Reactive Method Dispatch for Context-Oriented Programming

Engineer Bainomugisha

Ph.D. Public Defence
12th December 2012

Promoter: Prof. Dr. Wolfgang De Meuter

Software Languages Lab.
Vrije Universiteit Brussel, Belgium

Roadmap

Motivation

Reactive context-aware applications

Characteristics

Issues

Programming Language Approach

Language Requirements

State of the Art

The ICoDE Model

The Flute Language

Semantics and Validation

iScheme

An Executable Semantics for Flute

Validation

Motivation

Motivation

Reactive context-aware applications

Characteristics

Issues

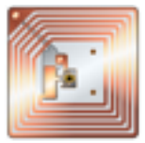
Sensor Equipment Mobile Devices

GPS

Camera

Compass

Accelerometer



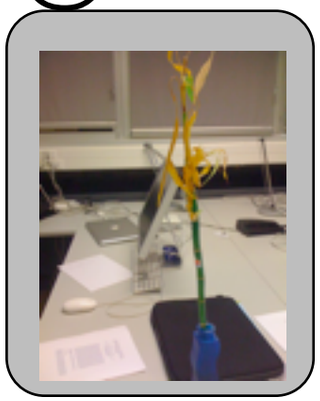
RFID

Sensor Equipment Mobile Devices

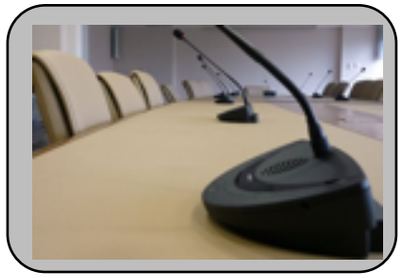
