

Model Based Testing of Data Constraints

Nicolae Paladi

Thomas Arts



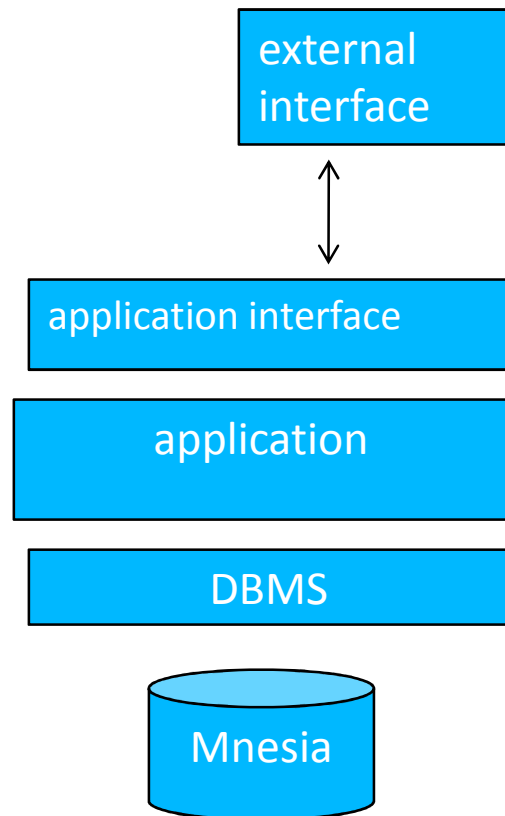
Problem

Assume

- An ER diagram and list of constraints on data
- A large Mnesia database implementing the above
- An application interface to access the database (reading/writing)

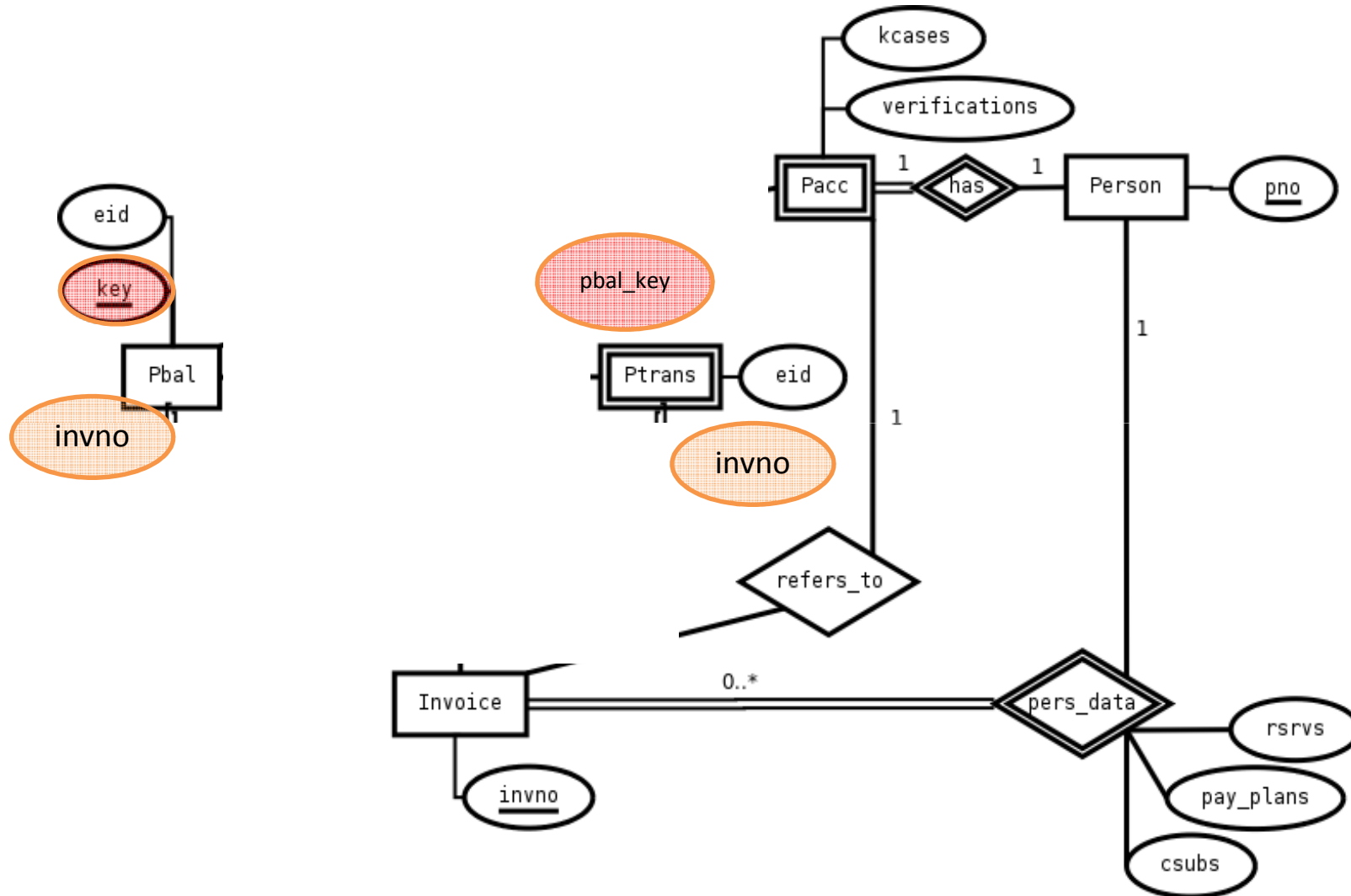
How can we be sure that the application respects the entity-relationship structure and the constraints on data?

