The Reactor Model: An Experiment in Declarative Concurrent Programming

*John Field (IBM Research)
Maria-Cristina Marinescu (U. Carlos III, Madrid)
Christian Stefansen (Google)

Datalog 2.0, Oxford
17 March 2010
Do these apps have anything in common?

cloud-based web 2.0

embedded network

real-time data analysis
Yes.

- collection of distributed, concurrent components
- components are loosely coupled by messages, persistent data
- irregular concurrency, driven by real-world data (“reactive”)
- high data volumes
- fault-tolerance important
Why this is hard: an AJAX parable
Web programming: state of the art

Web programming: state of the art
requires extensive rewriting when moving applications between tiers or updating functionality; doesn’t compose, scale well

 ZIP code:  
 City:  
 State:  

babble of languages

same logical data; many different physical representations

no notion of component interface for composition

fragile plumbing required to compose
A prettier picture: app composed from encapsulated, distributed components
Web apps just one instance of reactive distributed applications

- have distributed control
- integrate front-end, back-end, and database data
- typically combine presentation logic, “business logic”, data query and access
- functionality can evolve and migrate between tiers
- dynamic instantiation of logical processes
- complex failure modes
Our (long-term) goals

• develop programming models for reactive distributed applications which
  – have same model for presentation, business logic, data access
  – simplify composition, evolution, and application maintenance

• target applications: web applications, distributed business applications, “software supply chains”

• this talk: initial experiments on a highly declarative model for this class of application

• some ideas have been incorporated in more recent work on a new scripting language for distributed applications (Thorn)
Reactors: key attributes

Synthesis of ideas from

- *synchronous languages*: event handling, synchronization
- *datalog*: expressive data query language
  - NB: here we are using datalog for computation, not just query
- *actor model*: simple model for dynamic creation of processes, asynchronous process interaction
- *transactions*:
  - system evolves (only) from consistent state to consistent state
  - concurrent computations isolated from one another
- *XForms*: declarative specification of web forms
Basic reactor: “database” with external updates

- reactor encapsulates state in the form of fixed set of relations (sets of tuples)
- each update is a “bundle” of additions ($\Delta^+$) and deletions ($\Delta^-$), applied atomically
- $\Delta^+$ and $\Delta^-$ must be disjoint
- state updates arrive asynchronously, and are queued
- in most basic form of reaction, updates simply applied to previous state ($s_{i-1}$) to yield new state ($s_i$)
Inside a reaction: stimulus and response

- $s_i$ (response state): observable state of reaction $i$
- $^s s_i$ (stimulus state): result of applying update to $s_{i-1}$ (“pre-state”)
- in the absence of any computational rules, $s_i = ^s s_i$
Adding computation via *rules*

- reaction’s computation defined using datalog-style rules
- rules define value of each relation \( r, s, \ldots \) as function of
  - \( -r, -s, \ldots \) (pre-state values: values of \( r, s, \ldots \) in previous reaction)
  - \( ^r, ^s, \ldots \) (stimulus values: result of applying update bundle to \( -r, -s, \ldots \))
  - \( r, s, \ldots \) (can appear recursively in rules)
- reactor computation (*reaction*) is *atomic*: intermediate states of rule evaluation not visible to other reactors
Reactor rule evaluation

- rules evaluated whenever state is updated
- no change made to relation unless necessary to satisfy a rule
- semantics ensures unique solution
- rule declaration order irrelevant
  - easy to compose rules to yield new functionality without concern for data or control dependencies (“aspect”-like)
- extension of standard datalog semantics
- as usual, negation requires care

only **public** relations can be updated from outside

contents of ephemeral relations do not persist between reactions

public ephemeral relations can be used to model “events”

LHS negation denotes *deletion*

```log
public orders: (int, int, int). 
log: (int, int, int). 

public ephemeral updateLog: ().

log(orderid, itemid, qty) <- orders(orderid, itemid, qty), 
updateLog().
not log(orderid, itemid, qty) <- not orders(orderid, itemid, qty).
```
Multiple reactors

- Reactor $R_1$: reactor’s computation can generate new state updates for other reactors ...
- ... or the same reactor
- ... and can *dynamically* instantiate new reactors
Reactor references

- relations may contain references to other reactors
- reactors may be instantiated dynamically
- rules may refer to relations of remote reactors via references
Asynchronous interaction via future state \((r^\wedge)\)

- **future state value**: \(r^\wedge\)
- \(r^\wedge\) defines *asynchronous* state updates (to this or other reactors via references)
- rules compute \(r^\wedge\) as function of \(r, -r, ^r\)
Running example: order entry application