Reactive
Context-aware
Applications

Reactive Context-aware Applications

@Office



@Conference room

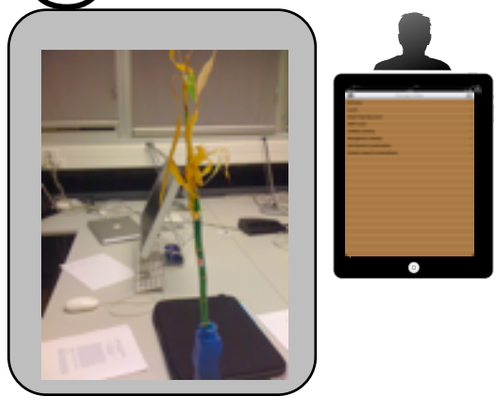


@Printer room

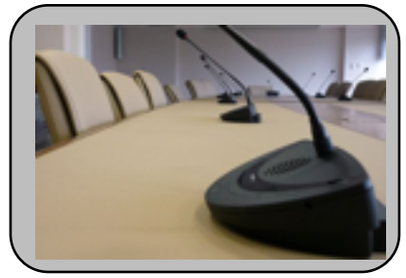
@Car

Reactive Context-aware Applications

@Office



@Conference room

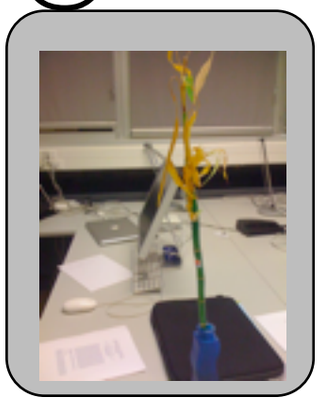


@Printer room

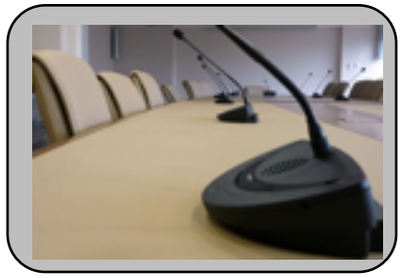
@Car

Reactive Context-aware Applications

@Office



@Conference room

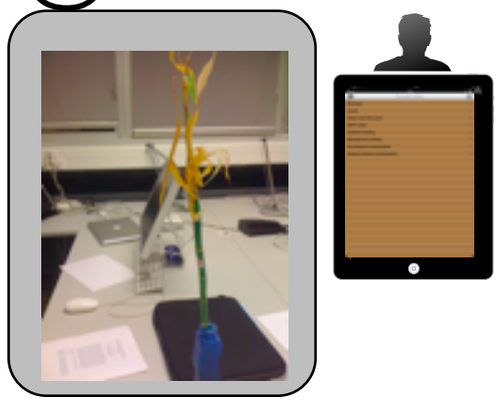


@Printer room

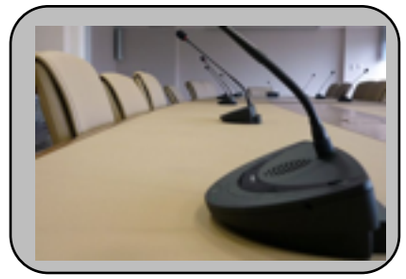
@Car

Reactive Context-aware Applications

@Office



@Conference room

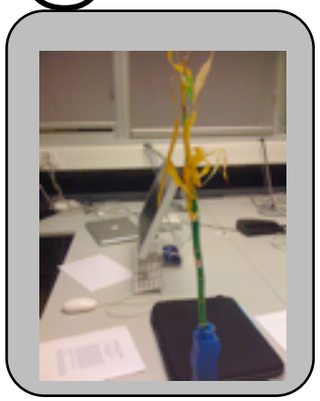


@Printer room

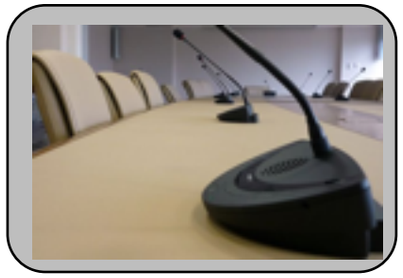
@Car

Reactive Context-aware Applications

@Office



@Conference room