Problem

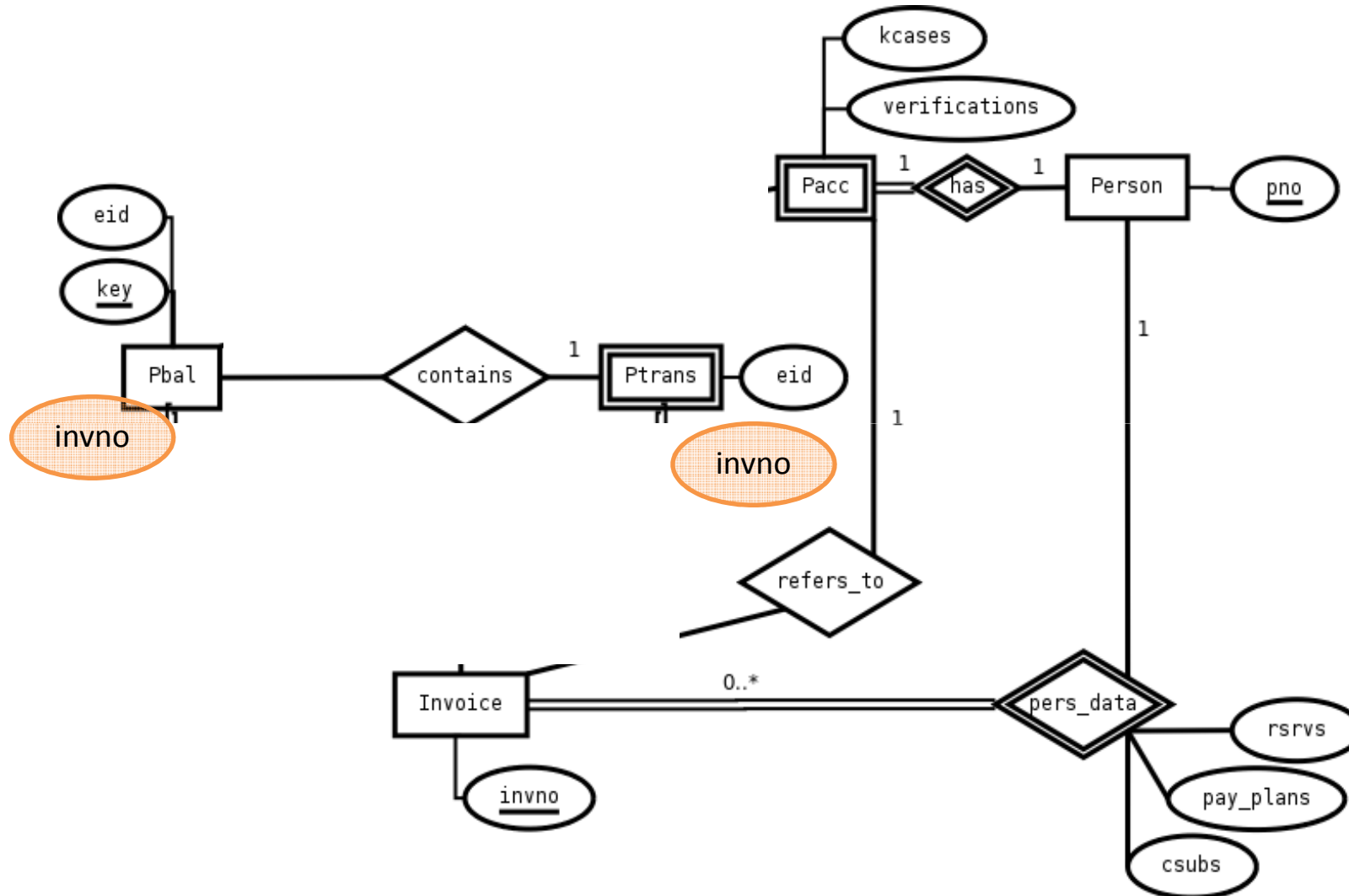


- The DBMS is normally trusted, so no need to test it
- What if referential integrity is not enforced by the DBMS? The constraints implemented in the business logic are both obscure and spread throughout the code

