Example: Italians Revisited

The italian-vs-english ontology from Slide 522 can be specified via Choice Rules (and implicit) Denials (Line 1: either italian or english allowes):

```
0{ italian(X), english(X) }1 :- person(X),      thing(X).
person(X) :- italian(X),                        thing(X).
person(X) :- english(X),                        thing(X).
1{ lazy(X), latinlover(X) }1 :- italian(X),     thing(X).
italian(X) :- lazy(X),                          thing(X).
italian(X) :- latinlover(X),                    thing(X).
0{ gentleman(X), hooligan(X) }2 :- english(X),  thing(X).
english(X) :- gentleman(X),                     thing(X).
english(X) :- hooligan(X),                      thing(X).
gentleman(X) :- latinlover(X),                  thing(X).
italian(e).
thing(e).
```

[Filename: Datalog/italians-english.s]

- do not query for `?- lazy(e)` , but inspect the stable model(s).
- there is a single (total) stable model where `lazy(e)` holds.

SMODELS: PLANNING

Give a specification P of the workflow including constraints:

- if at least one total stable model S exists, the specification including the constraints is satisfiable.
  S describes the plan that must be followed.
- if several total stable model exist, each one represents a possible execution.
  Different plans mean that choicepoints during the execution exist.
  - they can be decided a priori: add the intended atoms to P:
    - there is at least one stable model, but maybe still several ones.
  - ... or decide them during execution of the workflow (e.g. to be able to react upon external influences).

Planning Example: The Farmer's Puzzle

A farmer, travelling to the market with his dog, a goat, and a cabbage. He has to cross a small river, where a boat can be used.

- When using the boat, he can transport only one item.
- He can cross the river as often as he wants.
- When the dog stays on the same side as the goat, and the farmer is not there, the dog will kill and eat the goat.
- When the goat stays on the same side as the cabbage, and the farmer is not there, the goat will eat the cabbage.

Is it possible for the farmer to bring all items to the other side? If yes, how?

Example (Cont'd)

```
state(1..8).     % estimate the number of necessary states.
side(l).  side(r).
is(farmer,l,1).     % farmer is on the left side in state 1 (with all items)
thing(cabbage).  thing(goat).  thing(dog).
is(X,l,1) :- thing(X).
otherside(X,Y) :- side(X), side(Y), X != Y.
0{ transport(cabbage,N), transport(goat,N), transport(dog,N) }1 :- state(N), not finished(N).
:- transport(X,N), is(farmer,S1,N), is(X,S2,N), S1 != S2,
   thing(X), state(N), side(S1), side(S2).
is(X,S2,M) :- is(X,S1,N), thing(X), otherside(S1,S2), transport(X,N), M = N+1, not finished(N)
   state(N), side(S1).
is(X,S1,M) :- is(X,S1,N), thing(X), not transport(X,N), M = N+1, not finished(N),
   state(N), side(S1).     %% the "frame axiom"
is(farmer,S2,M) :- is(farmer,S1,N), otherside(S1,S2), M = N+1, not finished(N),
   state(N), side(S1).
:- is(cabbage,S,N), is(goat,S,N), not is(farmer,S,N),   state(N), side(S).
:- is(goat,S,N), is(dog,S,N), not is(farmer,S,N),       state(N), side(S).
finished(N) :- is(cabbage,r,N), is(goat,r,N), is(dog,r,N),        state(N).
finished(M) :- finished(N), M = N+1,                              state(N).
:- not finished(8).
```

[Filename: Datalog/farmer.s]

Example (Cont'd)

Two stable models:

1. first carry the goat to the other side (r)
   (it must not be left with the dog or with the cabbage).
2. go back,
3. bring either the dog or the cabbage to the other side,
4. go back with the goat,
5. bring the remaining item to the other side,
6. go back,
7. take the goat and bring it to the right bank again,
8. continue traveling to the market.

$\Rightarrow$ Step 3 is a choicepoint.

SMODELS: FURTHER EXAMPLES AND PUZZLES

- the smodels documentation (Section 6.1) contains
  - the Graph 3-Coloring problem,
  - some logical puzzles.
- Exercise: encode the Fish Puzzle (see Web page) in ASP and let smodels solve it.

