theta\ c$, or $c\ \theta\ x$ where $\theta \in \Sigma$.
- **Linear:** The signature Σ has the function symbols + and * and the predicates {=, ≠, <, >, ≤, ≥}. A primitive constraint has the form $c_1x_1 + \ldots + c_kx_k\ \theta\ b$.

Constraint Database

DEF: Let Φ be a constraint domain.

1. A *constraint k-tuple* is a finite conjunction $\phi_1 \wedge \ldots \wedge \phi_N$ where each ϕ_i is a primitive constraint in Φ.
2. A k-ary *constraint relation* is a finite set $r = \{\psi_1,\ldots,\psi_M\}$, where each ψ_i is a constraint k-tuple over the same variables.
3. The *formula corresponding to* the constraint relation r is the disjunction $\psi_1 \vee \ldots \vee \psi_M$.
4. A *constraint database* is a finite collection of constraint relations.

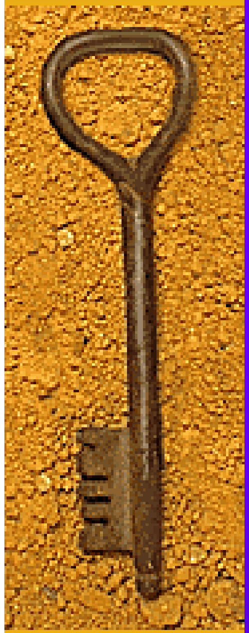
DATALOG^C Rule

$$R_0(t_{0,1},\ldots,t_{0,k0}) \text{ :- } R_1(t_{1,1},\ldots,t_{1,k1}),\ldots,\ R_0(t_{n,1},\ldots,t_{\mathrm{n},kn}), \psi_0$$

- ψ_0 is a constraint in the set of all variables in the rule.
- If $n = 0$ then the rule is called a *constraint fact*
- A constraint rule with n hypotheses may be applied to n constraint facts to produce m facts.
- The process of applying a rule to a set of facts requires a form of quantifier elimination.
- Intuitively the head of the rule holds if the body holds.

Constraint Domains in Trust Management

- In TM languages it is often useful to appeal to constraints from several domain.
- Some useful constraint domains in trust management:
  - **Tree:** Each constant takes the form $<a_1,\ldots,a_k>$, which represents the node for which $a_1,\ldots, a_k$ are the strings on the path from the root to a given node. A primitive constraint is of the form $x = y$ or $x\theta<a_1,\ldots,a_k>$ in which $\theta \in \{=, <, \leq, \prec, \prec=\}$. $x < <a_1,\ldots,a_k>$ means x is a child of the specified node and $x \prec <a_1,\ldots,a_k>$ means x is an ancestor of the specified node.
  - **Range:** Syntactically sugared order domain.
  - **Discrete w/ sets:** Syntactically sugared equality domain.



DATALOG^C Example

- An entity *A* grants *B* permission to connect to machines in the domain “neu.edu” at port 80 with a validity period from t_1 to t_3.

$$\textbf{grantConnect}(A, B, h, p, v) \text{ :- } h \preceq \langle\text{edu,neu}\rangle, p = 80, v \in [t_1,t_3].$$

- An entity *A* grants *B* permission to delegate the same permissions.

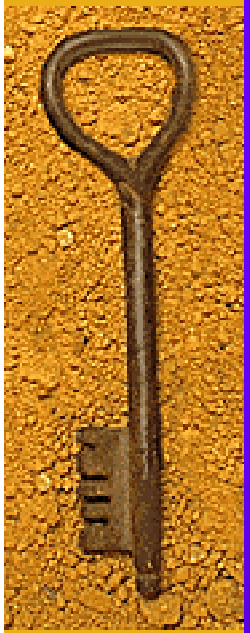
$$\textbf{grantConnect}(A, x, h, p, v) \text{ :- }$$
$$\textbf{grantConnect}(B, x, h, p, v), h \preceq \langle\text{edu,neu}\rangle, p = 80, v \in [t_1,t_3].$$

- An entity *B* grants *D* similar permissions for the ccs subdomain.

