From UML State-Machine Diagrams to Erlang

Ricardo J. Rodríguez, Lars-Åke Fredlund, Ángel Herranz

🄯 **All wrongs reversed**

{rjrodriguez, lfredlund, aherranz}@fi.upm.es

Universidad Politécnica de Madrid
Madrid, Spain

September 20th, 2013

XIII Jornadas sobre Programación y Lenguajes (PROLE)
Facultad de Informática, Universidad Complutense de Madrid

Outline

1. Motivation
2. A Transformation Approach
3. Related Work
4. Conclusions and Future Work

Outline

1. Motivation
2. A Transformation Approach
3. Related Work
4. Conclusions and Future Work

Motivation (I)

Software Development Life-Cycle

- Phased involved for developing (and maintaining) software systems and code
- Deployment (production): what happens when faults (or changes) raise?

Motivation (I)

Software Development Life-Cycle

- Phased involved for developing (and maintaining) software systems and code
- Deployment (production): what happens when faults (or changes) raise?
  - Were they initially taken into account? Otherwise → need to redesign

Motivation (I)

Software Development Life-Cycle

- Phased involved for developing (and maintaining) software systems and code
- Deployment (production): what happens when faults (or changes) raise?
  - Were they initially taken into account? Otherwise → need to redesign
- Model-Driven Engineering
  - Increase productivity, simplifying design
  - Maximise compatibility between systems

Motivation (I)

Software Development Life-Cycle

- Phased involved for developing (and maintaining) software systems and code
- Deployment (production): what happens when faults (or changes) raise?
  - Were they initially taken into account? Otherwise → need to redesign
- Model-Driven Engineering
  - Increase productivity, simplifying design
  - Maximise compatibility between systems

Verify correctness BEFORE deployment

Motivation (I)

Software Development Life-Cycle

- Phased involved for developing (and maintaining) software systems and code
- Deployment (production): what happens when faults (or changes) raise?
  - Were they initially taken into account? Otherwise → need to redesign
- Model-Driven Engineering
  - Increase productivity, simplifying design
  - Maximise compatibility between systems

Verify correctness BEFORE deployment

How?: Using model-checking

- Proofs of correctness
- Counter-examples (why not correct)

Motivation (II)

We have mixed. . .

- UML: standard *de facto* as modelling language
  - UML State Machines (UML-SMs): dynamic system behaviour
  - Assumption: intercommunication through asynchronous channels
- Erlang: functional and concurrent programming language
  - Native support for concurrency, distribution and fault tolerance
  - Concurrency based on asynchronous message passing
  - Widely used in the industry: T-Mobile, Ericsson, FB, WhatsApp. . .

Motivation (II)

We have mixed...

- UML: standard *de facto* as modelling language
  - UML State Machines (UML-SMs): dynamic system behaviour
  - Assumption: intercommunication through asynchronous channels
- Erlang: functional and concurrent programming language
  - Native support for concurrency, distribution and fault tolerance
  - Concurrency based on asynchronous message passing
  - Widely used in the industry: T-Mobile, Ericsson, FB, WhatsApp...

Contributions

- Minimise development time
  - Automatically generate Erlang code from UML-SMs

Motivation (II)

We have mixed. . .

- UML: standard *de facto* as modelling language
  - UML State Machines (UML-SMs): dynamic system behaviour
  - Assumption: intercommunication through asynchronous channels
- Erlang: functional and concurrent programming language
  - Native support for concurrency, distribution and fault tolerance
  - Concurrency based on asynchronous message passing
  - Widely used in the industry: T-Mobile, Ericsson, FB, WhatsApp. . .