"PROVE" $P \models \varphi$

- up to now: answer queries, look for ground atoms.
- It is only possible to prove that $S \models \varphi$ for all *stable* models of P:
- Prove $P \models_S \varphi$ (1st Alternative):
  - generate all stable models $S_1, \ldots, S_n$.
  - for each of them check whether $S_i \models \varphi$ (if not, it is a witness for a counterexample)
- Prove $P \models_S \varphi$ (2nd Alternative):
  - encode $\neg\varphi$ in a program P',
  - run smodels on $P \cup P'$,
  - if there is a stable model, then, $P \not\models_s \varphi$.

EXAMPLE: PARRICIDES IN GREEK MYTHODOLOGY

(from ESWC'07 SPARQL tutorial by Marcelo Arenas et al)

A parricide is a person who killed his/her father.

```
1{ parricide(X), nonParricide(X) }1 :- person(X).
person(iokaste).  nonParricide(iokaste).
hasChild(iokaste, oedipus).
person(oedipus).  parricide(oedipus).
marriedto(oedipus, iokaste).    %% irrelevant, but the reason why he became prominent
hasChild(oedipus, perineikes).
person(perineikes).     %% unknown whether parricide or not.
hasChild(perineikes, thesandros).
person(thesandros). nonParricide(thesandros).
parentOfParricide(X) :- hasChild(X,Y), parricide(Y).
parentOfNonParricide(X) :- hasChild(X,Y), nonParricide(Y).
parentOfParricideGrandparentOfNonParricide(X) :-
   parentOfParricide(X), hasChild(X,Y), parentOfNonParricide(Y).
```

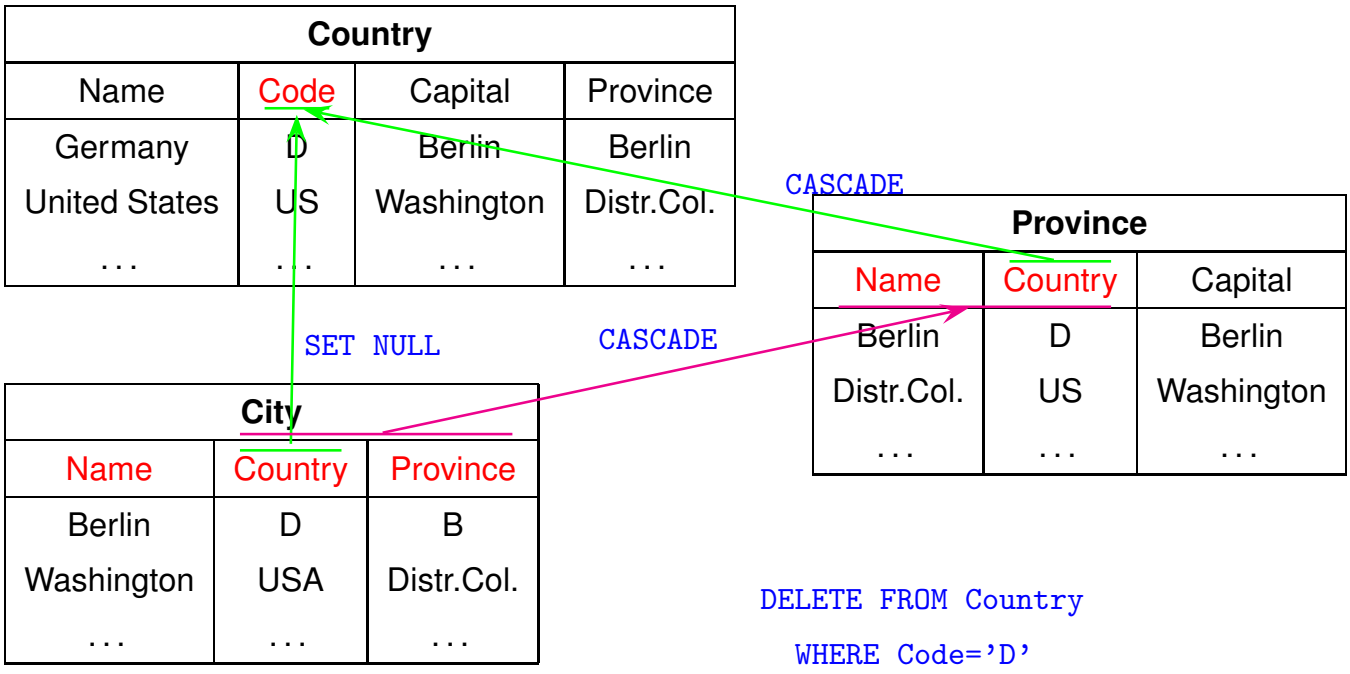
[Filename: Datalog/parricide.s]