Form + JScript Code

- City
- ZIP code
- State
- Credit Card Number

Submit

asynchronous interaction

Form Submission Servlet

Zip Database

Zip Lookup Servlet

Merchant Credit Server

User Credit Servers

synchronous interaction

synchronous interaction

synchronous interaction
Running example: order entry application
Order entry: validation and submission

FormController

InputWidget (city)

InputWidget (state)

InputWidget (zip)

ButtonWidget (submit)

reactor references for composition

SessionController

asynchronous request from FormController

asynchronous user inputs
Order entry: validation and submission

FormController

rCity: (ref InputWidget).
rState: (ref InputWidget).
rZip: (ref InputWidget).
rSubmit: (ref ButtonWidget).
rSession: (ref SessionController).

ephemeral validated: (string, string, string).
public ephemeral buttonPressed: ()

validated(z, c, s) <-
  zW.val(z), cW.val(c), sW.val(s),
  z <> "", c <> "", s <> "",
  rZip(zW), rCity(cW), rState(sW).

sC.commitOrder^ (z, c, s) <-
  buttonPressed(sB), validated(z, c, s),
  rSubmit(sB), rSession(sC).

SessionController

public ephemeral commitOrder:
  (string, string, string).

public ephemeral filled: ()

ButtonWidget (submit)

public label: (string).
public listeners: (ref FormController)

ephemeral pressed: ()

rF, buttonPressed(self) <-
  listeners(rF), pressed().

InputWidget (zip)

public label: (string).
public val: (string).
public ephemeral filled: ()

ephemeral committed: ()

public ephemeral filled: ()

Set of listeners that are notified when button pressed

references to UI widgets and session server

(private) ephemeral relation holds result of validation for current reaction only

References to relations of remote reactors

nullary ephemeral relation used to model UI events

Reference to future state of remote reference: spawns asynchronous reaction

References to UI widgets and session server
Running example: order entry application

Form + JScript Code

Zip Database

Zip Lookup Servlet

Form Submission Servlet
Order entry: adding "AJAX"-style zip lookup

- asynchronous response from AjaxController
- asynchronous request from FormController
- database accessed synchronously with receipt of request

- widgets updated synchronously with response
- input widgets for city, state, zip
- button widget for submit
Order entry: adding “AJAX”-style zip lookup

**FormController**

- `rAjax: (ref AjaxController).`
- `public ephemeral formFilled: ()`
- `public ephemeral lookupResult: (string, string).`
- `zC.lookupReq^(z, self) <- formFilled(zW), zW.val(z), rZip(zW), rAjax(aC).`
- `cW.val(c), sW.val(s) <- lookupResult(c, s), rCity(cW), rState(sW).`

**AjaxController**

- `rDatabase: (ref DB).`
- `public ephemeral lookupReq: (string, ref FormController).`
- `fC.lookupResult^(c, s) <- lookupReq(z, fC), dB(z, c, s), rDatabase(dB).`

**DB**

- `public zipCityState: (string, string, string).`

**InputWidget (zip)**

- `label: (string).`
- `val: (string).`
- `public ephemeral filled: ()`
Running example: order entry application
Order entry: dynamic widget instantiation

- FormController
  - InputWidget (city)
  - InputWidget (state)
  - InputWidget (zip)
  - ButtonWidget (submit)

- ZipController
  - InputWidget (card)
  - ButtonWidget (next)

- SessionController

- DB

Dynamically instantiate new widgets when "next" button pressed
Order entry: dynamic widget instantiation

FormController

rNext: (ref ButtonWidget)
rCard: (ref InputWidget)
rSubmit: (ref ButtonWidget)

rCard(new), rSubmit(new), not rNext(_) <-
buttonPressed(nB), validated(_,_,_),
-rNext(nB).

sC.commitOrder^(z, c, s, cc) <-
buttonPressed(sB), rSubmit(sB)
validated(z, c, s),
cW.val(cc), cc <> "", rCard(cW)
rSession(sC).

NB: must refer to pre-state to ensure stratification

new widgets instantiated when "next" button pressed

old widget reference removed

commitOrder now validates and sends credit card info
Order entry: adding synchronous / atomic account transfer

Transfer between user and merchant accts must occur synchronously and atomically.
Order entry: adding synchronous / atomic account transfer

SessionController

... 
cardAccts: (ref Acct, string).
rMerchantAcct: (ref Acct).
orderTotal: (int).
ephemeral rUserAcct: (ref Acct).

rUserAcct(uA) <- commitOrder(_,_,_,cc),
cardAccts(uA, cc).

uA.balance(b - amt) :=
  uA.-balance(b), orderTotal(amt),
  commitOrder(_,_,_,_),
rUserAcct(uA).

mA.balance(b + amt) :=
  mA.-balance(b), orderTotal(amt),
  commitOrder(_,_,_,_),
rMerchantAcct(mA).

