Improving software development using Erlang/OTP

a case study

Laura M. Castro Souto

MADS Group – Universidade da Coruña

June 10th, 2010

Outline

1. Introduction
2. Case study
3. Functional software development methodology
   - From requirements to analysis and design
   - Implementation of a paradigm shift
   - Ensuring functionality and quality through testing
4. Conclusions

1. Introduction
2. Case study
3. Functional software development methodology
   - From requirements to analysis and design
   - Implementation of a paradigm shift
   - Ensuring functionality and quality through testing
4. Conclusions

The questions

- How can we build **better software**. . .
  - when the domain is extremely **complex**?
  - when a lot of **expert knowledge** is required?
  - when we want robust, distributed, **fault-tolerant** systems?
  - when we look for versatile, **flexible**, adaptable applications?

The questions

- How can we build **better software**. . .
  - when the domain is extremely **complex**?
  - when a lot of **expert knowledge** is required?
  - when we want robust, distributed, **fault-tolerant** systems?
  - when we look for versatile, **flexible**, adaptable applications?

. . . using **functional technology**?

Our answer

A **declarative paradigm**-based software development methodology can achieve **significant improvement** by means of **quality assurance** methods

A declarative approximation is:

- more **suitable** to address and solve real-world problems
- **compatible** with traditional analysis and design techniques
- **powerful** to improve product quality

Case study

Advanced Risk Management Information System:

Tracking Insurances, Claims, and Exposures

Armistice

- A **complex management application** for a prominent field
- At the time (2002), **no alternatives** from clients' perspective
  - Today, still no comparable product in the market
- Specific and risky, client company did not have a R & D
  - Agreement with research group at local University

Project requirements

- Modelling and management of organisation resources
- Modelling and management of potential risks
- Modelling and management of contracted insurance policies
  - Including modelling and management of policy clauses
- Management of claims for accidents
  - Selection of the most suitable warranty

    (help decision support system)

Project information

- Client-server architecture
  - Multiplatform lightweight Java client
  - **Server completely developed using Erlang/OTP**
- Started in 2002, first on-site deployment in 2005
  - Under maintenance since 2008
- Development took around 200 person-months
  - Up to five developers (three of them full-time)
- Maintenance nowadays takes around 500 person-hours/year
- Total code size (server + client) $\sim (83000 + 66000)$ LOC

From requirements to analysis and design

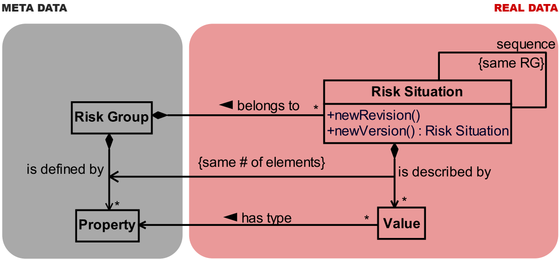
Requirements elicitation:

"*Risk situations have a number of specific properties, which may vary with time.*"

- Implication of **experts** is essential
- Structured and unstructured **interviews** provide good results
- **Multidisciplinary teams** are more productive

From requirements to analysis and design

- O.O. analysis and **standard UML** favour communication
- **Design patterns** help to avoid known pitfalls



From requirements to analysis and design

- O.O. analysis and **standard UML** favour communication
- **Design patterns** help to avoid known pitfalls
- **Model formalisation** as **declarative statements**
  - Provides **early validation**
  - Close to functional **implementation**, re-usable for **testing**
  - Valuable for **traceability**

Implementation of a paradigm shift

Functionality implementation:

"*Determination of policy relevance in the event of an accident.*"

1. **No conflict** between **object-oriented** analysis and design, and implementation approach
2. **High-level** algorithm description eases the **implementation** task, **improving efficiency**

Object orientation vs. declarative paradigm

The **object-oriented** perspective:

- is **closer to** the way we perceive **real entities**
- does **not** perform **well** when describing **behavioural details**
- grants **stability** of actors and interfaces

The **declarative** perspective:

- is **closer to** the way we specify **tasks and activities**
- does **not** work comfortably at the **big scale** of things
- provides **expressive** ways of implementing algorithms