Does $P \models_S \exists X$: *parentOfParricideGrandparentOfNonParricide(X)*?

- two possibilities: either Iokaste or Oedipus.

12.2 An Application: Semantics of Referential Actions in SQL

Consider again the problem of ambiguous semantics of referential actions from Slide 230:



SQL Standard

- The SQL Standard gives a hard-to-understand *procedural* specification of referential actions.
- Database Systems implemented only ON DELETE CASCADE (as optional alternative to ON DELETE NO ACTION) for a long time (late 90s)
- Nondeclarative semantics if ON DELETE SET NULL or ON UPDATE CASCADE are implemented.
  e.g. Oracle 11: the most recently defined (more exactly: activated) referential action is executed first.
- combination with transactions and PL/SQL and triggers becomes complex.

Concepts

- Intuitive concept of "Event Condition Action Rules":
  ON DELETE (of referenced tuple) CASCADE (update to referencing tuple(s)
- can be read as a declarative specification how integrity is to be maintained:
  "Whenever the set U of updates includes the deletion of a referenced tuple wrt. referential integrity constraint $S.\bar{B} \rightarrow R.\bar{A}$) the cascaded update of the referencing tuple(s) must also be contained in U. (etc. for other referential actions)"

$\Rightarrow$ Set-oriented characterization of all updates that "complete" a transaction wrt. referential integrity maintenance.

... from a declarative point of view, the semantics should be easy, and it *must* be unambiguous. This lead us to playing around with Datalog (and Statelog) around Easter 1996:

Publications

The following papers are accessible via `http://dblp.uni-trier.de/` (and partially via the DBIS Publications Web pages):

- B. Ludäscher, W. May und J. Reinert. Towards a Logical Semantics for Referential Actions in SQL. In *Proc. Intl. Workshop on Foundations of Models and Languages for Data and Objects: Integrity in Databases (FMLDO'96)*, Dagstuhl Castle, Germany, 1996.
- B. Ludäscher, W. May und G. Lausen. Referential Actions as Logical Rules. In *Proc. ACM Symposium on Principles of Database Systems (PODS'97)*, pp. 217–224, 1997.
- B. Ludäscher und W. May. Referential Actions: From Logical Semantics to Implementation. In *Proc. Intl. Conference on Extending Database Technology (EDBT'98)*, Springer LNCS 1377, pp. 404-418, 1998.
- W. May und B. Ludäscher. Understanding the Global Semantics of Referential Actions using Logic Rules. In *ACM Transactions on Database Systems (TODS)*, 27(4):343–397, 2002.

12.2.1 Straightforward Logical Semantics: Encoding as Rules

- consider only ON DELETE CASCADE/SET NULL:

```
% prov(Country) refs country(code) on delete cascade
del_province(PN,C,PPop,PA,PCap,PCapProv) :- del_country(N,C,Cap,CapP,A,P),
    province(PN,C,PPop,PA,PCap,PCapProv).
% city(Country) refs country(code) on delete set null
upd_city(CN,C,CP,CPop,Long,Lat,El,CN,null,CP,CPop,Long,Lat,El) :-
    del_country(N,C,Cap,CapP,A,P), city(CN,C,CP,CPop,Long,Lat,El).
% city(Country,Province) refs province(name,country) on delete cascade
del_city(CN,C,PN,CPop,Long,Lat,El) :-  del_province(PN,C,PPop,PA,PCap,PCapProv),
    city(CN,C,PN,CPop,Long,Lat,El).
inconsistent :-  del_city(CN,C,CP,CPop,Long,Lat,El),
                 upd_city(CN,C,CP,CPop,Long,Lat,El,CN2,C2,CP2,CPop2,Long2,Lat2,El2).
country("Germany","D","Berlin","Berlin",356910,83536115).
province("Berlin","D",3472009,889,"Berlin","Berlin").
city("Berlin","D","Berlin",3472009,13,52,null).
del_country("Germany","D","Berlin","Berlin",356910,83536115).
% ?- error.
```

[Filename: Datalog/refint.s]

- cascaded updates and “inconsistent” are true.