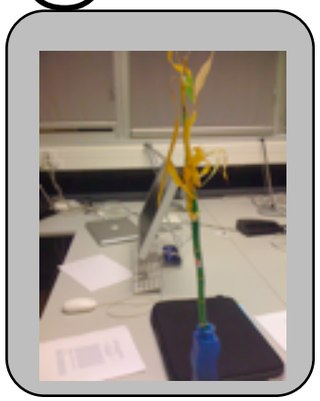


@Printer room

@Car

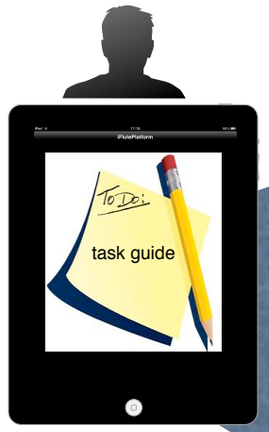
Reactive Context-aware Applications

@Office



@Printer room

@Conference room



to-call

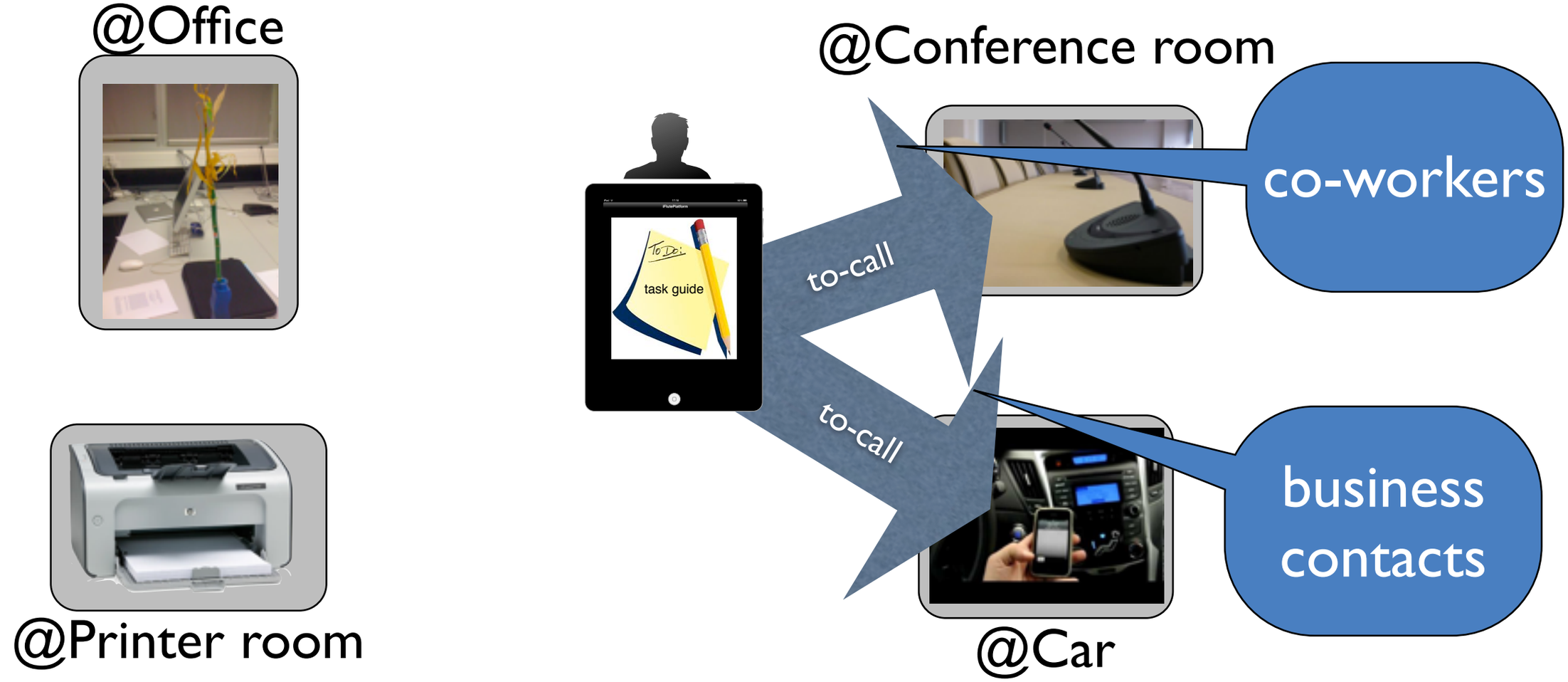
co-workers

to-call

business contacts

@Car

Reactive Context-aware Applications



Fundamental Characteristics:

1. **Context-constrained** executions.
2. **Sudden** interruptions.
3. **Prompt** adaptability.

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Context-independent

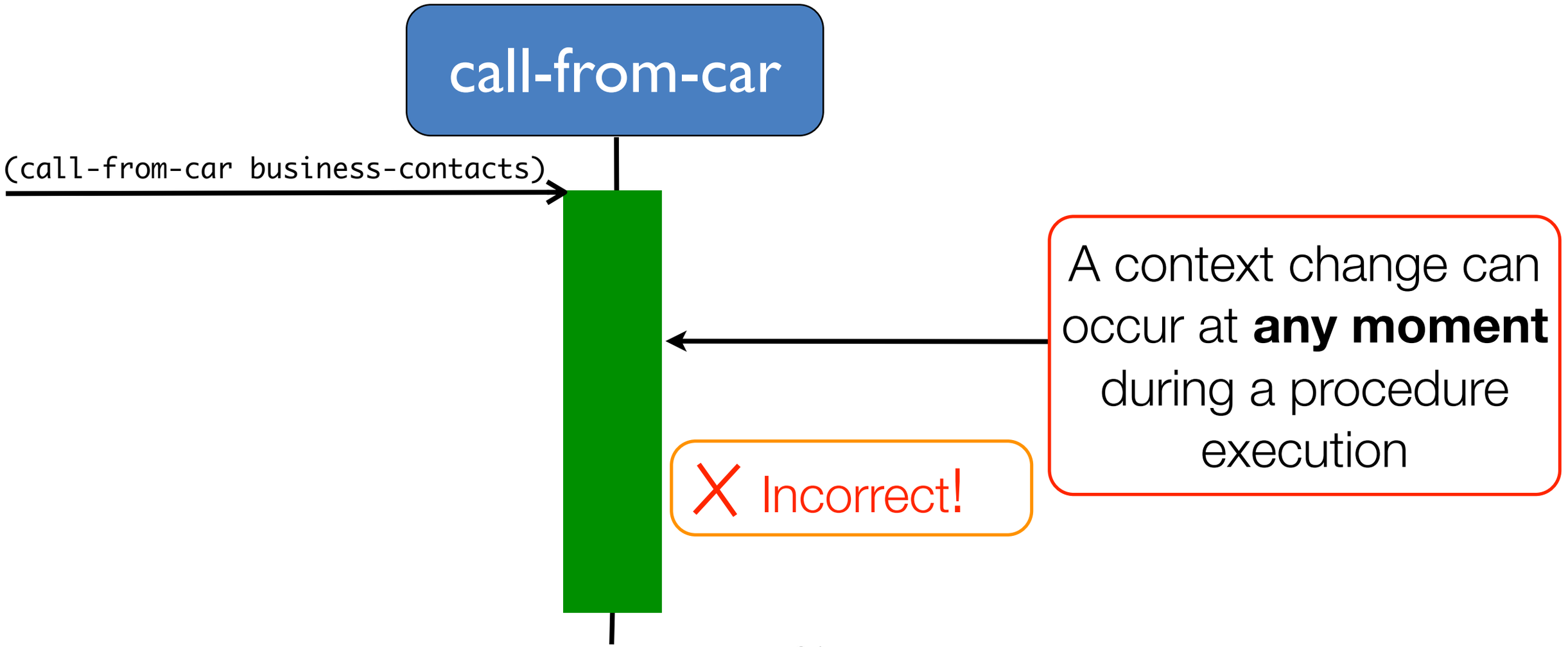
Programming Reactive Context-aware Applications

```
(if (is-in-car? location)
    (call-from-car business-contacts))
```

Programming Reactive Context-aware Applications

```
(if (is-in-car? location)
    (call-from-car business-contacts))
```

Not enough!