Objects in Erlang/OTP

Objects as **data structures**

Objects as **processes**

Objects in Erlang/OTP

Objects as **data structures**

- ✓ simple
- ✓ immutable
- ✓ coarse grain concurrency
- ✓ low resource usage

Objects as **processes**

Objects in Erlang/OTP

Objects as **data structures**

Objects as **processes**

- ✓ secure encapsulation
- ✓ ‘mutable’
- ✓ fine grain concurrency
- ✓ more resource consuming

Objects in Erlang/OTP

Objects as **data structures** for **short life, coarse concurrency**

- *business objects (risk objects, risk groups, risks, policies, accidents,...), data types, etc.*

Objects as **processes** for **long life, fine-grain concurrency**

- *data storage connection pool, user session manager, task server, file logger, configuration server, etc.*

Both are conveniently used in our study case ARMISTICE.

Implementation of real-world behaviour

"*Determination of policy relevance in the event of an accident.*"

In Erlang syntax:

```
Restriction = {hazard,  Hazard}
            | {formula, {Name, Value}}
            | {string,  Nuance}
            | {all, [Restriction]}
            | {any, [Restriction]}        | ...
```

Implementation of real-world behaviour

```
relevant_policy(Hazard, PolicyClauses) ->
  [ Clause || Clause <- PolicyClauses,
             relevant(Hazard, Clause) =/= false ].
```

```
relevant(H, {hazard, H})       -> true;
relevant(H, {hazard, NH})      -> false;
relevant(H, {string, Nuance}) -> Nuance;
relevant(H, {all, Clauses})    ->
        lists:foldl(fun(A, B) -> and(A, B) end, true,
                    [ relevant(H, C) || C <- Clauses ]);
relevant(H, {any, Clauses})    ->
        lists:foldl(fun(A, B) -> or(A, B) end, false,
                    [ relevant(H, C) || C <- Clauses ]);
...
```

Ensuring functionality and quality through testing

Testing

"There is a problem with ARMISTICE..."

Three types of **testing scenarios** (in theory):

1. Conformance of components to specification (**unit testing**)
2. Appropriate interaction of components (**integration testing**)
3. System behaviour in accordance with requirements (**validation**)

Ensuring functionality and quality through testing

Testing

"There is a problem with ARMISTICE..."

Three types of **testing scenarios** (in the real world):

1. **By-hand developer** ad-hoc tests on own code
2. **By-hand developer** functionality testing
3. **On-site user** validation

Unit testing of data types using properties

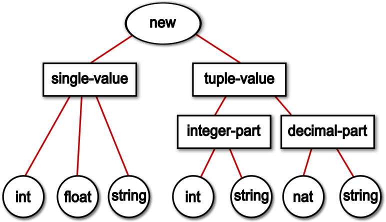
Unit testing

"Does the decimal custom data type conform to its specification?"

Sometimes a **custom implementation of a data type** is in place (i.e., ARMISTICE currency data type)

- *Requirement*: support for multiple currencies, exchange rates
- *Requirement*: 16 decimal digits precision
- *Technical requirement*: uniform marshalling/unmarshalling

Unit testing of data types using properties



Initial strategy:

- Use an automatic testing tool to **generate random decimal values**: QuickCheck

```
decimal() ->
  ?LET(Tuple, {int(), nat()}, decimal:new(Tuple)).
```

Unit testing of data types using properties

```
prop_sum_comm() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
          decimal:sum(D1, D2) == decimal:sum(D2, D1)).
```

Thousands of randomly generated **test cases will pass** for this kind of properties

Unit testing of data types using properties

```
prop_sum_comm() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
          decimal:sum(D1, D2) == decimal:sum(D2, D1)).
```

Thousands of randomly generated **test cases will pass** for this kind of properties but...

- **which other** properties do we add?
- **when** do we have **sufficiently** many of them?