Deletions: Encoding as Rules

- ON DELETE/UPDATE NO ACTION:
  - if there is a referencing tuple that is not deleted (or modified to reference another parent) in the same transaction, then the update is not allowed, e.g., Reference Organization(City,Country,Province) $\rightarrow$ City(City,Country,Province) ON DELETE NO ACTION ON UPDATE CASCADE

$\Rightarrow$ all potential updates during the transaction must be considered.

- `ext_ACTION` (“external”): the updates issued by the user/by the program.
- `pot_ACTION` (“potential”): all updates issued by the user/by the program or resulting from these by any referential action.

12.2.2 Deletions: Encoding as Rules

- Consider only deletions and ON DELETE CASCADE/NO ACTION.

```
pot_del_country(N,C,Cap,CapP,A,Pop) :- ext_del_country(N,C,Cap,CapP,A,Pop),
        country(N,C,Cap,CapP,A,Pop).
% Province(Country) refs Country(code) on delete cascade
pot_del_province(P,C,PPop,PA,PCap,PCapProv) :- pot_del_country(N,C,Cap,CapP,A,Pop),
        province(P,C,PPop,PA,PCap,PCapProv).
% City(Country) refs Country(code) on delete cascade
pot_del_city(CN,C,P,CPop,Long,Lat,El) :-
    pot_del_country(N,C,Cap,CapP,A,Pop),
    city(CN,C,P,CPop,Long,Lat,El).
% City(Country,Province) refs Province(name,country) on delete cascade
pot_del_city(CN,C,P,CPop,Long,Lat,El) :-
    pot_del_province(P,C,PPop,PA,PCap,PCapProv),
    city(CN,C,P,CPop,Long,Lat,El).
```

[Filename: Datalog/refint1.s]

Deletions: Encoding as Rules

- block deletions/updates if ON UPDATE NO ACTION
- propagate blocking upwards through CASCADES

```
pot_del_organization(O,N,Ci,Co,P,E) :- ext_del_organization(O,N,Ci,Co,P,E),
        organization(O,N,Ci,Co,P,E).
% refs from isMember to Country and Organization: ON DELETE CASCADE.
pot_del_isMember(C,O,T) :- pot_del_country(N,C,Cap,CapP,A,Pop),
        isMember(C,O,T).
pot_del_isMember(C,O,T) :- pot_del_organization(O,N,Ci,Co,P,E),
        isMember(C,O,T).
```

[Filename: Datalog/refint2.s]

Deletions: Encoding as Rules

- Organization(City,Country,Province) → City(City,Country,Province) ON DELETE NO ACTION ON UPDATE CASCADE
- block deletions/updates if ON UPDATE NO ACTION and child tuple remains,
- propagate blocking upwards through CASCADEs.

```
blk_del_city(Ci,Co,P,CPop,Long,Lat,El) :- pot_del_city(Ci,Co,P,CPop,Long,Lat,El),
  organization(O,N,Ci,Co,P,E), rem_organization(O,N,Ci,Co,P,E).
rem_organization(O,N,Ci,Co,P,E) :- organization(O,N,Ci,Co,P,E),
  not del_organization(O,N,Ci,Co,P,E).            %%% del not yet defined
blk_del_province(P,C,PPop,PA,PCap,PCapProv) :- pot_del_city(CN,C,P,CPop,Long,Lat,El),
  blk_del_city(CN,C,P,CPop,Long,Lat,El), pot_del_province(P,C,PPop,PA,PCap,PCapProv).
blk_del_country(N,C,Cap,CapP,A,Pop) :- pot_del_province(P,C,PPop,PA,PCap,PCapProv),
  blk_del_province(P,C,PPop,PA,PCap,PCapProv), pot_del_country(N,C,Cap,CapP,A,Pop).
```