Contributions

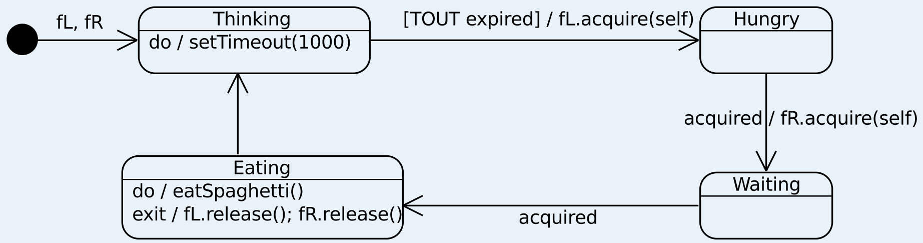
- Minimise development time
  - Automatically generate Erlang code from UML-SMs
- Detect problems in early stages (save efforts and costs)
  - Apply Erlang-based model checking techniques into UML-SMs

Outline

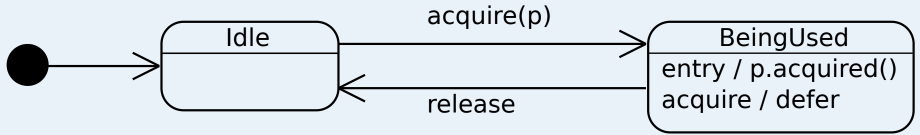
1. Motivation
2. A Transformation Approach
3. Related Work
4. Conclusions and Future Work

A Transformation Approach: The Dining Philosophers (I)

Philosopher


fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

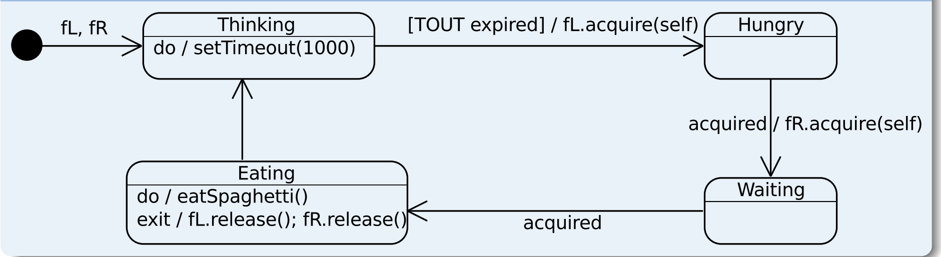
Fork



Please note: thinking time and fork grabbing order

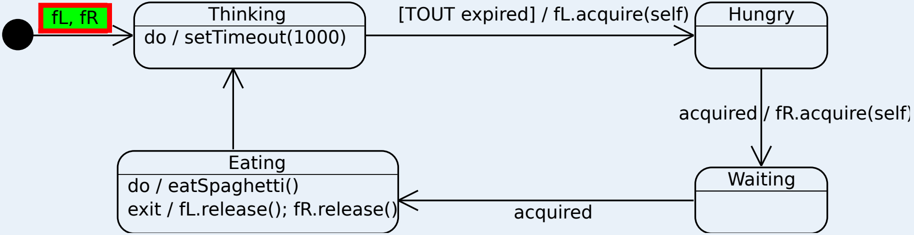
A Transformation Approach: The Dining Philosophers (II)

Philosopher


fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

A Transformation Approach: The Dining Philosophers (II)

Philosopher



fL, fR

Thinking

do / setTimeout(1000)

[TOUT expired] / fL.acquire(self)

Hungry

acquired / fR.acquire(self)

Waiting

acquired

Eating

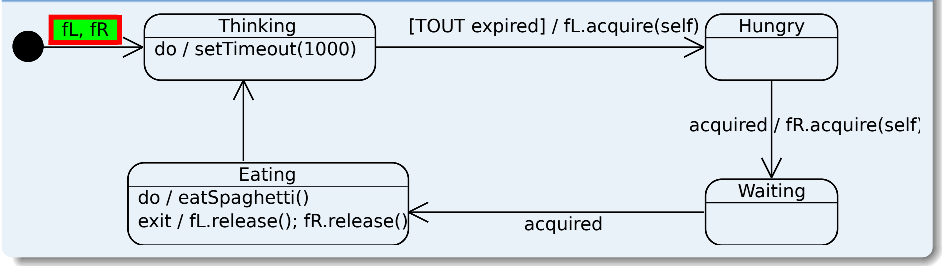
do / eatSpaghetti()

exit / fL.release(); fR.release()

$$\mathcal{P} = \{fL, fR\}$$
$$\mathcal{E} = \{acquired\}$$

A Transformation Approach: The Dining Philosophers (II)