Unit testing of data types using properties

Define a **model** for the data type:

$$
\begin{array}{rcl}
[\![sum(d_i, d_j)]\!] & \equiv & [\![d_i]\!] + [\![d_j]\!] \\
[\![subs(d_i, d_j)]\!] & \equiv & [\![d_i]\!] - [\![d_j]\!] \\
[\![mult(d_i, d_j)]\!] & \equiv & [\![d_i]\!] * [\![d_j]\!] \\
[\![divs(d_i, d_j)]\!] & \equiv & [\![d_i]\!] \;/\; [\![d_j]\!] \\
 & \ldots &
\end{array}
$$

```
decimal_model(Decimal) -> decimal:get_value(Decimal).
```

Erlang/C floating point implementation (IEEE 754-1985 standard)

Unit testing of data types using properties

```
prop_sum() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
    decimal_model(decimal:sum(D1, D2)) ==
      decimal_model(D1) + decimal_model(D2)).
```

Unit testing of data types using properties

```
prop_sum() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
    decimal_model(decimal:sum(D1, D2)) ==
       decimal_model(D1) + decimal_model(D2)).
```

Problem: errors show internal representation

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
....Failed! After 5 tests.
{{decimal,1000000000000000}, {decimal,11000000000000000}}
false
```

Unit testing of data types using properties

Use **symbolic data structures** in test generation:

```
decimal() ->
  ?LET(Tuple, {int(),nat()}, {call,decimal,new,[Tuple]}).

prop_sum() ->
  ?FORALL({SD1, SD2}, {decimal(), decimal()},
     begin
       D1 = eval(SD1),
       D2 = eval(SD2),
       decimal_model(decimal:sum(D1, D2)) ==
                    decimal_model(D1) + decimal_model(D2)
     end).
```

Unit testing of data types using properties

Errors reported with symbolic values are easier to understand:

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
........Failed! After 9 tests.
{{call,decimal,new,[2,1]}, {call,decimal,new,[2,2]}}
Shrinking..(2 times)
{{call,decimal,new,[0,1]}, {call,decimal,new,[0,2]}}
false
```

$$0.1 + 0.2 \neq 0.3 \text{ ?}$$

Indeed, due to unavoidable IEEE 754-1985 rounding problem.

Unit testing of data types using properties

Adjust the model:

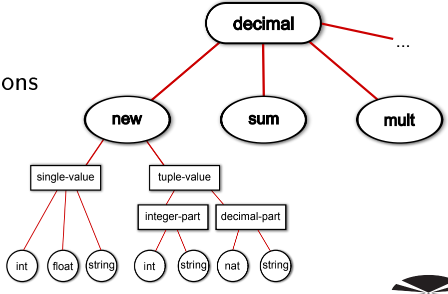
$$a \approx b \Leftrightarrow \left\{ \begin{array}{lll} |a| - |b| & < & \epsilon_{abs} \\ \dfrac{x - y}{x} & < & \epsilon_{rel}, x = \max(|a|, |b|), y = \min(|a|, |b|) \\ & & \epsilon_{abs} = 10^{-16}, \epsilon_{rel} = 10^{-10} \end{array} \right.$$

```
prop_sum() ->
  ?FORALL({SD1, SD2}, {decimal(), decimal()},
     begin
       D1 = eval(SD1),
       D2 = eval(SD2),
       equivalent(decimal_model(decimal:sum(D1, D2)),
                  decimal_model(D1) + decimal_model(D2))
     end).
```

Unit testing of data types using properties

Beware we may feel satisfied with this unit testing but:

- It is **not exhaustive**
  - We have not tested all possibilities for `decimal:new/1`
- It is **not complete**
  - We have not tested operations combination



Unit testing of data types using properties

Introduce **recursive generators** including 'generator' operations:

```
decimal() ->
  ?SIZED(Size, decimal(Size)).

decimal(0) ->
  {call,decimal,new,[oneof([int(), real(), dec_string(),
                            {oneof([int(), list(digit())]),
                             oneof([nat(), list(digit())])}
                             ])]};
decimal(Size) ->
  Smaller = decimal(Size div 2),
  oneof([decimal(0),
         {call, decimal, sum, [Smaller,Smaller]},
         {call, decimal, mult,[Smaller,Smaller]}, ...]).
```