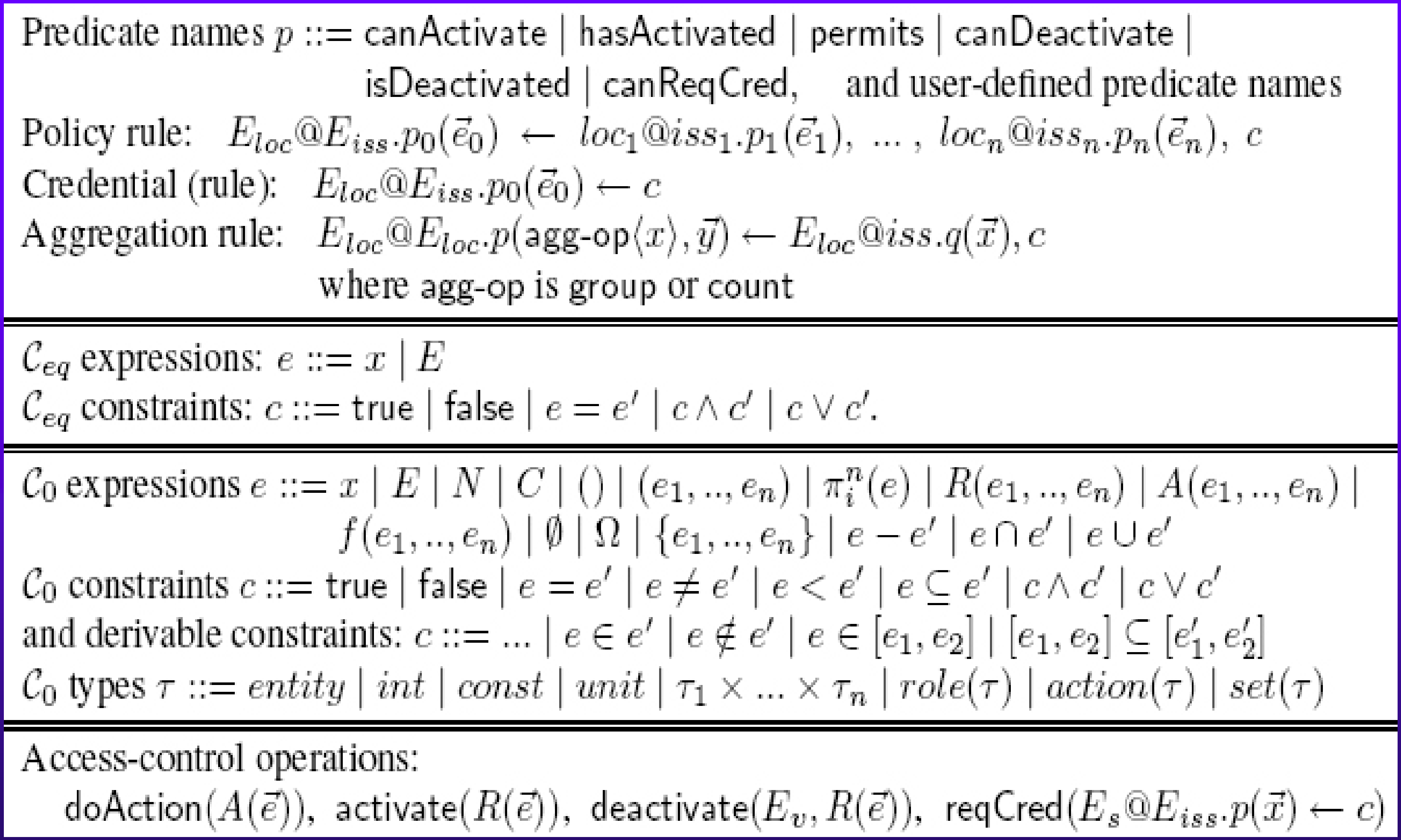
$$\textbf{grantConnect}(B, D, h, p, v) \text{ :- } h \preceq \langle\text{edu,neu,ccs}\rangle, v \in [t_2,t_4].$$

- From the above constraint facts and rules we can derive:

$$\textbf{grantConnect}(A, D, h, p, v) \text{ :- } h \preceq \langle\text{edu,neu,ccs}\rangle, p = 80, v \in [t_2,t_3].$$



Cassandra Syntax (again)