Philosopher

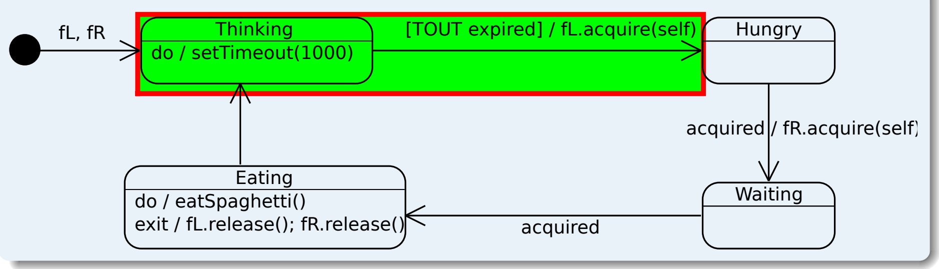

fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

$\mathcal{P} = \{fL, fR\}$
$\mathcal{E} = \{acquired\}$

```
-module(philosopher).
-export([start/2]).
start(FL, FR) ->
  spawn(fun() ->  thinking(FL, FR) end).
```

A Transformation Approach: The Dining Philosophers (II)

Philosopher

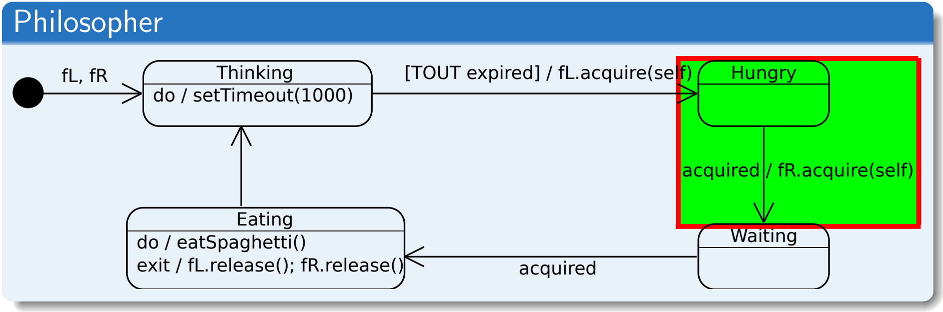

fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

$\mathcal{P} = \{fL, fR\}$
$\mathcal{E} = \{acquired\}$

```
-module(philosopher).
-export([start/2]).
start(FL, FR) ->
  spawn(fun() ->  thinking(FL, FR) end).
```

```
thinking(FL, FR) ->
  receive
    X -> thinking(FL, FR)
    after 1000 ->
      FL!{acquire, self()}, hungry(FL, FR)
  end.
```

A Transformation Approach: The Dining Philosophers (II)


Philosopher
fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

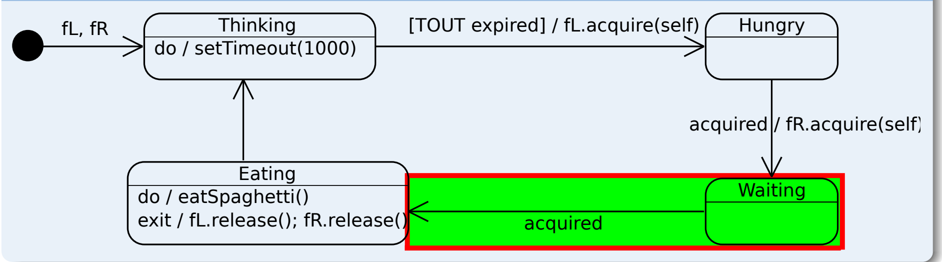
$\mathcal{P} = \{fL, fR\}$
$\mathcal{E} = \{acquired\}$

```
-module(philosopher).
-export([start/2]).
start(FL, FR) ->
  spawn(fun() ->  thinking(FL, FR) end).
thinking(FL, FR) -> ...
```

```
hungry(FL, FR) ->
 receive
   acquired ->
     FR!{acquire, self()}, waiting(FL, FR)
 end.
```