Programming Reactive Context-aware Applications

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list is empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (call-from-car (cdr contacts-list)))))))
```

Fundamental Research Questions:

1. How to **constrain an entire procedure execution** to the right context?

2. What to do when a **context change occurs** in the **middle of an ongoing execution**?

Manual Checks, Coroutines, Continuations, ...

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (if (is-in-car? location)
4       (show "calling business contacts")
5       (save/suspend))
6     (if (null? contacts-list)
7       (show "business contacts list is empty")
8       (let ((contact (car contacts-list)))
9         (if (is-in-car? location)
10          (show (contact-name contact))
11          (save/suspend))
12        (if (is-in-car? location)
13          (dial (phone-number contact))
14          (save/suspend))
15        (if (is-in-car? location)
16          (turn-on-phone-speaker)
17          (save/suspend))
18        (if (is-in-car? location)
19          (connect-to-car-speakers)
20          (save/suspend))
21        (if (dial-next-contact? (user-response))
22          (call-from-car (cdr contacts-list))))))))
```

Repetitive
context checks

Resumptions?

Context management?

Manual Checks, Coroutines, Continuations, ...

```
1 (define call-from-car
2   (lambda (contacts-list)
3     (if (is-in-car? location)
4       (show "calling business co
5       (save/suspend))
6     (if (null? contacts-list
7       (show "business cont                pty")
8       (let ((contact (c                  )))
9         (if (is-in-c
10          (show (co                    act))
11          (save/s
12        (if (i                         ion)
13          (di                        er contact))
14
15                                     location)
16                                     one-speaker)
17                                     spend))
18                                     n-car? location)
1                                      nect-to-car-speakers)
                                       ave/suspend))
2                                  if (dial-next-contact? (user-response))
22                                 (call-from-car (cdr contacts-list))))))
```

Need for a dedicated software technology

Repetitive
context checks

Resumptions?

Context management?

Reactive Method Dispatch for Context-Oriented Programming

Programming Language Approach

Language Requirements

State of the Art

The ICoDE Model

The Flute Language

Programming Language Requirements

```
(is-in-car? location)
```

```
1 (define to-call
2   (lambda (contacts-list)
3     (show "calling business contacts")
4     (if (null? contacts-list)
5       (show "business contacts list is empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-car-speakers)
11        (if (dial-next-contact? (user-response))
12          (to-call (cdr contacts-list)))))))
```

```
(is-in-conference-room? location)
```

```
1 (define to-call
2   (lambda (contacts-list)
3     (show "calling co-workers contacts")
4     (if (null? contacts-list)
5       (show "co-workers contacts list is empty")
6       (let ((contact (car contacts-list)))
7         (show (contact-name contact))
8         (dial (phone-number contact))
9         (turn-on-phone-speaker)
10        (connect-to-conference-equipment)
11        (if (dial-next-contact? (user-response))
12          (to-call (cdr contacts-list)))))))
```

Language

Requirements:

R.1 Chained Context Reactions

R.2 Contextual Dispatch

R.3 Reactive Dispatch

R.4 Context-dependent Interruptions

R.5 Context-dependent Resumptions

R.6 Reactive Scope Management

A Survey of the State of the Art

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming (COP) Languages						
ContextL and other Layer-based Languages	✗	✗	✗	✓	✗	✗
Lambic	✗	✗	✗	✓	✗	✗
Ambience	✗	✗	✗	✓	✗	✗
Contextual Values	✗	✗	✗	✓ variable	✗	✓ scoped construct
Functional Reactive Programming (FRP) Languages						
FrTime and its Siblings	✓ behaviours and events	✗	✗	✗	✓ same function re-evaluation	✗
Reactive SML	✗	✓ explicit	✓ explicit	✗	✗	✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explicit	✓ explicit	✗	✗	✗
First-class Continuations	✗	✓ explicit	✓ explicit	✗	✗	✗
Threads (Cooperative)	✗	✓ explicit	✓ explicit	✗	✗	✗
Other Programming Language Facilities						
Guards	✗	✓ blocking at the beginning	✓ from the beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

A Survey of the State of the Art

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming (COP) Languages						
ContextL and other Layer-based Languages	✗	✗	✗	✓	✗	✗
Lambic	✗	✗	✗	✓	✗	✗
Ambience	✗	✗	✗	✓	✗	✗
Contextual Values	✗	✗	✗	✓ variable	✗	✓ scoped construct
Functional Reactive Programming (FRP) Languages						
FrTime and its Siblings	✓ behaviours and events	✗	✗	✗	✓ same function re-evaluation	✗
Reactive SML	✗	✓ explicit	✓ explicit	✗	✗	✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explicit	✓ explicit	✗	✗	✗
First-class Continuations	✗	✓ explicit	✓ explicit	✗	✗	✗
Threads (Cooperative)	✗	✓ explicit	✓ explicit	✗	✗	✗
Other Programming Language Facilities						
Guards	✗	✓ blocking at the beginning	✓ from the beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