Predicate names p ::= canActivate | hasActivated | permits | canDeactivate | isDeactivated | canReqCred, and user-defined predicate names

Policy rule: $E_{loc}@E_{iss}.p_0(\vec{e}_0) \leftarrow loc_1@iss_1.p_1(\vec{e}_1), \ldots, loc_n@iss_n.p_n(\vec{e}_n), c$

Credential (rule): $E_{loc}@E_{iss}.p_0(\vec{e}_0) \leftarrow c$

Aggregation rule: $E_{loc}@E_{loc}.p(\text{agg-op}\langle x\rangle, \vec{y}) \leftarrow E_{loc}@iss.q(\vec{x}), c$
where agg-op is group or count

$\mathcal{C}_{eq}$ expressions: $e ::= x \mid E$

$\mathcal{C}_{eq}$ constraints: $c ::= \text{true} \mid \text{false} \mid e = e' \mid c \wedge c' \mid c \vee c'$.

$\mathcal{C}_0$ expressions $e ::= x \mid E \mid N \mid C \mid () \mid (e_1, .., e_n) \mid \pi_i^n(e) \mid R(e_1, .., e_n) \mid A(e_1, .., e_n) \mid f(e_1, .., e_n) \mid \emptyset \mid \Omega \mid \{e_1, .., e_n\} \mid e - e' \mid e \cap e' \mid e \cup e'$

$\mathcal{C}_0$ constraints $c ::= \text{true} \mid \text{false} \mid e = e' \mid e \neq e' \mid e < e' \mid e \subseteq e' \mid c \wedge c' \mid c \vee c'$

and derivable constraints: $c ::= \ldots \mid e \in e' \mid e \notin e' \mid e \in [e_1, e_2] \mid [e_1, e_2] \subseteq [e_1', e_2']$

$\mathcal{C}_0$ types $\tau ::= \mathit{entity} \mid \mathit{int} \mid \mathit{const} \mid \mathit{unit} \mid \tau_1 \times \ldots \times \tau_n \mid \mathit{role}(\tau) \mid \mathit{action}(\tau) \mid \mathit{set}(\tau)$

Access-control operations:
doAction$(A(\vec{e}))$, activate$(R(\vec{e}))$, deactivate$(E_v, R(\vec{e}))$, reqCred$(E_s@E_{iss}.p(\vec{x}) \leftarrow c)$

Constraint Domain C_0

- Atomic expressions can be variables, entities, integers, constants of various types, the empty set $\varnothing$ and the universal set Ω.
- Compound expressions can be built from atomic ones recursively: tuples $(e_1,\ldots,e_n)$, tuple projections $\pi^n_i(e)$, roles $R(e_1,\ldots,e_n)$, actions $A(e_1,\ldots,e_n)$, function applications $f(e_1,\ldots,e_n)$, and the set expressions.

Constraint Domain C_0 (cont.)

- Constraints include equalities $e = e'$, inequalities $e \neq e'$, integer orders $e < e'$, set containments $e \subseteq e'$, and conjunctions and disjunctions of those. It also includes constraints that can be defined in terms of the existing ones such as non-membership $e \not\subset e'$ *and* integer ranges $e \in [e_1, e_2]$.
- A type system where types τ are of the form *int*, *entity*, *const*, *unit*, $\tau_1 \times \ldots \times \tau_n$, *role*($\tau$), *action*($\tau$), and *set*($\tau$).

Roles and Actions

- *Roles* and *Actions* are parameterized for higher expressiveness
- For example Clinician(*org*, *spcty*) has parameters for the health organization and the specialty of the clinician
- Cassandra’s notion of role is more general than that typically used
- For example, the Access-denied-by-patient role may indicate that a record item is concealed by the patient

Predicates

There are six special predicates in Cassandra:

1. canActivate(*e*,*r*) expresses the fact that the entity *e* can activate the role *r*

2. hasActivated(*e*,*r*) indicates that *e* has activated *r*

3. permits(*e*,*a*) says the entity *e* can perform action *a*

Predicates (cont.)

4. canDeactivate(e_1,e_2,r) says e_1 can deactivate e_2's activation r. This can cause cascading deactivation
5. isDeactivated(e,r) indicates the deactivation of entity e's role r
6. canReqCred(e_1,e_2.p(e')) says that e_1 is allowed to request credentials issued by e_2 and asserting the predicate p(e') where e' is a vector.

