A Transactional Model and Platform for Designing and Implementing Reactive Systems

Justin R. Wilson

A dissertation presented to the Graduate School of Arts and Sciences of Washington University in partial fulfillment of the requirements for the degree of Doctor of Philosophy
November 2016
Saint Louis, Missouri
Introduction[1]

Transformational Program

Input → Processing → Output

Reactive Program

Environment → Program

Input → Processing

Output

Z. Manna and A. Pnueli.


Contributions

- Formal Model - Reactive Components
- Programming Language - rc$_{go}$
- Platform
- Scheduler Evaluation
Fundamental Characteristics of Reactive Systems

Time and State
Reason about the state of the program relative to the state of the environment.

Concurrency
A program and its environment may act at the same time.

Synchronous Systems
Program and environment share a common clock.

Asynchronous Systems
Program and environment do not share a common clock (this work).
Trends

Enabling Technologies

▶ *Form Factors*
   new domains

▶ *(Wireless) Networking*
   new frontiers

Consequences

▶ *Scale*
   large systems require
different techniques

▶ *Complexity*
   systems of systems of ...
The Goal

An abstraction that makes reactive programs easy to

- design
- implement
- test
- debug
- reuse

without sacrificing performance.
Limitations of the State of the Art: Threads and Events

Hazards

- deadlock
- livelock
- memory corruption

Brittle unless developers

- identify and protect shared state
- understand call graph
Challenges

Reduce Accidental Complexity

▶ implicit atomicity
▶ eliminate uncoordinated access to shared state

Achieve Principled Composition

1. Define units and means of composition
2. “Well-formed-ness”
3. Recursive Encapsulation - units may contain other units (hierarchical reasoning)
4. Interfaces - reason about interface instead of implementation
5. Compositionality - reason about whole from parts
6. Substitutional Equivalence - substitute definition for use
Related Work

Abstract Models for Analysis

- e.g., Calculus of Communicating Systems, Algebra of Communicating Processes
- Not practical for design and implementation

Processes-Oriented Models

- e.g., Cooperating Sequential Processes, Communicating Sequential Processes, Kahn Process Networks
- Difficult to rewrite $N$ processes as one process

Asynchronous Message Passing

- e.g., Actor Model
- Weak guarantees limit compositional design and reasoning
Related Work

UNITY

- State variables, parallel assignment statements, non-deterministic selection and execution
- Lack of encapsulation/interface precludes compositionality

I/O Automata

- State variables, input/output/internal actions, pre/post conditions, signatures (interfaces)
- Named-based composition limited in depth
Approach and Contributions

Reactive Components
Formal Model

rc_go
Programming Language

type Clock component {
  counter uint;
  flag bool;
  response push (t uint);
};
...

rc_go
Platform

rc_go
Scheduler
The Reactive Components Model

- **passive push port**
- **reaction**
- **transition**
- **activate**
- **active push port**
- **precondition**
- **action**
- **transition**
- **getter**
- **passive pull port**
- **call**
- **expression**
- **state variables**
The Reactive Components Model

- **state variables**
- **reaction**
- **transition**
- **activate**
- **call**
- **getter**
- **expression**

**Components:**
- Passive push port
- Active push port
- Active pull port
- Passive pull port

**Actions:**
- Precondition
- Action

**Procedures:**
- Interrogate state variables
- Transition
- Transition
- Call

**Terms:**
- Man
- Getter
- Precondition
- Expression
The Reactive Components Model

passive push port

reaction

transition

precondition

action

transition

activate

call

getter pull port

expression

active push port

state variables
The Reactive Components Model
The Reactive Components Model

- **passive push port**
- **active pull port**
- **passive push port**
- **active pull port**
- **getter**
- **expression**
- **transition**
- **transition**
- **precondition**
- **action**
- **call**
- **activate**
- **reaction**
- **state variables**
The Reactive Components Model

- **reaction**
- **transition**
- **activate**
- **call**
- **getter**
- **expression**
- **passive**
- **push port**
- **active**
- **pull port**
- **precondition**
- **state variables**
The Reactive Components Model