A Survey of the State of the Art

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Man…
Context-Oriented Programming (COP) Languages						
ContextL and other Layer-based Languages	✗	✗	✗	✓	✗	
Lambic	✗	✗	✗	✓		✗
Ambience	✗	✗	✗	✓		✗
Contextual Values	✗	✗	✗	✓ varia…		✓ scoped construct
Functional Reactive Programming (FRP) Languages						
FrTime and its Siblings	✓ behaviours and events	✗	✗	✗	✓ same function re-evaluation	✗
Reactive SML	✗	✓ explicit	✓ explicit	✗	✗	✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explicit	✓ explicit	✗	✗	✗
First-class Continuations	✗	✓ explicit	✓ explicit	✗	✗	✗
Threads (Cooperative)	✗	✓ explicit	✓ explicit	✗	✗	✗
Other Programming Language Facilities						
Guards	✗	✓ blocking at the beginning	✓ from the beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

Explicit Layer and Context Activation, …

A Survey of the State of the Art

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming (COP) Languages						
ContextL and other Layer-based Languages	✗	✗	✗	✓	✗	✗
Lambic	✗	✗	✗	✓	✗	✗
Ambience	✗	✗	✗	✓	✗	✗
Contextual Values	✗	✗	✗	✓ variable	✗	✓ scoped construct
Functional Reactive Programming (FRP) Languages						
FrTime and its Siblings	✓ behaviours and events	✗	✗	✗	✓ same function re-evaluation	✗
Reactive SML	✗	✓ explicit	✓ explicit	✗	✗	✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explicit	✓ explicit	✗	✗	✗
First-class Continuations	✗	✓ explicit	✓ explicit	✗	✗	✗
Threads (Cooperative)	✗	✓ explicit	✓ explicit	✗	✗	✗
Other Programming Language Facilities						
Guards	✗	✓ blocking at the beginning	✓ from the beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

Bainomugisha, E. *et.al.* (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

A Survey of the State of the Art

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming (COP) Languages						
ContextL and other Layer-based Languages	✗	✗	✗	✓	✗	✗
Lambic	✗	✗	✗	✓	✗	✗
Ambience	✗	✗	✗	✓	✗	✗
Contextual Values	✗	✗	✗	✓ variable	✗	✗
Functional Reactive Programming (FRP) Languages						
FrTime and its Siblings	✓ behaviours and events	✗	✗	✗	✓ same f… ev…	
Reactive SML	✗	✓ explicit	✓ explicit	✗	✗	✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explicit	✓ explicit	✗	✗	✗
First-class Continuations	✗	✓ explicit	✓ explicit	✗	✗	✗
Threads (Cooperative)	✗	✓ explicit	✓ explicit	✗	✗	✗
Other Programming Language Facilities						
Guards	✗	✓ blocking at the beginning	✓ from the beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

No Contextual Dispatch, …

Bainomugisha, E. *et.al.* (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

A Survey of the State of the Art

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming (COP) Languages						
ContextL and other Layer-based Languages	✗	✗	✗	✓	✗	✗
Lambic	✗	✗	✗	✓	✗	✗
Ambience	✗	✗	✗	✓	✗	✗
Contextual Values	✗	✗	✗	✓ variable	✗	✓ scoped construct
Functional Reactive Programming (FRP) Languages						
FrTime and its Siblings	✓ behaviours and events	✗	✗	✗	✓ same function re-evaluation	✗
Reactive SML	✗	✓ explicit	✓ explicit	✗	✗	✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explicit	✓ explicit	✗	✗	✗
First-class Continuations	✗	✓ explicit	✓ explicit	✗	✗	✗
Threads (Cooperative)	✗	✓ explicit	✓ explicit	✗	✗	✗
Other Programming Language Facilities						
Guards	✗	✓ blocking at the beginning	✓ from the beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

Bainomugisha, E. *et.al.* (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

A Survey of the State of the Art

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming (COP) Languages						
ContextL and other Layer-based Languages	✗	✗	✗	✓	✗	✗
Lambic	✗	✗	✗	✓	✗	✗
Ambience	✗	✗	✗	✓		✗
Contextual Values	✗	✗	✗	✓ variab		✓ scoped construct
Functional Reactive Programming (FRP) Languages						
FrTime and its Siblings	✓ behaviours and events	✗	✗			✗
Reactive SML	✗	✓ explicit	✓ exp			✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explicit	✓ exp		✗	✗
First-class Continuations	✗	✓ explicit	✓ explicit		✗	✗
Threads (Cooperative)	✗	✓ explicit	✓ explicit	✗	✗	✗
Other Programming Language Facilities						
Guards	✗	✓ blocking at the beginning	✓ from the beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