A Transformation Approach: The Dining Philosophers (II)

Philosopher

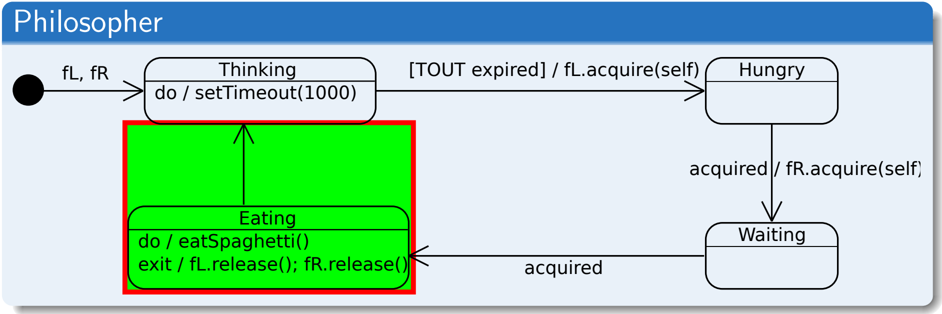

fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()
acquired
Waiting

$\mathcal{P} = \{fL, fR\}$
$\mathcal{E} = \{acquired\}$

```
-module(philosopher).
-export([start/2]).
start(FL, FR) ->
  spawn(fun() -> thinking(FL, FR) end).
thinking(FL, FR) -> ...
hungry(FL, FR) -> ...
```

```
waiting(FL, FR) ->
receive
  acquired -> eating(FL, FR)
end.
```

A Transformation Approach: The Dining Philosophers (II)


Philosopher
fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

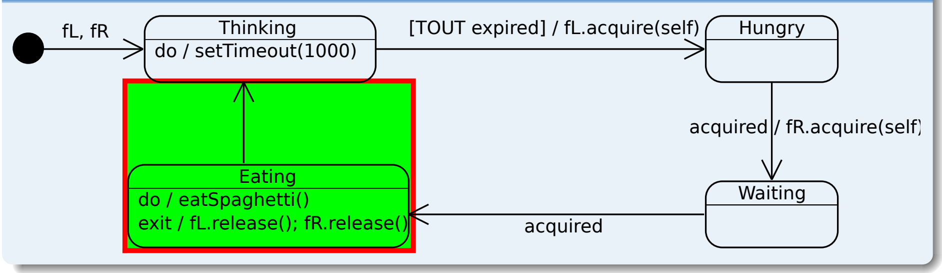
$\mathcal{P} = \{fL, fR\}$
$\mathcal{E} = \{acquired\}$

```
-module(philosopher).
-export([start/2]).
start(FL, FR) ->
  spawn(fun() -> thinking(FL, FR) end).
thinking(FL, FR) -> ...
hungry(FL, FR) -> ...
waiting(FL, FR) -> ...
```

```
eating(FL, FR) ->
  eatSpaghetti(),
  FL!release, FR!release,
  thinking(FL, FR).
```

A Transformation Approach: The Dining Philosophers (II)

Philosopher


fL, fR
Thinking
do / setTimeout(1000)
[TOUT expired] / fL.acquire(self)
Hungry
acquired / fR.acquire(self)
Waiting
acquired
Eating
do / eatSpaghetti()
exit / fL.release(); fR.release()

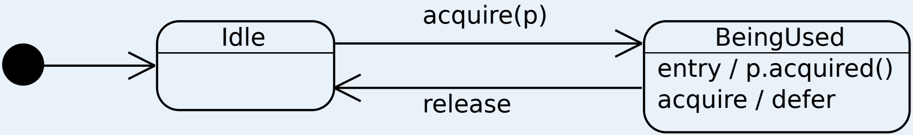
$\mathcal{P} = \{fL, fR\}$
$\mathcal{E} = \{acquired\}$

```
-module(philosopher).
-export([start/2]).
start(FL, FR) ->
  spawn(fun() ->  thinking(FL, FR) end).
thinking(FL, FR) -> ...
hungry(FL, FR) -> ...
waiting(FL, FR) -> ...
eating(FL, FR) -> ...
```