[Filename: Datalog/refint3.s]

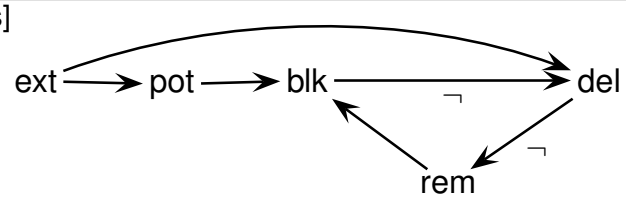
Deletions: Encoding as Rules

- (if any update is blocked do nothing).
- execute (and appropriately cascade) all external updates that are not blocked.

```
del_country(N,C,Cap,CapP,A,Pop) :- ext_del_country(N,C,Cap,CapP,A,Pop),
    country(N,C,Cap,CapP,A,Pop), not blk_del_country(N,C,Cap,CapP,A,Pop).
del_province(P,C,PPop,PA,PCap,PCapProv) :- del_country(N,C,Cap,CapP,A,Pop),
    province(P,C,PPop,PA,PCap,PCapProv).
del_city(CN,C,P,CPop,Long,Lat,El) :- del_country(N,C,Cap,CapP,A,Pop),
    city(CN,C,P,CPop,Long,Lat,El).
del_city(CN,C,P,CPop,Long,Lat,El) :- del_province(P,C,PPop,PA,PCap,PCapProv),
    city(CN,C,P,CPop,Long,Lat,El).
del_isMember(C,O,T) :- del_country(N,C,Cap,CapP,A,P), isMember(C,O,T).
del_organization(O,N,Ci,Co,P,E) :- ext_del_organization(O,N,Ci,Co,P,E),
    organization(O,N,Ci,Co,P,E), not blk_del_organization(O,N,Ci,Co,P,E).
del_isMember(C,O,T) :- del_organization(O,N,Ci,Co,P,E), isMember(C,O,T).
```

[Filename: Datalog/refint4.s]

- Dependency graph:



Deletions: Encoding as Rules

```
country("Germany","D","Berlin","Berlin",356910,83536115).
province("Berlin","D",3472009,889,"Berlin","Berlin").
city("Berlin","D","Berlin",3472009,13,52,null).
ext_del_country("Germany","D","Berlin","Berlin",356910,83536115).
organization("EU","European Union","Brussels","B","Brabant","1992-02-07").
isMember("D","EU","member").
% lparse -n 0 refint1.s refint2.s refint3.s refint4.s refint-del-d.s| smodels
```

[Filename: Datalog/refint-del-d.s]

There is only one stable model (i.e., it coincides with the well-founded model) which is total:

```
pot_del_country("Germany","D","Berlin","Berlin",356910,83536115)
pot_del_province("Berlin","D",3472009,889,"Berlin","Berlin")
pot_del_city("Berlin","D","Berlin",3472009,13,52,null)
pot_del_isMember("D","EU","member")
del_country("Germany","D","Berlin","Berlin",356910,83536115)
del_province("Berlin","D",3472009,889,"Berlin","Berlin")
del_city("Berlin","D","Berlin",3472009,13,52,null)
del_isMember("D","EU","member")
rem_organization("EU","European Union","Brussels","B","Brabant","1992-02-07")
```


Deletions: Encoding as Rules

```
ext_del_country("Belgium","B","Brussels","Brabant",30510,10170241).
country("Belgium","B","Brussels","Brabant",30510,10170241).
city("Brussels","B","Brabant",951580,null,null,null).
province("Brabant","B",2253794,3358,"Brussels","Brabant").
organization("EU","European Union","Brussels","B","Brabant","1992-02-07").
isMember("B","EU","member").
% lparse -n 0 refint1.s refint2.s refint3.s refint4.s refint-del-b1.s| smodels
```

[Filename: Datalog/refint-del-b1.s]