Repetitive Context Checks, Manual Resumptions, …

Bainomugisha, E. *et.al.* (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

There is a Need for a New COP Paradigm

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming Languages						
ContextL and oth Languages						✖
Lambic						✖
Ambience						✖
Contextual Values						✔ scoped construct
Functional Re						
FrTime and its Siblings	✔ events	✖	✖	✖	✔ evaluation	✖
Reactive SML	✖	✔ explic	✔ explicit	✖	✖	✖
Advanced Control Flow Constructs						
Coroutines	✖	✔ explic	✔ explicit	✖	✖	✖
First-class Continuation						✖
Threads (Cooperative)						✖
Other Programmin						
Guards	✖	✔ beginning	✔ beginning	✖	✖	✖
Assertions	✖	✔ explicit abort	✖	✖	✖	✖
Invariants	✖	✔ only at the beginning and the end	✖	✖	✖	✖

Reactive Programming
+
Context-Oriented Programming
+
Implicit Interruptions

Reactive Method Dispatch for Context-Oriented Programming

Bainomugisha, E. *et.al.* (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

There is a Need for a New COP Paradigm

Bainomugisha, E. *et.al.* (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

	Chained Context Reactions	Context-dependent Interruptions	Context-dependent Resumptions	Contextual Dispatch	Reactive Dispatch	Reactive Scope Management
Context-Oriented Programming Languages						
ContextL and oth Languages						✗
Lambic						✗
Ambience						✗
Contextual Values						✓ scoped construct
Functional Re						
FrTime and its Siblings	✓ events	✗	✗	✗	✓ evaluation	✗
Reactive SML	✗	✓ explic	✓ explicit	✗	✗	✗
Advanced Control Flow Constructs						
Coroutines	✗	✓ explic	✓ explicit	✗	✗	✗
First-class Continuation						✗
Threads (Cooperative)						✗
Other Programmin						
Guards	✗	✓ beginning	✓ beginning	✗	✗	✗
Assertions	✗	✓ explicit abort	✗	✗	✗	✗
Invariants	✗	✓ only at the beginning and the end	✗	✗	✗	✗

Reactive Programming
+
Context-Oriented Programming
+
Implicit Interruptions

Reactive Method Dispatch for Context-Oriented Programming

Reactive Method Dispatch for Context-Oriented Programming

Programming Language Approach

Language Requirements

State of the Art

The ICoDE Model

The Flute Language

The ICoDE Model: Design Space

Language Property	Design Considerations
Property #1 Predicated procedures	D.C #1.1 Associating a context predicate with a procedure - As part of procedure definition - Outside the procedure definition
	D.C #1.2 Propagation of context predicates - Dynamic context predicate propagation - Lexical context predicate propagation - No propagation of context predicates
	D.C #1.3 Grouping of related predicated procedures
Property #2 Representation of context as reactive values	
Property #3 Reactive dispatching	D.C #3.1 Dealing with ambiguous context predicates
	D.C #3.2 Handling return values
Property #4 Interruptible executions	D.C #4.1 Implicit interruptions
	D.C #4.2 Explicit interruptions
Property #5 Resumable executions	D.C #5.1 Proactive resumptions
	D.C #5.2 Event-driven resumptions
Property #6 Scoped state changes	

Bainomugisha, E. *et.al*. (2012). Interruptible Context-dependent Executions: A Fresh Look at Programming Context-aware Applications. SPLASH/*OnWard! 2012*