A UML instance is an Erlang process

A Transformation Approach: The Dining Philosophers (III)

Fork



Idle

acquire(p)

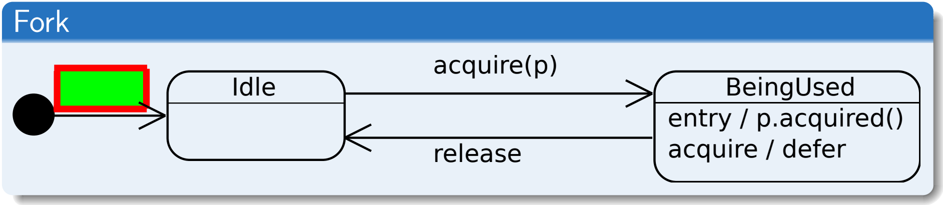
release

BeingUsed

entry / p.acquired()

acquire / defer

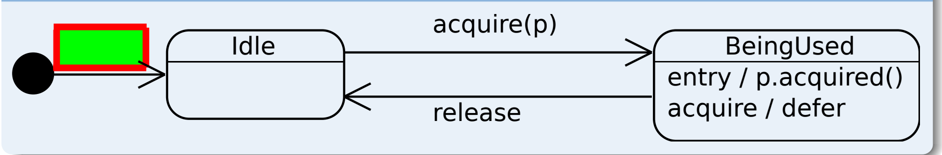
A Transformation Approach: The Dining Philosophers (III)



$\mathcal{P} = \emptyset$
$\mathcal{E} = \{acquire, release\}$

A Transformation Approach: The Dining Philosophers (III)

Fork

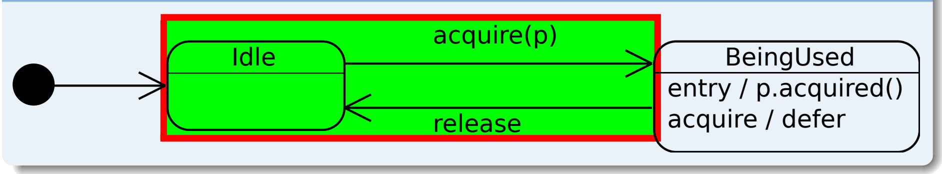


$\mathcal{P} = \emptyset$
$\mathcal{E} = \{\mathit{acquire}, \mathit{release}\}$

```
-module(fork).
-export([start/0]).
start() ->
   spawn(fun() -> idle() end).
```

A Transformation Approach: The Dining Philosophers (III)

Fork



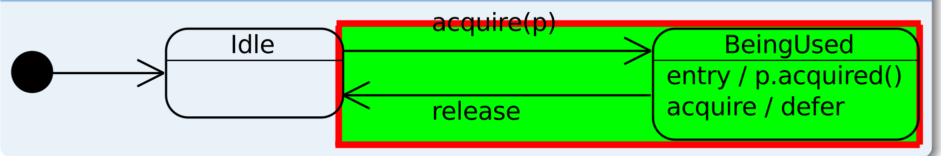
$\mathcal{P} = \emptyset$
$\mathcal{E} = \{\textit{acquire}, \textit{release}\}$

```
-module(fork).
-export([start/0]).
start() ->
   spawn(fun() -> idle() end).
```

```
idle() ->
  receive
    {acquire, P} -> beingUsed(P);
    X -> idle()
  end.
```

A Transformation Approach: The Dining Philosophers (III)

Fork



$\mathcal{P} = \emptyset$
$\mathcal{E} = \{acquire, release\}$

```
-module(fork).
-export([start/0]).
start() ->
   spawn(fun() -> idle() end).
idle() -> ...
```

```
beingUsed(P) ->
 P!acquired(),
 receive
   release -> idle()
 end.
```

A Transformation Approach: The Dining Philosophers (IV)

Wait! Explain me about `defer`...

A Transformation Approach: The Dining Philosophers (IV)

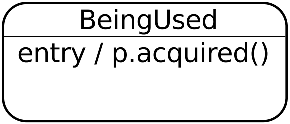
Wait! Explain me about `defer`...