- **passive push port**
- **active pull port**
- **state variables**
- **reaction**
- **transition**
- **activate**
- **action**
- **precondition**
- **call**
- **getter**
- **expression**

**interrogate state variables**
The Reactive Components Model

- Passive
  - Push port

- Reactive
  - Reaction
  - Transition
  - Activate

- Action
  - Transition
  - Precondition
  - Call

- Getter
  - Pull port

- State variables

- Expressions
Transitions

Conceptually, a *transition* is an atomic parallel assignment statement:

\[
\text{var1, var2, ...} \coloneqq \text{expr1, expr2, ...}
\]

*Mutable Phase* \(\coloneqq\) *Immutable Phase*

Transactions

A *transaction* is a set of transitively linked transitions.

(Logical) Execution

Repeatedly execute transactions (order is not determined).

Fairness

All transactions are selected an infinite number of times.
Clock System Example

**Sampler**
- Flag: false
- Flag := false
- Request()
- Flag := true
- Response(t)
- Flag := false

**Clock**
- Flag: false
- Request()
- Flag := true
- Response(t)
- Flag := true
- Counter()
- Flag := false

**Counter**
- Count: 0
- True
- Count := count + 1
- Count

Counter()
Clock System Example

Sampler
flag (false)
flag = false
request()
flag := true
response(t)
flag := false

Clock
flag (false)
request()
flag := true
response(t)
flag := false

clock

Counter
count (0)
ture
count := count + 1
count

counter()
Clock System Example 🕒

**Sampler**
- `flag (false)`
- `flag = false`
- `request()`
- `flag := true`
- `response(t)`
- `flag := false`

**Clock**
- `flag (false)`
- `request()`
- `flag := true`
- `response(t)`
- `flag := false`

**Counter**
- `count (0)`
- `true`
- `count := count + 1`
- `counter()`
- `count`
Clock System Example

Sampler
flag (false)
flag := false
request()
flag := true
response(t)
flag := false

Clock
flag (false)
flag := true
request()
flag := true
response(t)
flag := false

Counter
count (0)
ture
count := count + 1
counter()
count

counter()
Clock System Example

**Sampler**
- flag (false)
  - flag = false
  - request()
  - flag := true
  - response(t)
  - flag := false

**Clock**
- flag (false)
  - request()
  - flag := true
  - response(t)
  - flag := false

**Counter**
- count (0)
  - true
  - count := count + 1
  - counter()
  - count

```plaintext
flag := false
request()
flag := true
response(t)
flag := false
flag := true
request()
flag := true
response(t)
count := count + 1
counter()
count
```
Simplification of Clock System

Reactive components allow principled composition.

\[
\begin{align*}
\text{s\_flag (false)} & \quad \text{c\_flag (false)} & \quad \text{count (0)} \\
\text{s\_flag = false} & \quad \text{c\_flag = true} & \quad \text{true} \\
\text{s\_flag := true} & \quad \text{c\_flag := true} & \quad \text{count := count + 1} \\
\text{s\_flag := false} & \quad \text{c\_flag := false} & \quad \text{count}
\end{align*}
\]
Platform Requirements

▶ Strict enforcement of the model
▶ Reference semantics and linked data structures
▶ Efficient inter-component communication

- $\text{rc}_{\text{go}}$ is based on Go
  - Go is like C with methods, interfaces, and garbage collection
- $\text{rc}_{\text{java}}$ would be different

**Static System Assumption:**
Number and configuration of components is fixed.
Model → Language
rcg0: Keywords for Defining Components

```go
type Clock component {
    counter uint;
    flag bool;
    response push (t uint);  // active
    counter pull () uint;    // active
};

init (c *Clock) Initialize () {.}
action (c $const *Clock) Respond (this.flag) {.}
reaction (c $const *Clock) Request () {.}
getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
```
rc_go: Keywords for Defining Components

type Clock component {
    counter uint;
    flag bool;
    response push (t uint);  // active
    counter pull () uint;    // active
};

init (c *Clock) Initialize () {.}
action (c $const *Clock) Respond (this.flag) {.}
reaction (c $const *Clock) Request () {.}
getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
rcgo: Keywords for Defining Components