Reactive Method Dispatch for Context-Oriented Programming

Programming Language Approach

Language Requirements

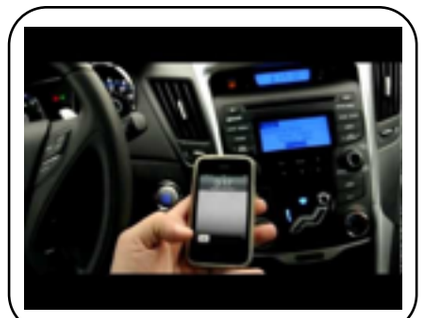
State of the Art

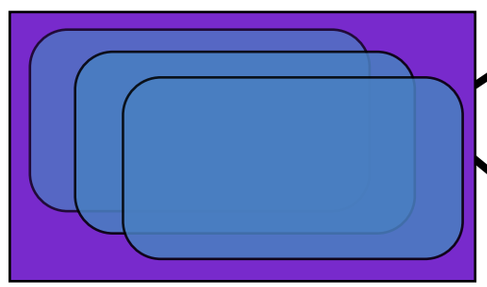
The ICoDE Model

The Flute Language

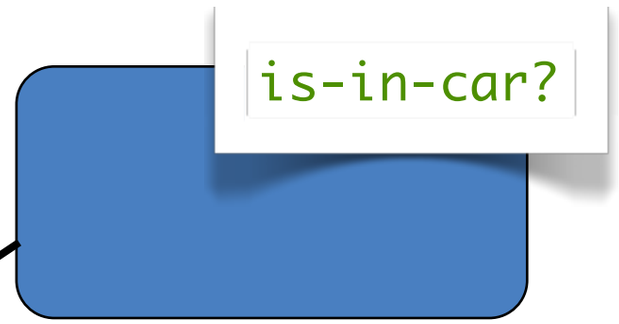
The ICoDE Model in Flute

Context sources
as reactive values





to-call **modal**



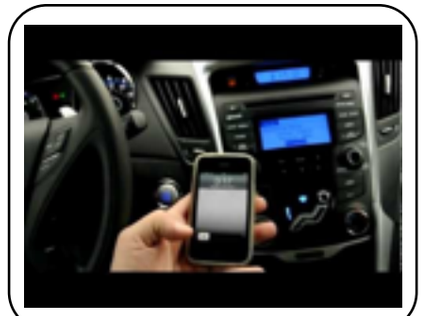
call-from-car **mode**

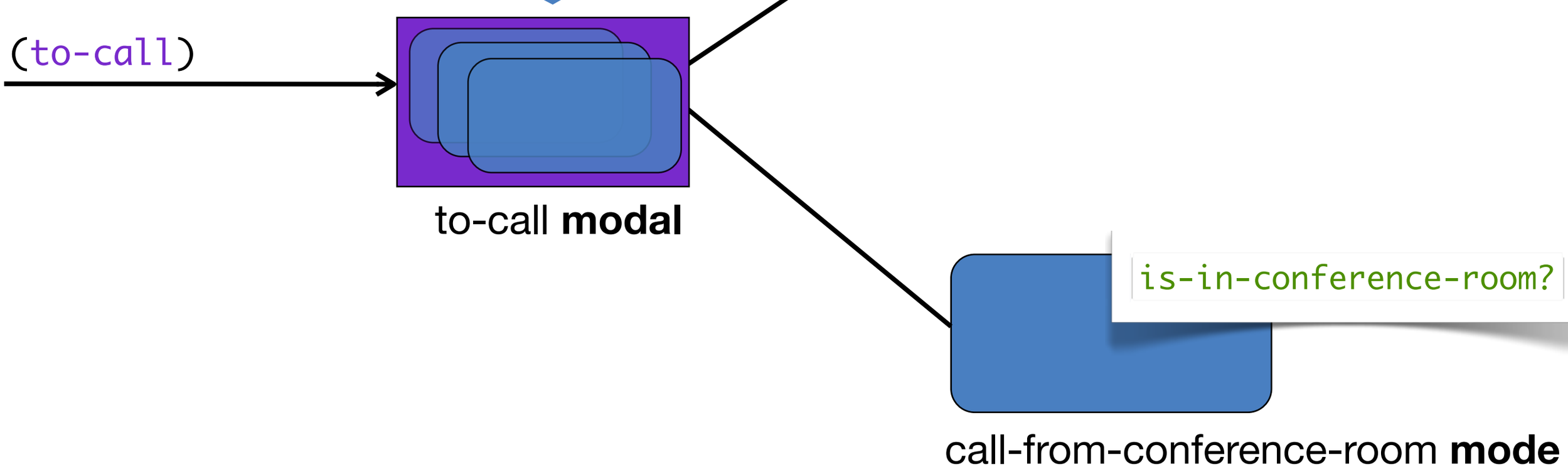
call-from-conference-room **mode**

Bainomugisha, E. *et.al*. (2012). Interruptible Context-dependent Executions: A Fresh Look at Programming Context-aware Applications. SPLASH/*OnWard! 2012*