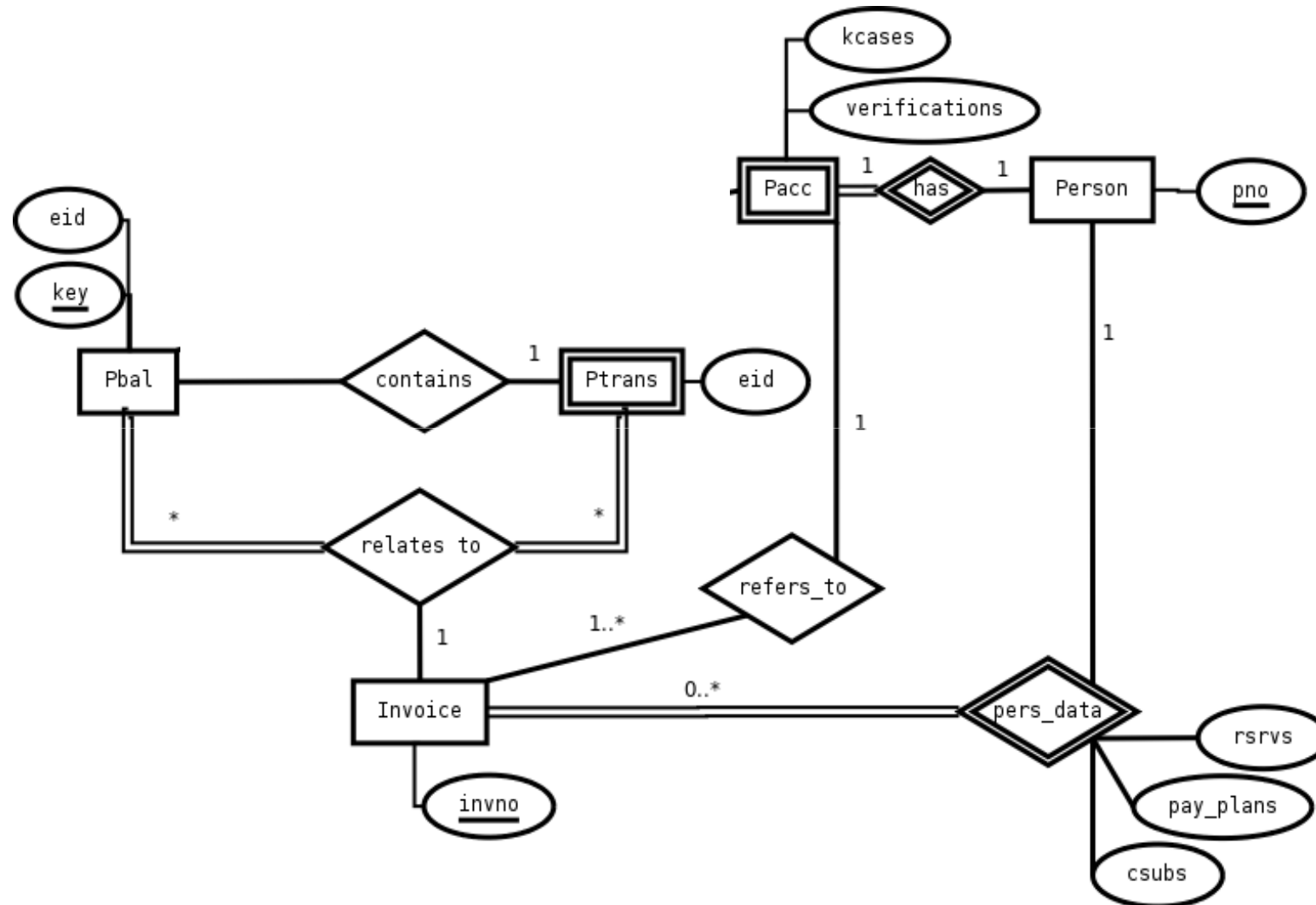
E/R diagram



E/R diagram



E/R diagram



Relations and constraints

In the usual case of lacking specification, reverse engineering is necessary to identify and **SPECIFY** the constraints.

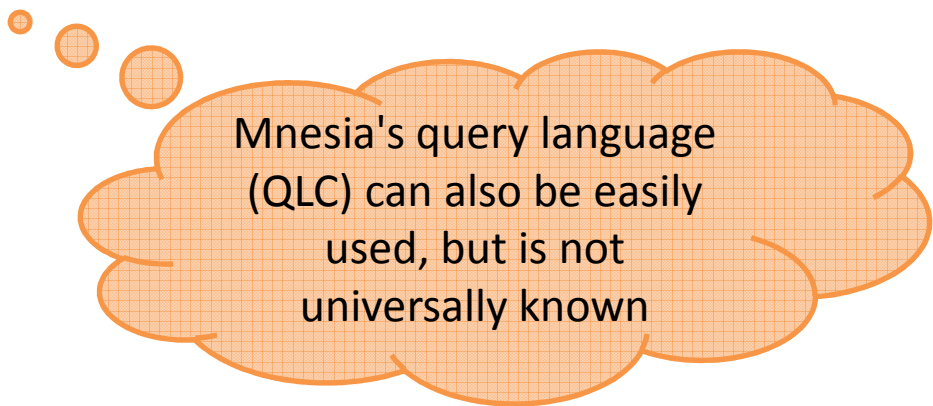
SQL is a suitable language to specify the constraints in.

Specification determines what to **test** for!

Relations and constraints

In the usual case of lacking specification, reverse engineering is necessary to identify and **SPECIFY** the constraints.

SQL is a suitable language to specify the constraints in.



Mnesia's query language (QLC) can also be easily used, but is not universally known

Specification SQL

```
SELECT `ptrans`.`ano`  
FROM ptrans, pbal  
WHERE  
  ((`ptrans`.`pbal key` = `pbal`.`key`)  
AND NOT  
  (`ptrans`.`invno` = `pbal`.`invno` ))
```

OK if this query
results in empty set



Invariants

SQL query can be formulated as QLC invariants

invariant pbal() ->

```
Q = qlc:q([Pb#pbal.ano | |
          Pb <- mnesia:table(pbal),
          PTrans <- mnesia:table(ptrans),
          Pb#pbal.key == PTrans#ptrans.pbal_key,
          Pb#pbal.invno /= Inv#ptrans.invno]),
```

```
{atomic, Response} =
```

```
  mnesia:transaction(fun() -> qlc:e(Q) end),
```

```
Response == [].
```

Test method

(Castro & Arts 2009)

General test idea:

- Check the invariants on the database
- Call the interface functions under test for a random number of times
- Check the invariants on the database

If invariants hold, constraints are not violated

Easy?