Unit testing of data types using properties

Shrinking customisation can improve counterexample quality:

```
decimal(Size) ->
  Smaller = decimal(Size div 2),
  oneof([decimal(0),
    ?LETSHRINK([D1, D2], [Smaller, Smaller],
               {call, decimal, sum, [D1, D2]}),
    ?LETSHRINK([D1, D2], [Smaller, Smaller],
               {call, decimal, mult, [D1, D2]}), ...]).
```

Unit testing of data types using properties

Error scenarios (**negative testing**) must be checked at properties:

```
prop_divs() ->
 ?FORALL({SD1, SD2}, {decimal(), decimal()},
    begin
      D1 = eval(SD1),
      D2 = eval(SD2),
      case equivalent(decimal_model(D2), 0.0) of
        true ->
          {'EXIT',_} = catch (decimal_model(D1)/
                                        decimal_model(D2)),
          {error, _} = decimal:divs(D1, D2);
        false ->
          equivalent(decimal_model(decimal:divs(D1, D2)),
                          decimal_model(D1)/decimal_model(D2))
      end
    end).
```

Unit testing of data types using properties

Generator must be **well-defined**, with additional base case test:

```
defined(E) ->
  case catch eval(E) of
    {'EXIT', _} -> false
    _Value      -> true;
  end.

well_defined(G) ->
   ?SUCHTHAT(E, G, defined(E)).

decimal() ->
  ?SIZED(Size, well_defined(decimal(Size))).

prop_new() ->
  ?FORALL(SD,decimal(0),is_float(decimal_model(eval(SD)))).
```

Unit testing of data types using properties

Methodology to **guarantee full testing**:

1. Definition of a **suitable model** for the data type
2. Generation of **well-defined, symbolic values**
   - including all productive operations
3. Definition of one **property for each operation**
   - considering expected failing cases
4. **Fine-tuning of shrinking** preferences

State machine-based integration testing

Integration testing

"Do the appropriate components interact as expected when creating a new risk group?"

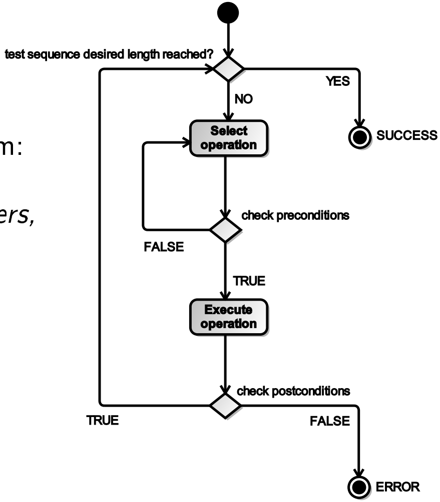
- Software is usually structured in different **components**
- Must be **tested** on their own to ensure they perform correctly, **in combination** to ensure they interact properly
- Components treated as working **black boxes**

QuickCheck state machine testing

`initial_state`

Load information about the system:

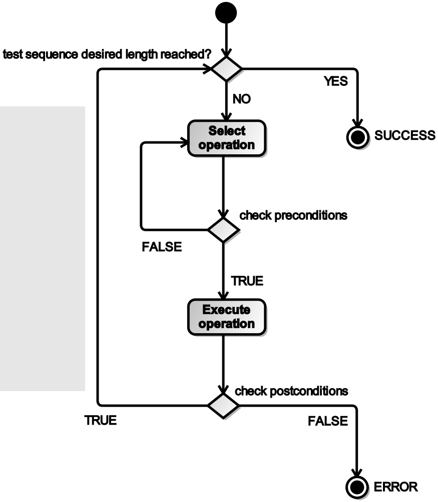
- *user sessions, risk groups identifiers, risk objects identifiers, etc.*