Assume that current state is `BeingUsed`, and `acquire` event is received

A Transformation Approach: The Dining Philosophers (IV)

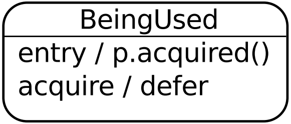
Wait! Explain me about `defer`...

Assume that current state is `BeingUsed`, and `acquire` event is received



BeingUsed

entry / p.acquired()



BeingUsed

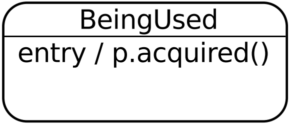
entry / p.acquired()

acquire / defer

A Transformation Approach: The Dining Philosophers (IV)

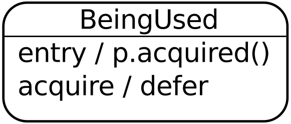
Wait! Explain me about `defer`...

Assume that current state is `BeingUsed`, and `acquire` event is received



BeingUsed

entry / p.acquired()



BeingUsed

entry / p.acquired()
acquire / defer

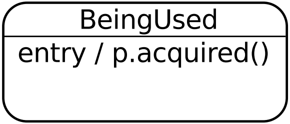
- `acquire` is received
- Something to do?

- `acquire` is received
- Something to do?

A Transformation Approach: The Dining Philosophers (IV)

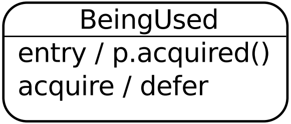
Wait! Explain me about `defer`...

Assume that current state is `BeingUsed`, and `acquire` event is received



BeingUsed

entry / p.acquired()



BeingUsed

entry / p.acquired()

acquire / defer

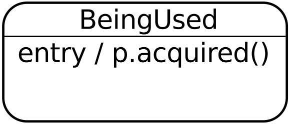
- `acquire` is received
- Something to do?
  - No. Do nothing.

- `acquire` is received
- Something to do?
  - Yes. Defers it.

A Transformation Approach: The Dining Philosophers (IV)

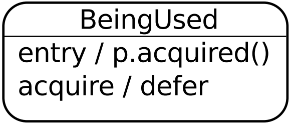
Wait! Explain me about `defer`...

Assume that current state is `BeingUsed`, and `acquire` event is received



BeingUsed

entry / p.acquired()



BeingUsed

entry / p.acquired()

acquire / defer

- `acquire` is received
- Something to do?
  - No. Do nothing.

Event has been discarded!

- `acquire` is received
- Something to do?
  - Yes. Defers it.

Event is (eventually) handled

A Transformation Approach: The Dining Philosophers (IV)

Example system startup

```
run(N) ->
  Forks = lists:map (fun (_) -> fork:start() end, lists:
  lists:foreach
    (fun ({L,R}) -> philosopher:start(L, R) end, adjacen

adjacent([])      -> [];
adjacent([X|Xs]) -> lists:zip([X] ++ Xs, Xs ++ [X]).
```

A Transformation Approach: The Dining Philosophers (IV)

Example system startup

```
run(N) ->
  Forks = lists:map (fun (_) -> fork:start() end, lists:
  lists:foreach
    (fun ({L,R}) -> philosopher:start(L, R) end, adjacen

adjacent([])     -> [];
adjacent([X|Xs]) -> lists:zip([X] ++ Xs, Xs ++ [X]).
```

Using McErlang to verify correctness

```
> mce:start
    (#mce_opts{program=fun () -> dining:run(2) end,
               monitor=mce_mon_deadlock}).
...
*** Monitor failed
monitor error:
deadlock
```

A Transformation Approach: Algorithm (VI)

- Input data: UML-SM
- Output data: Erlang source code

A Transformation Approach: Algorithm (VI)

- Input data: UML-SM
- Output data: Erlang source code

Algorithm steps (abstractedly)