Model

- In order to test arbitrary sequences of interface functions a QuickCheck state machine model is defined.
- The state of the model only contains information necessary to generate valid sequences.
- The state of the system is checked by invariants.

Model

We need generators for the records used as arguments when calling the interface functions under test:

```
item() ->
  #item{artno = nat(),
        description = list(char()),
        flags = 0,
        discount = nat(),
        quantity = quantity()}.
```

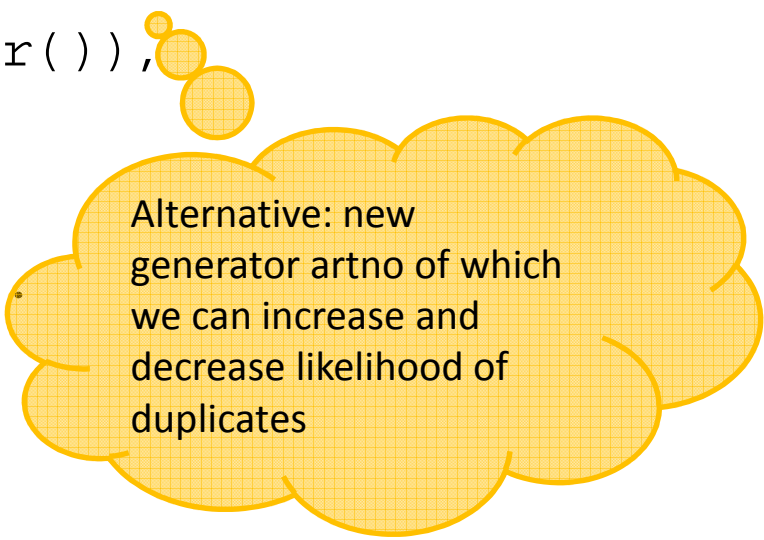
The generator `nat()` gives rather small values and hence good possibility to generate sequences that access earlier created items.

Model

We need generators for the records used as arguments when calling the interface functions under test:

```
item() ->
#item{artno = nat(),
      description = list(char()),
      flags = 0,
      discount = nat(),
      quantity = quantity()}.

```



Alternative: new generator artno of which we can increase and decrease likelihood of duplicates

Model

Another generator example:

either just a few or very many

```
quantity() ->  
  ?LET({N,T,I},{nat(), choose(0,1), laregint()} ,  
      N + T*abs(I)).
```

Model

We need to call interface functions. For example, checking whether there is an active reservation.

Some values need to make sense, some can be arbitrary random. Those that need to make sense are later on guided by the state machine model.



interface function

```
estore_server:handler('undefined',  
  {call, activate_reservation,  
    [Reservation, Items, Pno, (...)]})},
```


Model

Each interface functions is embedded in a local version:

```
activate_reservation(Reservation, Items, Pno) ->
  Result =
    estore_server:handler('undefined',
      {call, activate_reservation,
        [Reservation, Items, Pno, (...)]}),
  Person = person:read_d(Pno),
  Blacklisted = (Person#person.blacklisted == 1),
  case Result of
    {false, {response, [{array, ["no risk", Invno]}}}
      when not Blacklisted -> Result;
    {false, {response, {fault, -4, "blocked"}}}
      when Blacklisted -> Result;
    _ -> exit(unexpected_value)
  end.
```