QuickCheck state machine testing

initial_state

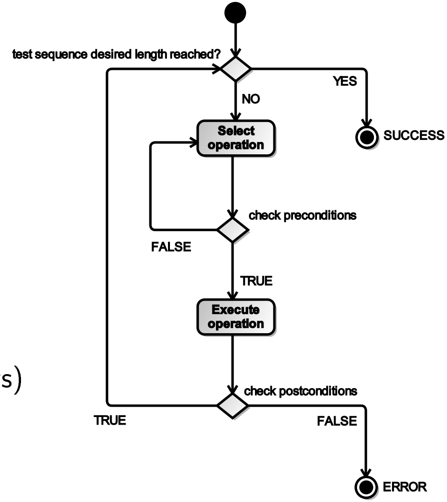
```
-record(state, {user_sessions,
                groups,
                objects}).

initial_state()->
  #state{user_sessions = [],
         groups        = [],
         objects       = []}.
```



QuickCheck state machine testing

`command`

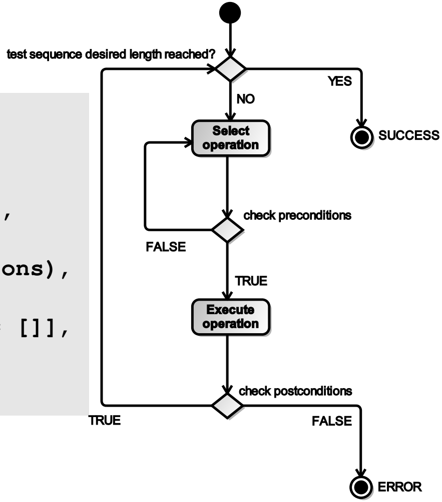
- Pick one of possible service invocations
- Use `frequency` to specify probabilities
- Introduce 'anomalies' (i.e., delays)



QuickCheck state machine testing

command

```
command(S) ->
  frequency(
    ...
    [{7,{call, risk_group_facade,
        new_risk_group,
       [oneof(S#state.user_sessions),
        group_name()]}}
    || S#state.user_sessions =/= []],
    ...
  ).
```

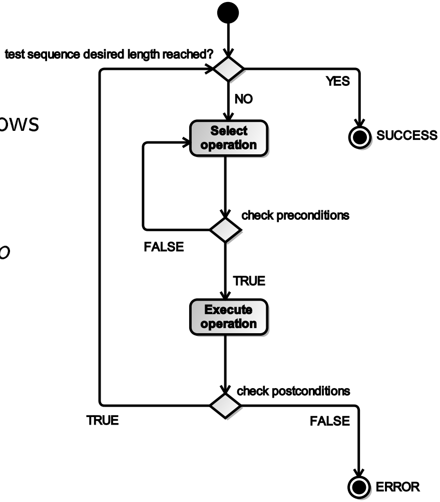


QuickCheck state machine testing

`precondition`

Check whether current status allows invocation of chosen service:

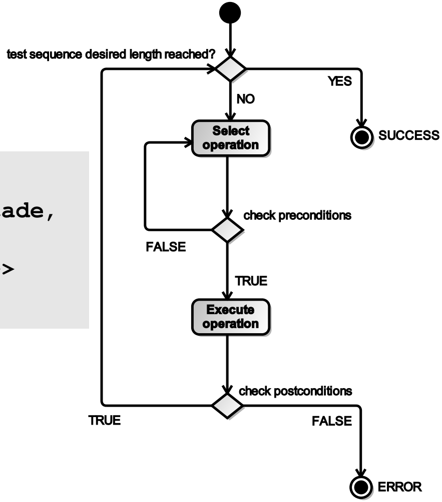
- *no previous conditions required to create a new risk group*



QuickCheck state machine testing

precondition

```
precondition(S,
             {call,risk_group_facade,
              new_risk_group,
             [SessionID, Name]})->
  true;
```

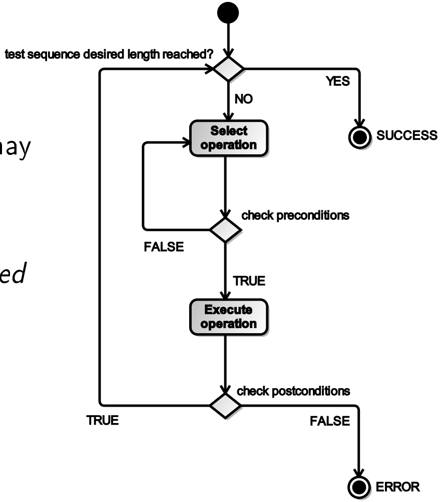


QuickCheck state machine testing

`next_state`

Service invocation execution may affect the test state:

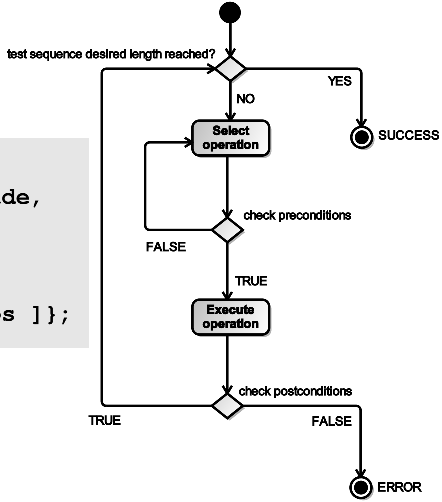
- *a new risk group identifier is added*



QuickCheck state machine testing

next_state

```
next_state(S, Value,
           {call,risk_group_facade,
            new_risk_group,
            Arguments}) ->
  S#state{groups =
          [ Value | S#state.groups ]};
```

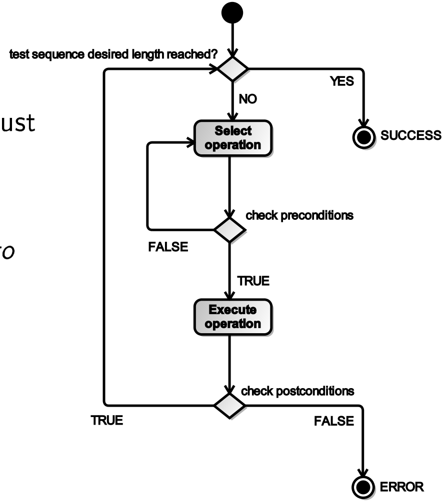


QuickCheck state machine testing

`postcondition`

Check properties the system must hold after service execution:

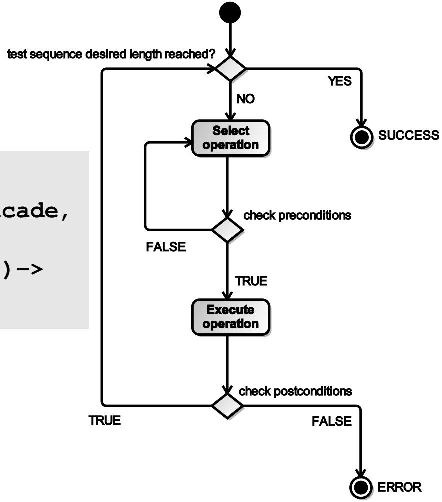
- *no previous conditions required to create a new risk group*



QuickCheck state machine testing

`postcondition`

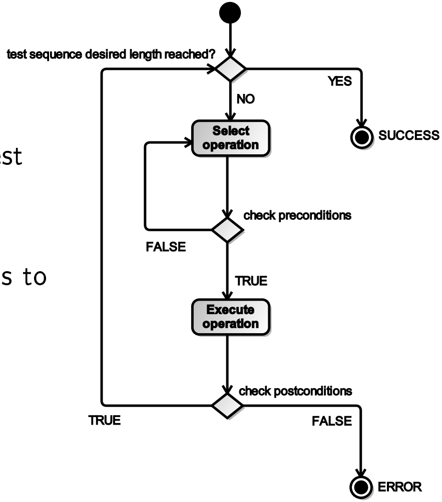
```
postcondition(S,
              {call,risk_group_facade,
               new_risk_group,
               Arguments}, Result)->
  true;
```