type Clock component {
  counter uint;
  flag bool;
  response push (t uint); // active
  counter pull () uint; // active
};

init (c *Clock) Initialize () {.}
action (c $const *Clock) Respond (this.flag) {.}
reaction (c $const *Clock) Request () {.}
getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
  s.sampler.Request -> s.clock.Request;
  s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
```go
package rce

type Clock component {
    counter uint;
    flag bool;
    response push (t uint); // active
    counter pull () uint; // active
}

init (c *Clock) Initialize () {.}
action (c $const *Clock) Respond (this.flag) {.}
reaction (c $const *Clock) Request () {.}
getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
```
type Clock component {
    counter uint;
    flag bool;
    response push (t uint); // active
    counter pull () uint; // active
};

init (c *Clock) Initialize () {.}
action (c $const *Clock) Respond (this.flag) {.}
reaction (c $const *Clock) Request () {.}
getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
rcgo: Keywords for Defining Components

type Clock component {
    counter uint;
    flag bool;
    response push (t uint); // active
    counter pull () uint; // active
};

init (c *Clock) Initialize () {.}
action (c $const *Clock) Respond (this.flag) {.}
reaction (c $const *Clock) Request () {.}
getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
rc go: Keywords for Defining Components

type Clock component {
    counter uint;
    flag bool;
    response push (t uint); // active
    counter pull () uint; // active
};

init (c *Clock) Initialize () {.}

action (c $const *Clock) Respond (this.flag) {.}

reaction (c $const *Clock) Request () {.}

getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
rcgo: Keywords for Defining Components

type Clock component {
    counter uint;
    flag bool;
    response push (t uint); // active
    counter pull () uint; // active
};

init (c *Clock) Initialize () {
}
action (c $const *Clock) Respond (this.flag) {
}
reaction (c $const *Clock) Request () {
}
getter (c $const *Counter) Counter () uint {
}
bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
rcgo: Keywords for Defining Components

type Clock component {
  counter uint;
  flag bool;
  response push (t uint); // active
  counter pull () uint; // active
};

init (c *Clock) Initialize () {.}
action (c $const *Clock) Respond (this.flag) {.}
reaction (c $const *Clock) Request () {.}
getter (c $const *Counter) Counter () uint {.}

bind (s *System) BindAll {
  s.sampler.Request -> s.clock.Request;
  s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
rcgo: Keywords for Defining Components

type Clock component {
    counter uint;
    flag bool;
    response push (t uint); // active
    counter pull () uint; // active
};

init (c *Clock) Initialize () {.
action (c $const *Clock) Respond (this.flag) {.
reaction (c $const *Clock) Request () {.
getter (c $const *Counter) Counter () uint {.

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
rcgo: Keywords for Defining Components

```go
type Clock component {
    counter uint;
    flag bool;
    response push (t uint); // active
    counter pull () uint; // active
};

init (c *Clock) Initialize () {...
action (c $const *Clock) Respond (this.flag) {...
reaction (c $const *Clock) Request () {...
getter (c $const *Counter) Counter () uint {...

bind (s *System) BindAll {
    s.sampler.Request -> s.clock.Request;
    s.clock.counter <- s.counter.Counter;
}

instance s System Initialize ();
```
Model → Language

- Model elements:
  - Reaction
  - Transition
  - Activate
  - Precondition
  - Call
  - Getter
  - Expression

- Language elements:
  - State variables
  - Active push port
  - Passive push port
  - Active pull port
  - Passive pull port

Diagram shows the flow between model and language components, illustrating how reactions, transitions, and activations are connected through various ports and conditions.
rcgo: The Immutable Phase

Immutability

- `const` - lvalue and lvalues derived through indirection are immutable.
- `$const` - lvalues derived through indirection are immutable.

```
action (c $const *Clock) Respond (this.flag) {
reaction (c $const *Clock) Request () {
getter (c $const *Counter) Counter () uint {
```