Again, a total unique stable model = well-founded model – it contains blockings:

```
ext_del_country("Belgium","B","Brussels","Brabant",30510,10170241)
pot_del_country("Belgium","B","Brussels","Brabant",30510,10170241)
pot_del_isMember("B","EU","member")
pot_del_province("Brabant","B",2253794,3358,"Brussels","Brabant")
pot_del_city("Brussels","B","Brabant",951580,null,null,null)
rem_organization("EU","European Union","Brussels","B","Brabant","1992-02-07")   <<<<
blk_del_city("Brussels","B","Brabant",951580,null,null,null)                    <<<<
blk_del_province("Brabant","B",2253794,3358,"Brussels","Brabant")               <<<<
blk_del_country("Belgium","B","Brussels","Brabant",30510,10170241)              <<<<
```

Deletions: Encoding as Rules

```
ext_del_country("Belgium","B","Brussels","Brabant",30510,10170241).
ext_del_organization("EU","European Union","Brussels","B","Brabant","1992-02-07").
country("Belgium","B","Brussels","Brabant",30510,10170241).
city("Brussels","B","Brabant",951580,null,null,null).
province("Brabant","B",2253794,3358,"Brussels","Brabant").
organization("EU","European Union","Brussels","B","Brabant","1992-02-07").
isMember("B","EU","member").
% lparse -n 0 refint1.s refint2.s refint3.s refint4.s refint-del-b2.s| smodels
```

[Filename: Datalog/refint-del-b2.s]

Again, a total unique stable model = well-founded model:

```
ext_del_country("Belgium","B","Brussels","Brabant",30510,10170241)
ext_del_organization("EU","European Union","Brussels","B","Brabant","1992-02-07")
pot_del_ [...]
del_country("Belgium","B","Brussels","Brabant",30510,10170241)
del_city("Brussels","B","Brabant",951580,null,null,null)
del_province("Brabant","B",2253794,3358,"Brussels","Brabant")
del_isMember("B","EU","member")
del_organization("EU","European Union","Brussels","B","Brabant","1992-02-07")
```


Deletions: Encoding as Rules

- Change the reference from City to Province from CASCADE to NO ACTION:
  (in refint1.s and refint4.s)

```
pot_del_country(N,C,Cap,CapP,A,Pop) :- ext_del_country(N,C,Cap,CapP,A,Pop),
        country(N,C,Cap,CapP,A,Pop).
% Province(Country) refs Country(code) on delete cascade
pot_del_province(P,C,PPop,PA,PCap,PCapProv) :- pot_del_country(N,C,Cap,CapP,A,Pop),
        province(P,C,PPop,PA,PCap,PCapProv).
% City(Country) refs Country(code) on delete cascade
pot_del_city(CN,C,P,CPop,Long,Lat,El) :-
    pot_del_country(N,C,Cap,CapP,A,Pop),
    city(CN,C,P,CPop,Long,Lat,El).
% City(Country,Province) refs Province(name,country) on delete no action <<<<<<<<
blk_del_province(P,C,PPop,PA,PCap,PCapProv) :-
  pot_del_province(P,C,PPop,PA,PCap,PCapProv),
  city(CN,C,P,CPop,Long,Lat,El), rem_city(CN,C,P,CPop,Long,Lat,El).
rem_city(CN,C,P,CPop,Long,Lat,El) :- city(CN,C,P,CPop,Long,Lat,El),
                                not del_city(CN,C,P,CPop,Long,Lat,El).
```

[Filename: Datalog/refint1b.s]

Deletions: Encoding as Rules

```
del_country(N,C,Cap,CapP,A,Pop) :- ext_del_country(N,C,Cap,CapP,A,Pop),
    country(N,C,Cap,CapP,A,Pop), not blk_del_country(N,C,Cap,CapP,A,Pop).
del_province(P,C,PPop,PA,PCap,PCapProv) :- pot_del_country(N,C,Cap,CapP,A,Pop),
    del_country(N,C,Cap,CapP,A,Pop), province(P,C,PPop,PA,PCap,PCapProv).
del_city(CN,C,P,CPop,Long,Lat,El) :- pot_del_country(N,C,Cap,CapP,A,Pop),
    del_country(N,C,Cap,CapP,A,Pop), city(CN,C,P,CPop,Long,Lat,El).
del_isMember(C,O,T) :- pot_del_country(N,C,Cap,CapP,A,P),
    del_country(N,C,Cap,CapP,A,P), isMember(C,O,T).
del_organization(O,N,Ci,Co,P,E) :- ext_del_organization(O,N,Ci,Co,P,E),
    organization(O,N,Ci,Co,P,E), not blk_del_organization(O,N,Ci,Co,P,E).
del_isMember(C,O,T) :- del_organization(O,N,Ci,Co,P,E), isMember(C,O,T).
```

[Filename: Datalog/refint4b.s]

- using pot_del is sometimes redundant, and only required for providing smodels with a domain predicate.