QuickCheck state machine testing

Failure

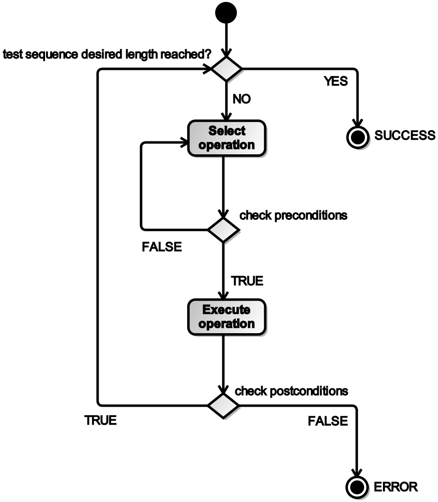
- QuickCheck shrinks the failing test case to the *shortest sequence* of commands which invocation leads to the same error



QuickCheck state machine testing

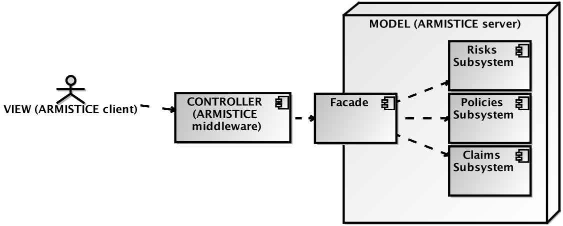
Success

- Proceed to next test case, or
- Exit reporting test pass



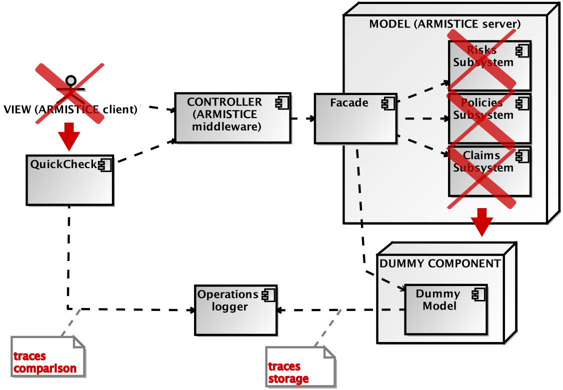
QuickCheck state machine-based integration testing

Integration test success:



QuickCheck state machine-based integration testing

Integration test success: verification of expected function calls (**dummy component**)



- **Reduced effort**
- **Collateral effects avoidance**
- **Early-stage** problems detection

QuickCheck state machine-based integration testing

Original source code:

```
new_risk_group(SessionID, GroupName, GroupDescription)->
  {ok, GroupData} =
        risk_group_dao:new_skeleton(SessionID),
  [{oid,GroupID}, {code,GroupCode},
   {name,DefName},{desc,EmptyDescription}] = GroupData,
  ok = risk_group_dao:update(SessionID, GroupID,
                             GroupName,
                             GroupDescription),
  ok = risk_group_dao:unlock(SessionID, GroupID),
  {ok, [{oid, GroupID}, {code, GroupCode},
        {name, GroupName}, {desc, GroupDescription}]}.
```

QuickCheck state machine-based integration testing

Dummy component source code:

```
new_risk_group(SessionID, GroupName, GroupDescription)->
  GroupData = [{oid,group_id()}, {code,group_code()},
               {name,GroupName}, {desc,GroupDescription}],
  op_logger:add({risk_group_dao, new_skeleton,
                [SessionID], {ok, GroupData}}),
  op_logger:add({risk_group_dao, update,
                [SessionID, GroupID, GroupCode,
                 GroupName, GroupDescription], ok}),
  op_logger:add({risk_group_dao, unlock,
                [SessionID, GroupID], ok}),
  {ok, GroupData}.
```

QuickCheck state machine-based integration testing

Testing state machine **postcondition** and **property**:

```
postcondition(S, {call,risk_group_facade,
                  new_risk_group,Arguments}, Result)->
  Operations = op_logger:get({new_risk_group}),
  check(new_risk_group, {S,Arguments,Operations,Result});

prop_integration() ->
  ?FORALL(Commands, commands(?MODULE),
    begin
      {History,S,Result} = run_commands(?MODULE,Commands),
      Result == ok
    end).
```

QuickCheck state machine-based integration testing

Methodology to fully **test component integration**:

- **Internal state** stores **minimal information** for test generation
- **Transitions are operations** to be tested
- Interactions are performed against **dummy objects**
- **Preconditions** are true, **postconditions** check correct sequence of interactions

Data integrity validation via business rules testing

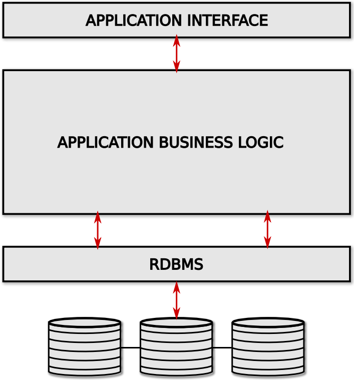
Validation

"Does the system enforce that there is only one policy renewal under construction at a time?"