Acct (user)

public balance: (int).

FAIL <- balance(x), balance(y),
   x <> y.
FAIL <- balance(x), x < 0.

Acct (merchant)

public balance: (int).

FAIL <- balance(x), balance(y),
   x <> y.
FAIL <- balance(x), x < 0.

Fails is syntactic sugar for a predefined inconsistency, hence can be used to define consistency constraints

Examples of inconsistent rules:
  r(...) <- .
  not r(...) <- .
  such rules are mutually unsatisfiable

Reactions that yield inconsistency roll back to pre-state
Reactor semantics: some issues (I)

• Handling reactor references:
  – given reference of form \( c . r(x) \) \( <- \ldots \)
  – rewrite as \( r'(c, x) \) \( <- \ldots \)

• Head negation:
  – compute positive and negative information separately
  – explicitly check for consistency before committing result
  – translate rules to datalog with goal (rhs) negation only

• Rules required to be \textit{stratified} to ensure unique solution
  – stratification check must account for remote references
Reactor semantics: some issues (II)

- Scope *extrusion* semantics for composite synchronous reactions defines which reactor instances are affected
  - scope of reaction extrudes to reactor R whenever another reaction attempts to write to response state of relation of R

- Optimistic concurrency control
  - composite reactions operate on *snapshots* of reactor state
  - reaction rolls back, update re-queued if states of reactors have changed prior to commitment

- Reactor instantiation:
  - in each reaction...
  - for each *new* instance in rule head...
  - ...generate a globally unique reference for each combination of values bound to variables in the rules
  - modeled using functors
Related work

- actor model
- transactional languages (Argus, Camelot)
- synchronous languages (Esterel, Lustre, Signal, ...)
- concurrent constraint programming
- process calculi (Pi calculus, Join calculus, Ambient calculus, ...)
- “tierless” web programming models (Links, Hop, Google Web Toolkit...)
- “stream-oriented” web programming models (Flapjax, ...)
- modern relational databases (triggers, constraints, ...)
- various web services standards
- event-condition-action rules
- Axml
- <your missing reference goes here>
Review

• data, rather than ports or channels as interface to process
• expressive data query and transformation
• declarative, data-driven, compositional specification of functionality in an "aspect-like" manner
• constraints/assertions expressible in core language
• synchronous and asynchronous interaction in same model
• simple transactional model
• dynamic process creation
Details

• Theoretical Computer Science ‘09
Questions?
Head Negation

- public orders: (int, int, int).
- log: (int, int, int)

log(id, itemid, qty) <- orders(id, itemid, qty).
not log(id, itemid, qty) <- not orders(id, itemid, qty).

- ephemeral orders⁺: (int, int, int).
- ephemeral orders⁻: (int, int, int).
- ephemeral log⁺: (int, int, int).
- ephemeral log⁻: (int, int, int).

log⁺(id, itemid, qty) <- ^orders(id, itemid, qty),
orders⁺(id, itemid, qty).

log⁻(id, itemid, qty) <- not ^orders(id, itemid, qty),
^log(id, itemid, qty),
not orders⁺(id, itemid, qty).

if (log⁺ ∩ log⁻ = ∅) then log := ^log U log⁺ \ log⁺
Reactor syntax

**REACTOR** ::= `def reactor-type-name = { {DECL .} `}

**ENUM** ::= `enum enum-type-name = { {atom-name ,}+ `}

**DECL** ::= `( RELATION-DECL | RULE-DECL )`

**RELATION-DECL** ::= `[public | public write | public read][ephemeral] rel-name : ( {TYPE ,} `)

**RULE-DECL** ::= `HEAD-CLAUSE <- {BODY-CLAUSE ,}+

**HEAD-CLAUSE** ::= `[not] [var-name .]rel-name[^] ( {(_ | var-name) ,} `)

**BODY-CLAUSE** ::= `[not][var-name .][^|-]rel-name ( {(_ | var-name) ,} ` | BASIC-PREDICATE

**BASIC-PREDICATE** ::= `EXP (< | > | <> | =) EXP

**TYPE** ::= `int | string | enum-type-name | ref reactor-type-name`

**EXP** ::= `var-name | NUMERIC-LITERAL | STRING-LITERAL | EXP (+ | - | * | / | %) EXP | new reactor-name | self`