Writing Business Rules Engines in Mercury

Ian MacLarty

24 July 2009

missioncriticalit.com

What is a Business Rules Engine?

- Domain Experts create rules that define the behaviour of the system.
- Rules are of the form: if *condition* then *consequence*.
- Rules act on a *model* of the system.
- For example: if *period in empoyment* < *3 months and assets* < *$10000* then *reject the loan*.
- Makes it easier for non-developers to adjust the behaviour of the system.
- Separates “business” knowledge from I.T knowledge.

Example: Ilog JRules (IBM)

- Rules act on Java (.NET) objects directly by invoking methods.
- For example:

  if `applicant.getEmployedMonths() < 3` *and*
  `applicant.getAssets() < 10000` then
  `applicant.rejectLoan()`

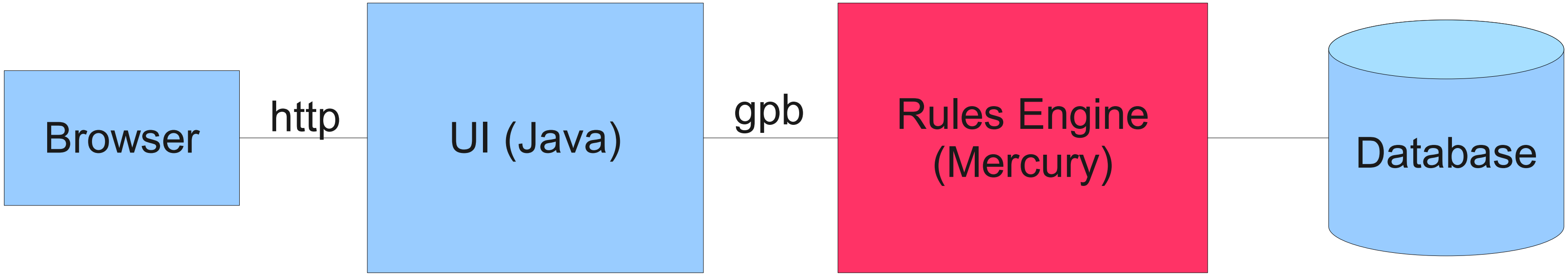
- Actions or conditions may have side effects.
  - Can only execute the rules one way.
  - Users must worry about priority of rules.
  - Limited debugging (no declarative debugging, no retry)

MC Rules Engine

- Declarative (FOL)
- Based on SWRL (rules) and OWL (model)
- Many different ways to use rules:
  - Compute results
  - Error messages
  - Work out what questions to ask user to try to achieve a particular result
  - Declarative debugging
- Hopefully simpler for domain experts

Where the rules engine fits in

Simplified architecture of a “typical” MC app:



Modelling language (OWL)

- *Classes* (sets of *individuals*)
  - Subclass, Union, Intersection
  - Complement (not that useful because of OWA)
  - E.g. Applicant, RejectedApplicant
- *Properties* (binary relations)
  - Functional, Transitive, Symmetric
  - Domain, Range
  - E.g. months_employed, assets

Rule language (SWRL)

- Horn clauses
- Allowed atoms:
  - Class literals
  - Property literals
  - “Builtin” literals
- E.g.

```
months_employed(?applicant, ?months) ∧
lessThan(?months, 3) ∧
assets(?applicant, ?assets) ∧
lessThan(?assets, 10000.0)
→ RejectedApplicant(?applicant)
```

Some more (real) example rules:

- Compute risk tolerance:

```
risk_tolerance_score(?investor, ?score) ∧
greaterThan(?score, 32) ∧
lessThanOrEqual(?score, 48)
→ risk_tolerance(?investor, defensive)
```

- Validation rule:

```
retirement_savings_premium(?investor, ?premium)
→ lessThanOrEqual(?premium, 870.0)
```

Evaluating the rules in Mercury

```
:- pred swrl_query(snapshot(Store)::in,
                   Builtins::in,
                   Program::in,
                   swrl_conjunction::in,
                   set(swrl_substitution)::out) is det
   <= ( rdf_store(Store),
        builtins_structure(Builtins),
        swrl_program(Program) ).
```

Reading the database without the IO state.

- Some of the rule engines use backtracking, so can't take the IO state.
- snapshot(Store) represents a snapshot of the database of type Store.
- Queries on a snapshot always return the same results, so it can be pure without requiring the IO state.
- Enforced using *repeatable read* transaction.
- Can only create a snapshot by opening a transaction:

```
:- pred transaction(
    pred(snapshot(Store), T)::in(pred(in, out) is det),
    Store::rdfin, Store::rdfout, io::di, io::uo) is det
    <= rdf_store(Store).
```

Custom Builtins

- The SWRL spec allows for custom builtins.
- We allow custom builtins by supplying a typeclass:

```
:- typeclass builtins_structure(Structure) where [
     pred evaluate_builtin(snapshot(Store)::in,
        Structure::in, builtin_id::in, swrl_args::in,
        builtin_result::out) is det
     <= rdf_store(Store)
].

:- type builtin_result
    --->   ok(set(swrl_substitution))
    ;      unbound_var
    ;      not_supported.
```

Notes on the builtins structure typeclass

- Lack of IO state means builtins cannot have side effects and must be pure (i.e. produce the same results for the same inputs).
- Important that builtins don't have side effects, because that would limit how we can evaluate the rules (would impose an operational semantics)
- If the arguments are not sufficiently instantiated then the builtin can return 'unbound_var' and the engine can delay the builtin until more arguments are instantiated.

Some example builtins

- Standard builtins:
  - add, subtract, multiply, greaterThan, lessThan, etc
- Get the current date:
  - today(?today)
  - Current date set in builtins_structure, so always returns same result when evaluating a query and not IO state required.
- Evaluate a spreadsheet:
  - eval_spreadsheet(“data.ods”, “A1”, ?input, “B2”, ?output)
  - Spreadsheet parsed and stored in builtins_structure before query run.

Top-down, non-deterministic engine

- First engine implemented.
- Can do expensive re-evaluation (no tabling)
- Does not handle rules such as:

    partner(?x, ?y) → partner(?y, ?x)

- Was the main reason for adding snapshots and omitting the IO state from the query predicate.

Tracing engine

- Generates a trace.
- Required re-implementing non-deterministic engine to be deterministic, so that we could thread a trace state around.
- Trace used to do declarative debugging.
- Also to generate proof trees for validation error messages.

age(?investor, ?age) → greaterThanOrEqual(?age, 18)

Mercury Tabled engine

- Non-deterministic.
- Use Mercury's memoing to avoid recomputation (can be expensive when querying databases).
- Use Mercury's minimal model tabling to handle rules such as: partner(?x, ?y) → partner(?y, ?x)
- Required a few “dirty tricks” to get right (e.g. memoing snapshot by pointer)
- Buggy, and debugging difficult.
- Not really sufficient control over memo-table (e.g. couldn't clear table for one particular snapshot).

Transparent Tabled engine

- Deterministic.
- Thread around an explicit memo table.
- Inspired by OLDT resolution.
- Much more control over memo table.
- Code quite simple (only ~450 lines).
- Performance very good so far.
- Easier to implement optimisations with deterministic code (harder to reason about operational semantics with non-det code).

Transparent Tabled engine benchmarks:

Test	MC	Pellet
1	0.28	7.73
2	0.51	27.46
3	0.84	552.81
4	0.24	8.51

Other engines

- “Set” engine – tries to group queries to the database, so that joins can be done on the SQL database.
- Constraint solving engine?

Demo...

Questions?