Deletions: The Resulting Models

```
% lparse -n 0 refint1b.s refint2.s refint3.s refint4b.s refint-del-d.s| smodels
```

- 3 stable models, 2 of them total, one partial.

```
ext_del_country("Germany","D","Berlin","Berlin",356910,83536115)
pot_del_country("Germany","D","Berlin","Berlin",356910,83536115)
pot_del_province("Berlin","D",3472009,889,"Berlin","Berlin")
pot_del_city("Berlin","D","Berlin",3472009,13,52,null)
pot_del_isMember("D","EU","member")
del_province("Berlin","D",3472009,889,"Berlin","Berlin")
del_country("Germany","D","Berlin","Berlin",356910,83536115)
del_city("Berlin","D","Berlin",3472009,13,52,null)
del_isMember("D","EU","member")
```

```
ext_del_country("Germany","D","Berlin","Berlin",356910,83536115)
pot_del_country("Germany","D","Berlin","Berlin",356910,83536115)
pot_del_[...]
blk_del_country("Germany","D","Berlin","Berlin",356910,83536115)
blk_del_province("Berlin","D",3472009,889,"Berlin","Berlin")
rem_city("Berlin","D","Berlin",3472009,13,52,null)
```

- in the partial stable model (=the well-founded model), all `pot_del` are true, all `blk_del` and all `del` are undefined.

Deletions: Interpreting the Resulting Models

3 stable models:

1. the well-founded, partial one:
   all `pot_del` are true, the `blk_del` and `del` are undefined.

- two total stable ones:
  2. all `pot_del` are true, the `del` are true, and the `blk_del` are false (since the reason for `blk_del_prov(...)` is deleted).
  3. all `pot_del` are true, the `blk_del` are true, and the `del` are false (since the deletion of the province is not allowed due to blocking).

Application-Specific Priorities (cf. EDBT paper)

- the "intended" one is the stable model (2) that gives priority to deletions against blockings.
- including game-theoretic interpretation:
  - "del_p($\bar{x}$)" is won if deletion is possible
  - counter-moves: "how?" and "what about this referencing tuple q($\bar{y}$)",
  - justification "cascaded from "del_r($\bar{z}$)" and "claim: del_q($\bar{y}$)",
  - infinite games (cycling around via deleted blocking tuples) are won for the deleter.
  - the well-founded model can be used $\rightarrow$ polynomial.

Deletions: Encoding as Rules

- Sets of (external) delete requests are monotonic:
  Let $del(D)$ denote the actual set of (cascaded) deletions on the database.
- $del(D_1 \cup D_2) = del(D_1) \cup del(D_2)$,
- If D_1 and D_2 are admissible (wrt. NO ACTION), then $D_1 \cup D_2$ is admissible
  (but not vice versa, recall the "Belgium" example – $D_1 \cup D_2$ is admissible even though D_1 alone is not admissible).
- with CASCADE and SET NULL, conflicts can arise;
- with DELETE+CASCADE/SET NULL and UPDATE even more conflicts can arise.

12.2.3 Updates: Encoding as Rules

- with updates, the modeling is more involved:
  - updates can create new foreign keys,
  - usually such updates are cascaded from the appropriate parent,
  - but overlapping FKs induce interferences with other K/FKs.
  - Consider `S.X→R.Y ON UPDATE NO ACTION` and $r(a), r(b)$ and $s(a)$.
    A transaction that modifies $r(a) \to r(c)$ and $s(a) \to s(b)$ is admissible.