Actions/reactions start in immutable phase.
Getters only called in immutable phase.
The Mutable Phase (Activations)

```go
action (this $const *QueueProcessor) _dequeue (!this.queue.Empty()) {

    var m $const *MetaData =
        this.lookupMetadata(this.queue.Front())

    activate Out (this.queue.Front(), m) {
        this.queue.Pop()
    }
}
```
rcgo: The Mutable Phase (Activations)

```go
action (this $const *QueueProcessor) _dequeue (!this.queue.Empty()) {

    var m $const *MetaData =
        this.lookupMetadata(this.queue.Front())

    activate Out (this.queue.Front(), m) {
        this.queue.Pop()
    }
}
```

list of push ports to activate
rcgo: The Mutable Phase (Activations)

```go
action (this $const *QueueProcessor) _dequeue (!this.queue.Empty()) {

    var m $const *MetaData =
        this.lookupMetadata(this.queue.Front())

    activate Out (this.queue.Front(), m) {
        this.queue.Pop()
    }
}
```

this is not $const in activate body
rcgo: The Mutable Phase (Activations)

```go
action (this $const *QueueProcessor) _dequeue (!this.queue.Empty()) {
    var m $const *MetaData =
    this.lookupMetadata(this.queue.Front())

    activate Out (this.queue.Front(), m) {
        this.queue.Pop()
    }
}
```

activate terminates action/reaction
rc_go: Reference Semantics

```go
reaction (this $const *C) R (w *W) {
    if (interesting(w)) {
        activate {
            this.w = w // Save for later
        }
    }
}
```

Compositionality is lost!

Safe Reference Semantics

- component state is private
- `$foreign` attribute for pointers (read, write, save)

```go
reaction (this $const *C) R (w $foreign *W) {
    if (interesting(w)) {
        var w2 *W = w.DeepCopy()
        activate {
            this.w = w2 // Save for later
        }
    }
}
```
rcgo: Move Semantics

action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...
    }
    activate port (w) {
        ...
    }
}

reaction (this $const *C2) R (w $foreign *heap W) {
    var y *heap W = move(w)
    // Or
    var z *W = merge(w)
    ...
}
rcgo: Move Semantics

```go
action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...
    }
    activate port (w) {
        ...
    }
}
```

```go
reaction (this $const *C2) R (w $foreign *heap W) {
    var y *heap W = move(w)
    // Or
    var z *W = merge(w)
    ...
}
```

allocate a new heap
rcgo: Move Semantics

```go
action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...
    }
    activate port (w) {
        ...
    }
}

reaction (this $const *C2) R (w $foreign *heap W) {
    var y *heap W = move(w)
    // Or
    var z *W = merge(w)
    ...
}
```

- Direct allocations to `w`
- `change` takes ownership and unboxed content of `w`
- `move` and `merge` will return `nil`
- All variables outside of `change` are foreign
rc.go: Move Semantics

```go
action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...
    }
    activate port (w) {
        ...
    }
}

reaction (this $const *C2) R (w $foreign *heap W) {
    var y *heap W = move(w)
    // Or
    var z *W = merge(w)
    ...
}
```

x is a *W
rcgo: Move Semantics

```go
action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...
    }
    activate port (w) {
        ...
    }
}

reaction (this $const *C2) R (w $foreign *heap W) {
    var y *heap W = move(w)
    // Or
    var z *W = merge(w)
    ...
}
```

all variables outside of change are foreign
rcgo: Move Semantics

```go
action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...  
    }
    activate port (w) {
        ...
    }
}

reaction (this $const *C2) R (w $foreign *heap W) {
    var y *heap W = move(w)
    // Or
    var z *W = merge(w)
    ...
}
```

take ownership
rcgo: Move Semantics

```go
action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...
    }
    activate port (w) {
        ...
    }
}

reaction (this $const *C2) R (w $foreign *heap W) {
    var y *heap W = move(w)
    // Or
    var z *W = merge(w)
    ...
}
```

take ownership and unbox
rcgo: Move Semantics

```go
action (this $const *C1) A (precondition) {
    var w *heap W = new(heap W)
    change (w, x) {
        x.foo = 3
        ...
    }
    activate port (w) {
        ...
    }
}

reaction (this $const *C2) R (w $foreign * heap W) {
    var y * heap W = move(w)
    // Or
    var z * W = merge(w)
    ...
}
```

content behind w may be gone (change, move, merge will return nil)
Enforcing Sound Composition: Non-determinism

Problem
Suppose there are two transitions $t_1$ and $t_2$:

$t_1 : x, y := 1, 2$
$t_2 : x, z := 3, 4$

Composing them results in a non-deterministic transaction!