1. Store parameters of initial transition ($\mathcal{P}$)
2. Create the Erlang header (`module`, `export`, `start`)
3. Create a set of triggered events of current state ($\mathcal{E}$)
4. Iterate for each state in the UML-SM
    1. Convert `entry`, `do` activities to message passing
    2. Special case: timeout activities
    3. Iterate in the output transitions
        1. Fill a `receive` Erlang skeleton properly
    4. Convert `exit` activity to message passing

Outline

1. Motivation

2. A Transformation Approach

3. Related Work

4. Conclusions and Future Work

Related Work (I)

Automatic code generation
(multi-threaded behaviour and asynchronous communication)

Translator compiler

- C code from finite state machines with a synchronous semantics
    - PM-FORMS-03,AFLTY-ISORC-10
- Aynchronous semantics with a state table to reacts to events
    - NT-SEA-03,KNNZ-ICSE-00
- Design pattern forms implementing state machines
    - TKUY-ICRA-01
- Java thread per state-chart and Java objects to represent event queues
    - KM-TOOLS-02

Related Work (I)

Automatic code generation
(multi-threaded behaviour and asynchronous communication)

Translator compiler

- C code from finite state machines with a synchronous semantics
  - PM-FORMS-03,AFLTY-ISORC-10
- Aynchronous semantics with a state table to reacts to events
  - NT-SEA-03,KNNZ-ICSE-00
- Design pattern forms implementing state machines
  - TKUY-ICRA-01
- Java thread per state-chart and Java objects to represent event queues
  - KM-TOOLS-02

Interpreter to manage multi-threading and event queues → Erlang

Related Work (II)

Other model checkers

- Branching time model-checking using JACK
  - GLM-HASE-99
- Linear-time model checking using PROMELA
  - LMM-FAC-99
- UML class diagrams, UML-SMs and UML Communication diagrams verified using Maude LTL
  - CEC-IJSEA-12

Outline

1 Motivation

2 A Transformation Approach

3 Related Work

4 Conclusions and Future Work

Conclusions and Future Work (I)

Conclusions

- UML: standard as semi-formal modelling language
  - UML-SM: models system dynamics and its interaction
    - Modelling of concurrent and distributed systems
- Erlang: functional language
  - Good support for concurrency and distribution

Conclusions and Future Work (I)

Conclusions

- UML: standard as semi-formal modelling language
  - UML-SM: models system dynamics and its interaction
    - Modelling of concurrent and distributed systems
- Erlang: functional language
  - Good support for concurrency and distribution

UML-SM → Erlang code

Conclusions and Future Work (I)

Conclusions

- UML: standard as semi-formal modelling language
  - UML-SM: models system dynamics and its interaction
    - Modelling of concurrent and distributed systems
- Erlang: functional language
  - Good support for concurrency and distribution

UML-SM $\rightarrow$ Erlang code

Contributions

- Reduce development time
  - By automatically generating Erlang skeleton code
- Enables validation of UML-SMs at an early development stage
  - Erlang-based model checking and testing techniques

Conclusions and Future Work (I)

Conclusions

- UML: standard as semi-formal modelling language
  - UML-SM: models system dynamics and its interaction
    - Modelling of concurrent and distributed systems
- Erlang: functional language
  - Good support for concurrency and distribution

UML-SM → Erlang code

Contributions

- Reduce development time
  - By automatically generating Erlang skeleton code
- Enables validation of UML-SMs at an early development stage
  - Erlang-based model checking and testing techniques
- Also an alternative to Erlang code for “behaviour” pattern

Conclusions and Future Work (II)

Future Work

- Extend to additional UML-SM constructs
  - Preemptive UML-SM activities
  - Substates
  - Entry, exit or alternative potins
  - ...
- Apply to some real examples
- Tool support
  - Plugin for some UML CASE tool (e.g. Eclipse)

From UML State-Machine Diagrams to Erlang

Ricardo J. Rodríguez, Lars-Åke Fredlund, Ángel Herranz

🄯 **All wrongs reversed**

{rjrodriguez, lfredlund, aherranz}@fi.upm.es

Universidad Politécnica de Madrid
Madrid, Spain

September 20th, 2013

XIII Jornadas sobre Programación y Lenguajes (PROLE)
Facultad de Informática, Universidad Complutense de Madrid