  $\Rightarrow$ not only tuple-based, but key-foreign-key-based.
  - different cascaded modifications can be applied at the same time to a tuple.
- Recall:
  - SET NULL cannot create new foreign keys – null values cannot violate any SQL integrity constraint (except NOT NULL).
  - actual conflicts in a "diamond" from a single update can only result from CASCADE and SET NULL since then *different* changes are applied.
- the approach must cover the general "worst case", not only "intuitive" cases.

Updates: Encoding as Rules

- for every key and foreign key:
  pot_chg_R_K_1...K_k(X_1,...,X_n,Y_1,...,Y_k) and chg_R_K_1...K_k(X_1,...,X_n,Y_1,...,Y_k)
  (change f/key attributes $K_1, \ldots, K_k$ of tuple $R(X_1, \ldots, X_n)$ to $Y_1, \ldots, Y_k$).
- propagation for every FK/K reference (ON UPDATE CASCADE):
  pot_prp_R_P_R_C_F_1...F_k(X_1,...,X_n,Y_1,...,Y_k)
  (propagate update from R_P's keys to FK $F_1, \ldots, F_k$ of tuple $R_C(X_1, \ldots, X_n)$).
- user updates: projection to the keys and foreign keys $(K_1, ..., K_k)$ of R:
  pot_prp_ext_R_K_1...K_k(X_1,...X_n,Yi_1,...,Yi_k) :-
  ext_mod_R(X_1,...,X_n,Y_1,...,Y_n), (Yi_1,...,Yi_k) != (Xi_1,...,Xi_k).
- collect propagated changes to keys/foreign keys (overlappings!):
  Simplified pattern: Consider two "incoming" propagations from $R_{P_1}.A \to R_c.K_1$ and $R_{P_1}.B \to R_C.K_2$ concern the key (K_1, K_2) of R_C:
  pot_chg_R_C_K_1,K_2(X_1,...,X_n,Y_1,Y_2) :-
  pot_prp_R_{P_1}_R_C_K_1(X_1,...,X_n,Y_1), pot_prp_R_{P_2}_R_C_K_2(X_1,...,X_n,Y_2),
  (Y_1,Y_2) != (X_{K_1},X_{K_2}).
  (for the fully general rule see (CH_1) in the TODS paper)

Updates: Encoding as Rules

Changes of Primary Keys $R_P(K_1, \ldots, K_k)$ are then handled according to the referential actions of their "child tuples"

- $R_C(F_1, \ldots, F_k)$ REFERENCES $R_P(K_1, \ldots, K_k)$ ON UPDATE NO ACTION:
  blk_chg_R_P_K_1,...,K_k(X_1,...,X_n,Y_1,...,Y_k) :-
  pot_chg_R_P_K_1,...,K_k(X_1,...,X_n,Y_1,...,Y_k), rem_refd_R_P_R_C_F_1...F_k(Y_1,...,Y_k).
- $R_C(F_1, \ldots, F_k)$ REFERENCES $R_P(K_1, \ldots, K_k)$ ON UPDATE CASCADE:
  pot_prp_R_P_R_C_F_1...F_k(Z_1,...,Z_n,Y_1,...,Y_k) :-
  pot_chg_R_P_K_1,...,K_k(X_1,...,X_n,Y_1,...,Y_k),
  π[F_1,...,F_k](Z_1,...,Z_n) = π[K_1,...,K_k](X_1,...,X_n).
  and (block propagation if change of child is blocked)
  blk_prp_R_P_R_C_F_1...F_k(X_1,...,X_n,Y_1,...,Y_k) :-
  pot_prp_R_P_R_C_F_1...F_k(X_1,...,X_n,Y_1,...,Y_k),
  blk_chg_R_C_F_1,...,K_k(X_1,...,X_n,Y_1,...,Y_k)
  and (block parent change if propagation to some child is blocked)
  blk_chg_R_P_K_1,...,K_k(X_1,...,X_n,Y_1,...,Y_k),
  pot_chg_R_P_K_1,...,K_k(X_1,...,X_n,Y_1,...,Y_k),
  blk_prp_R_P_R_C_F_1...F_k(Z_1,...,Z_n,Y_1,...,Y_k),
  π[F_1,...,F_k](Z_1,...,Z_n) = π[K_1,...,K_k](X_1,...,X_n).