$t_1 || t_2 : x, y, x, z := 1, 2, 3, 4$

Challenge
Reject programs with non-deterministic transactions.

Solution
Let component instances be a proxy for their state variables.
Enforcing Sound Composition: The Transaction Graph

Transaction Graph follow activations through code and bindings
Action/Reaction → Activation → Port → Reaction → ...

* indicates instance is mutated
Enforcing Sound Composition: The Transaction Graph

Transaction Graph follow activations through code and bindings

Action/Reaction $\rightarrow$ Activation $\rightarrow$ Port $\rightarrow$ Reaction $\rightarrow$ ...

* indicates instance is mutated
Enforcing Sound Composition: The Transaction Graph

Transaction Graph follow activations through code and bindings
Action/Reaction $\rightarrow$ Activation $\rightarrow$ Port $\rightarrow$ Reaction $\rightarrow$ ...

* indicates instance is mutated
Enforcing Sound Composition: The Transaction Graph

Transaction Graph follow activations through code and bindings
Action/Reaction $\rightarrow$ Activation $\rightarrow$ Port $\rightarrow$ Reaction $\rightarrow$ ...

* indicates instance is mutated
Enforcing Sound Composition: The Transaction Graph

Transaction Graph follow activations through code and bindings
Action/Reaction $\rightarrow$ Activation $\rightarrow$ Port $\rightarrow$ Reaction $\rightarrow$ ...

* indicates instance is mutated
Enforcing Sound Composition: The Transaction Graph

Transaction Graph follow activations through code and bindings

Action/Reaction → Activation → Port → Reaction → ...

* indicates instance is mutated
Enforcing Sound Composition: Instance Sets

**Instance Set** self and union of instances of children

- Activations at same level are *mutually exclusive*
- Ports and reactions are activated as a group
Enforcing Sound Composition: Instance Sets

Instance Set: self and union of instances of children

- Activations at same level are *mutually exclusive*
- Ports and reactions are activated as a group
Platform Details

Synchronized Two-Phase Calling Convention

▶ Transactions can be executed efficiently

Heaps

▶ Each component has one (or more) heaps
▶ Scoping and one level of indirection enforces encapsulation
▶ Encapsulation enables parallel garbage collection

I/O

▶ Extend language with host OS types, e.g., file descriptors
▶ Non-blocking I/O for fairness
▶ Timer + UDP → Prototype SNTP client
The Scheduling Problem

Fairly pick an idle transaction that is safe* with respect to the set of active transactions.
Scheduler Design Criteria

When are preconditions evaluated?
▶ before execution (*lazy*) or after execution (*eager*)

How is safety checked?
▶ every time (*oblivious*) or *a priori* (*knowledgeable*)

How are race conditions handled?
▶ avoid (*cautious*) or recover (*speculative*)

Is the execution of a transaction physically atomic?
▶ yes (*non-preemptive*) or no (*preemptive*)
Scheduler Implementation

Instance Scheduler

Queue of Instances

Thread 1

Thread 2

Execute all transactions in instance

Partitioned Scheduler

Thread 1

Thread 2

Thread 3

Thread 4

Termination Ring

List of Transactions

Idle

Idle

Exec

Wait

Wait

Idle
Scheduler Evaluation: AsyncClock

- Machine: 2 cores
- Instance/Partitioned: 2 threads
- Thread: 3 threads

*Thread suffers excessive context switches*

Instance/Partitioned are viable multi-threaded event schedulers!
Scheduler Evaluation: SyncClock

- Machine: 2 cores
- Instance/Partitioned: 2 threads
- Thread: 2 threads

Compilation beats interpretation!
Conclusions and Future Work

Conclusions

1. Reactive component model enables principled composition/decomposition of reactive systems.
2. Reactive component semantics can be checked efficiently.
3. Empirical results suggest that reactive components could compete with optimized multi-threaded approaches.