Model

Each interface functions is embedded in a local version:

```
activate_reservation(Reservation,  
  Result =  
    estore_server:handler('undefined',  
      {call, activate_reservation,  
        [Reservation, Items, Pno, (...)]}),  
    Person = person:read_d(Pno),  
    Blacklisted = (Person#person.blacklisted == 1),  
    case Result of  
      {false, {response, [{array, ["no risk", Invno]}}}   
        when not Blacklisted -> Result;  
      {false, {response, {fault, -4, "blocked"}}}   
        when Blacklisted -> Result;  
      _ -> exit(unexpected_value)  
    end.
```

return Result if
business logic is
not violated.
*Checks simple
relationship*

Positive and
Negative Test
in one

Commands generator

```
command(S) ->
  frequency(
    [{3, {call, ?MODULE, delete_invoice,
          [elements(S#state.invoices)]}}
     || S#state.invoices /= []] ++
    [{3, {call, ?MODULE, add_pbals,
          [elements(S#state.invoices), elements(S#state.pnos)]}}
     || S#state.invoices /= [], S#state.pnos /= []] ++
    [{5, {call, ?MODULE, activate_reservation,
          [reserve(), list(item())]}}] ++
    [{1, {call, ?MODULE, deactivate,
          [elements(S#state.invoices)]}}
     || S#state.invoices /= []] ++
    [{5, {call, ?MODULE, add_kcase, [
          elements(S#state.invoices), choose(2,100),
          elements([2,3])]}}
     || S#state.invoices /= []]).
```

State transitions

In the **state of the model** we save all **keys** needed in consecutive operations.

```
initial_state() ->
  #state{invoices = [], kcases = [],
        pnos = [], pbals = [], deleted = [],
        blacklisted = [], pay_plans = [], rejecte = []}.
```

We save the state variables each time we invoke a command created by the command generator, e.g.:

```
next_state(S, _, {call, _, activate_reservation,
                  [Ocr, _Items, Pno]}) ->
  S#state{invoices = [Ocr | S#state.invoices],
         pnos = [Pno | S#state.pnos]};
```

RESULTS

- 43 of 87 tables considered
- ER diagram with 23 entities, 36 relations and 250 attributes
- 24 constraints formalized, of which 8 were irrelevant or not true (gained system knowledge)
- Two constraints revealed errors when tested with QuickCheck

Quis custodiet ipsos custodies?

Several constraints passed very many tests.

- How do we know our specification makes sense?
- We used Fault injection (both compile time and runtime injections) to see if the QuickCheck model would identify the injected fault.

Quis custodiet ipsos custodies?

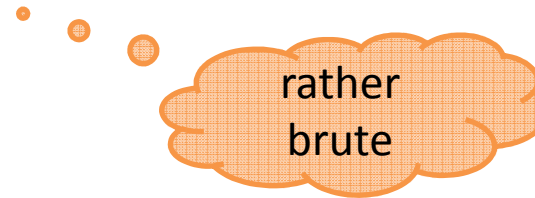
Example constraint:

```
SELECT invoice.pno
FROM invoice, estore_data
WHERE (invoice.eid = estore_data.eid
      AND NOT (invoice.pno in
              estore_data.customers))
```

Whenever a new customer makes a purchase in the estore, s/he is added to the list of customers in the estore_data table of the corresponding estore.

Quis custodiet ipsos custodies?

Injected fault: change source code such that `estore_data` is emptied each time a new invoice is added.



QuickCheck detected the fault and gave a short sequence leading to it.

.... the existing traditional regression test suite did not notice the fault but this fault would be noticed immediately in operation

Summary

In order to test database applications one can use the following method:

- Reverse engineer the database to obtain an ER model.
- Analyze the ER model to determine initial data constraints.
- Analyze the source code to identify other business logic constraints.
- Verify the obtained constraints with the developers
- Create local version of interface functions
- Design state machine model
- Run QuickCheck

Conclusions

- Identification and validation of data constraints often done informally; without tools ER models and constraints in comments are quickly outdated.
- In our case: Infrastructure setup accounted for roughly 20% of the effort, compared to 80% spent on studying the system and elicitation of constraints.
- Robust and versatile test value generation mechanisms provided by QuickCheck allow to identify failing constraints even in intensively used, well tested systems.
- QuickCheck specification is comprehensive documentation of constraints.

Disclaimer: The code snippets used in the paper have been altered and do not represent the actual code base used within the project.