- Complex, **data-intensive** software usually handles **great number** of business **objects** with **complex relationships**
- **Few** basic data consistency constraints are **enforced by storage media** (i.e., DBMS)
  - Too complex, change dynamically, non-trivial calculations,...

Data integrity validation via business rules testing

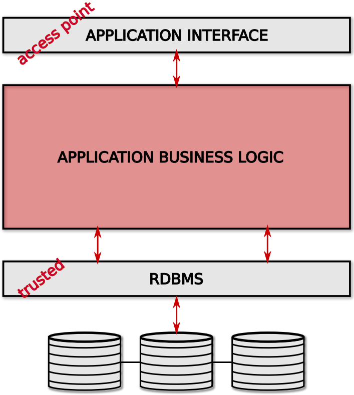
Ensure that **business rules** are **respected at all times**:



- **Sequences** of interface calls could violate them
- Unit testing is **not enough**

Data integrity validation via business rules testing

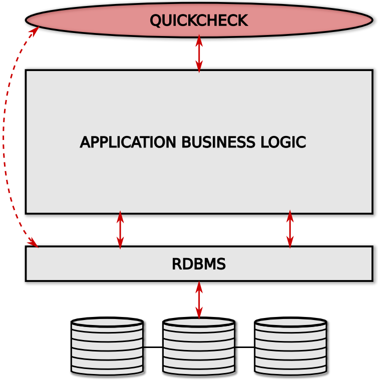
Ensure that **business rules** are **respected at all times**:



- **Sequences** of interface calls could violate them
- Unit testing is **not enough**

Data integrity validation via business rules testing

Ensure that **business rules** are **respected at all times**:



- **Sequences** of interface calls could violate them
- Unit testing is **not enough**

Data integrity validation via business rules testing

Use QuickCheck **state machine** model:

initial_state minimum to generate related test cases

commands interface functions to be tested

precondition `true`

next_state test internal state update

postcondition `true`

Data integrity validation via business rules testing

Business rules compliance check:

- specify **BR as SQL sentence**
- define an **invariant** function to evaluate **BR**

```
business_rule(Connection) ->
   RenConsCount =
    db_interface:process_query(Connection,
      "SELECT COUNT(*) "
      "  FROM renewal "
      " WHERE ren_constr IS TRUE "
      "   AND ren_policy IN (SELECT pol_number "
                             " FROM policy ) "
      " GROUP BY ren_policy "),
   ([] == [ R || R <- RenConsCount, R > 1]).
```

Data integrity validation via business rules testing

```
invariant() ->
  {ok, Connection} = db_interface:start_transaction(),
  Result = business_rule(Connection),
  ...
  db_interface:rollback_transaction(Connection),
  Result.

prop_business_logic() ->
  ?FORALL(Commands, commands(?MODULE),
    begin
      true = invariant(),
      {History,S,Result} = run_commands(?MODULE,Commands),
      PostCondition = invariant(),
      clean_up(S),
      PostCondition and (Result == ok)
    end).
```

Data integrity validation via business rules testing

Methodology to **test business rules**:

- **Minimal internal state** to conduct related operations
- **Transitions** are public interface exported **functions**
- **Preconditions** and **postconditions** are true
- **Business rules** (formulated as SQL sentences) are tested as **invariants** after test execution

Conclusions

Functional programming represents a **serious alternative** for developing **real-world software**:

- A **high-level** abstraction tool, close to **concepts and requirements**
- Very **expressive and effective** way of **implementing** complex algorithms
- A context disposed to **powerful testing** techniques