Future Work

- Extend model for introduction and dynamic composition
- Explore additional scheduler classes and algorithms
Questions?
Bonus Slides
Synchronized Two-Phase Calling Convention

To execute a transaction:

1. Depth-first traversal of transaction graph (immutable phase)

2. Execute continuation of activate statements (mutable phase)

*Maps well to existing architectures!*
Heaps

Problem
Move state from one component to another efficiently

Challenge
Maintain isolation of components (encapsulation)
Heaps: Approach

- Run-time maintains a stack of heaps
- Top heap services allocation requests
- Change scoping rules enforce encapsulation
- Encapsulation enables parallel garbage collection

One level of indirection for move and merge

* heap W

heap link

heap (W root)

0x12345678

0x90ABCDEF

heap object

0x56781234
Heaps: Approach

- Run-time maintains a stack of heaps
- Top heap services allocation requests
- Change scoping rules enforce encapsulation
- Encapsulation enables parallel garbage collection

One level of indirection for move and merge:

* heap W
  
  heap object

0x12345678 → 0x90ABCDEF → heap (W root)

0x56781234
I/O

Problem
Interact with external systems (through operating system)

Challenges

▶ Maintain fairness
▶ Avoid accidental sharing of state

Approach

1. Introduce FileDescriptor type.
2. Wrap FileDescriptors in reactive components.
3. Introduce readable and writable predicates.
4. Introduce functions for manipulating FileDescriptors. All operations are non-blocking to ensure fairness.
Example: Simple Network Time Protocol (SNTP)
Example: UdpParticipant

type UdpParticipant component {
    fd FileDescriptor;
    outQueue Queue;
    inQueue Queue;
    Receive push (msg $foreign UdpMessage);
}

init (this *UdpParticipant) Init () {
    this.fd = udp_socket()
}
Example: UdpParticipant

type UdpParticipant component {
    fd FileDescriptor;
    outQueue Queue;
    inQueue Queue;
    Receive push (msg $foreign UdpMessage);
}

init (this *UdpParticipant) Init () {
    this.fd = udp_socket()
}

wrap a FileDescriptor
Example: UdpParticipant

type UdpParticipant component {
  fd FileDescriptor;
  outQueue Queue;
  inQueue Queue;
  Receive push (msg $foreign UdpMessage);
}

init (this *UdpParticipant) init () {
  this.fd = udp_socket();
}
Example: UdpParticipant

reaction (this $const * UdpParticipant)
Send (msg $foreign UdpMessage) {
    var m UdpMessage = msg.Copy ();
    activate {
        this.outQueue.Push (m);
    }
}

action (this $const * UdpParticipant)
_send (!this.outQueue.Empty () && writable (this.fd)) {
    activate {
        var m UdpMessage = this.outQueue.Front ();
        sendto (this.fd, m.host, m.port, m.msg);
        this.outQueue.Pop ();
    }
}
Example: UdpParticipant

reaction (this $const * UdpParticipant)
Send (msg $foreign UdpMessage) {
    var m UdpMessage = msg.Copy ();
    activate {
        this.outQueue.Push (m);
    };
}

action (this $const * UdpParticipant)
_send (!this.outQueue.Empty () && writable (this.fd)) {
    activate {
        var m UdpMessage = this.outQueue.Front ();
        sendto (this.fd, m.host, m.port, m.msg);
        this.outQueue.Pop ();
    }
}
Example: UdpParticipant

reaction (this $const * UdpParticipant)
Send (msg $foreign UdpMessage) {
    var m UdpMessage = msg.Copy ();
    activate {
        this.outQueue.Push (m);
    };
}

action (this $const * UdpParticipant)
_send ( !this.outQueue.Empty () && writable (this.fd)) {
    activate {
        var m UdpMessage = this.outQueue.Front ();
        sendto (this.fd, m.host, m.port, m.msg);
        this.outQueue.Pop ();
    }
}