Policy writers can also introduce user-defined predicates.

Rules

- Cassandra extends DATALOG^C's predicates by adding a notion of an issuing entity and a storage location.

$$E_{loc}@E_{iss}.p_0(e_0) \leftarrow loc_1@iss_1.p_1(e_1),\ldots,loc_n@iss_n.p_n(e_n),\ c$$

- p_i are the predicate names
- e_i are (possibly empty) expression tuples
- c is a constraint from some fixed constraint domain
- E_{loc} and E_{iss} are the location and issuer of the rule
- loc_i and iss_i are entities or entity-typed variables

Rules (cont.)

- A policy rule of the form

$$E_{loc}@E_{iss}.p_0(e_0) \leftarrow c$$

  is called a *credential*.

- If it is sent over the network it can be though of as a certificate asserting $p_0(e_0)$, signed and issued by E_{iss}, and belonging to and stored at E_{loc}.
- Usually $E_{loc} = E_{iss}$
- A Cassandra *policy* of an entity E_{loc} is a finite set of rules (including credentials) with location E_{loc}.

Rules (cont.)

- A body predicate

$$B@C.p(e)$$

can refer to a remote location.

- Say A is the local entity. If $B \neq A$ then A will contact B over the network and delegate authority to B to deduce the predicate.
- Before attempting the reduction B will first deduce

$$B@B.\mathrm{canReqCred}(A,C.p(e))$$

Cassandra Example

- Suppose A's policy contains the rules:

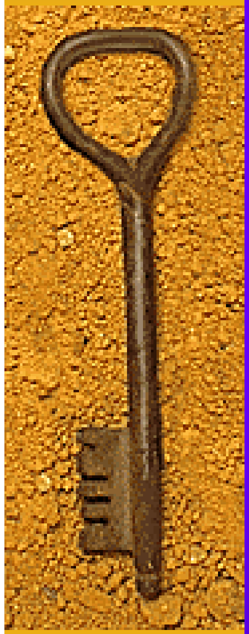
$$R_1 \equiv A@A.\text{likes}(A,x) \leftarrow \text{x@y.likes}(y,x),\ x \neq y$$
$$R_2 \equiv A@B.\text{likes}(B,A) \leftarrow \text{true}$$

- So, say A tries to deduce whether A likes herself:

$$A@y.\text{likes}(y,A) \leftarrow A \neq y$$

- Using R_2 A can deduce:

$$A@A.\text{likes}(A,A)$$



Cassandra Example (cont.)

- Now say *C*'s policy contains the rules:

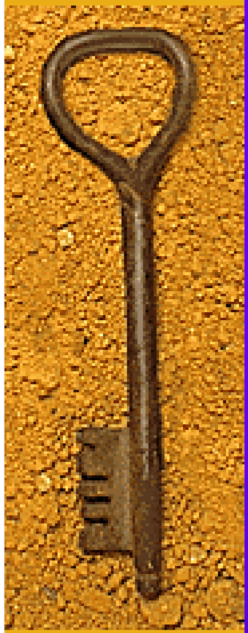
$$R_3 \equiv C@D.\text{likes}(D,C) \leftarrow \text{true}$$

- So, *A* tries to deduce whether she likes *C*. Assuming *A* cannot do this locally, *A* automatically requests from *C*:

$$C@y.\text{likes}(y,C) \leftarrow C \neq y$$

- *C* first must check whether *A* is honor the request:

$$C@C.canReqCred(A,y.likes(y,C)) \leftarrow C \neq y$$



Cassandra Example (cont.)

- *C* first must check whether to honor the request:

$$C@C.canReqCred(A,y.likes(y,C)) \leftarrow C \neq y$$

- The result will either be false or some constraint on the variable *y*. Perhaps *C* cannot reveal information from *E* to *A*, so it returns the constraint $y \neq E$. So *C* tries to deduce:

$$C@y.likes(y,C) \leftarrow C \neq y \wedge y \neq E$$

- *C* deduces this using R_3 and returns *C@D.likes(D,C)* to *A*, which allows *A* to deduce *A@A.likes(A,C)*.

Questions?