Contextual and Reactive Dispatching

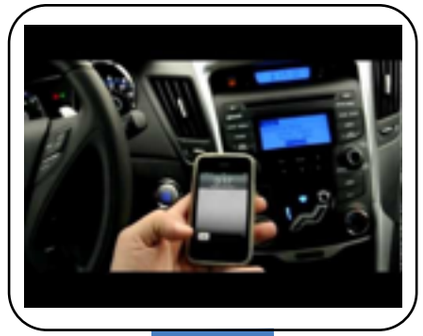
Context sources
as reactive values

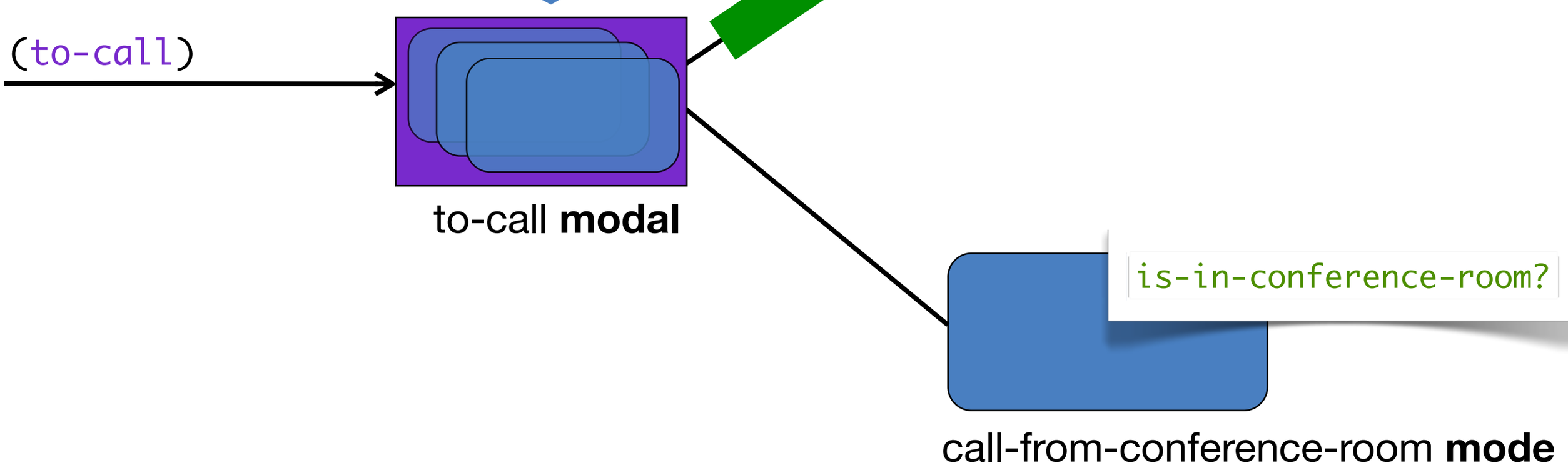




Contextual and Reactive Dispatching

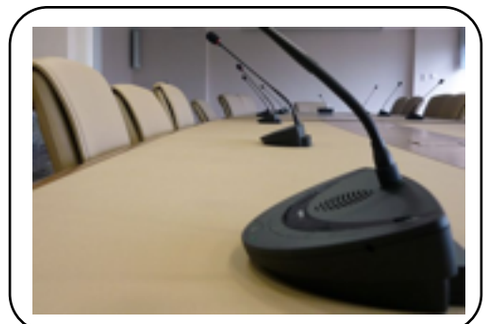
Context sources
as reactive values

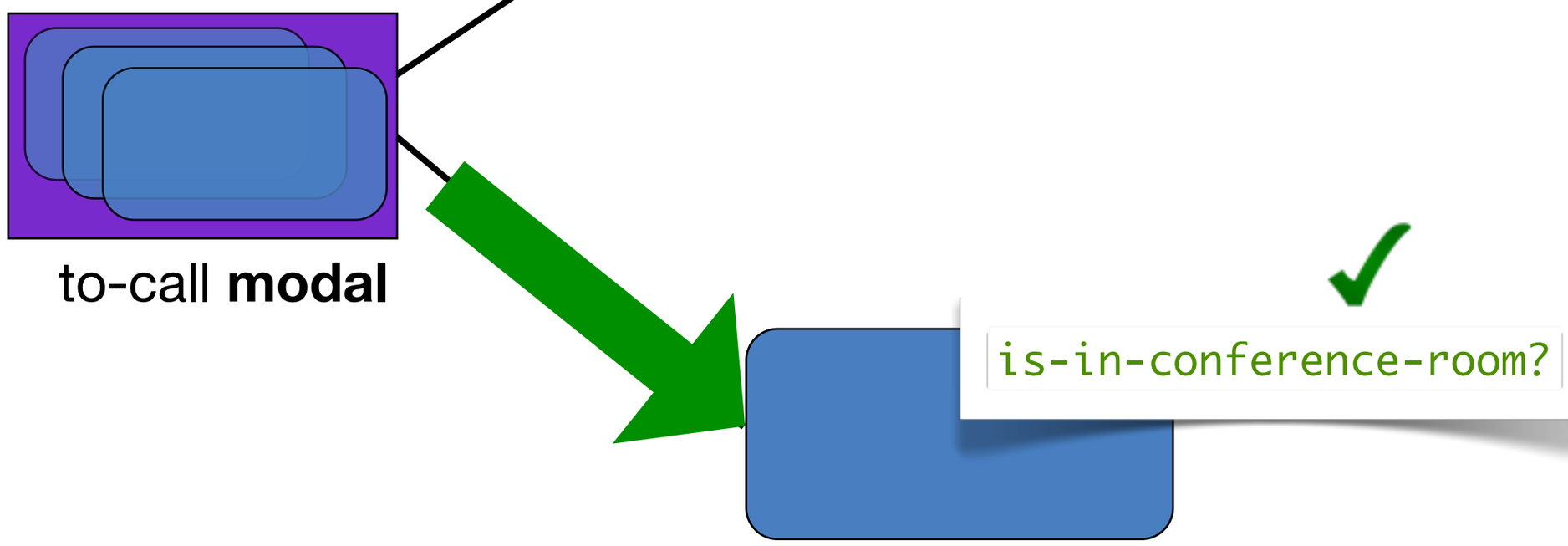




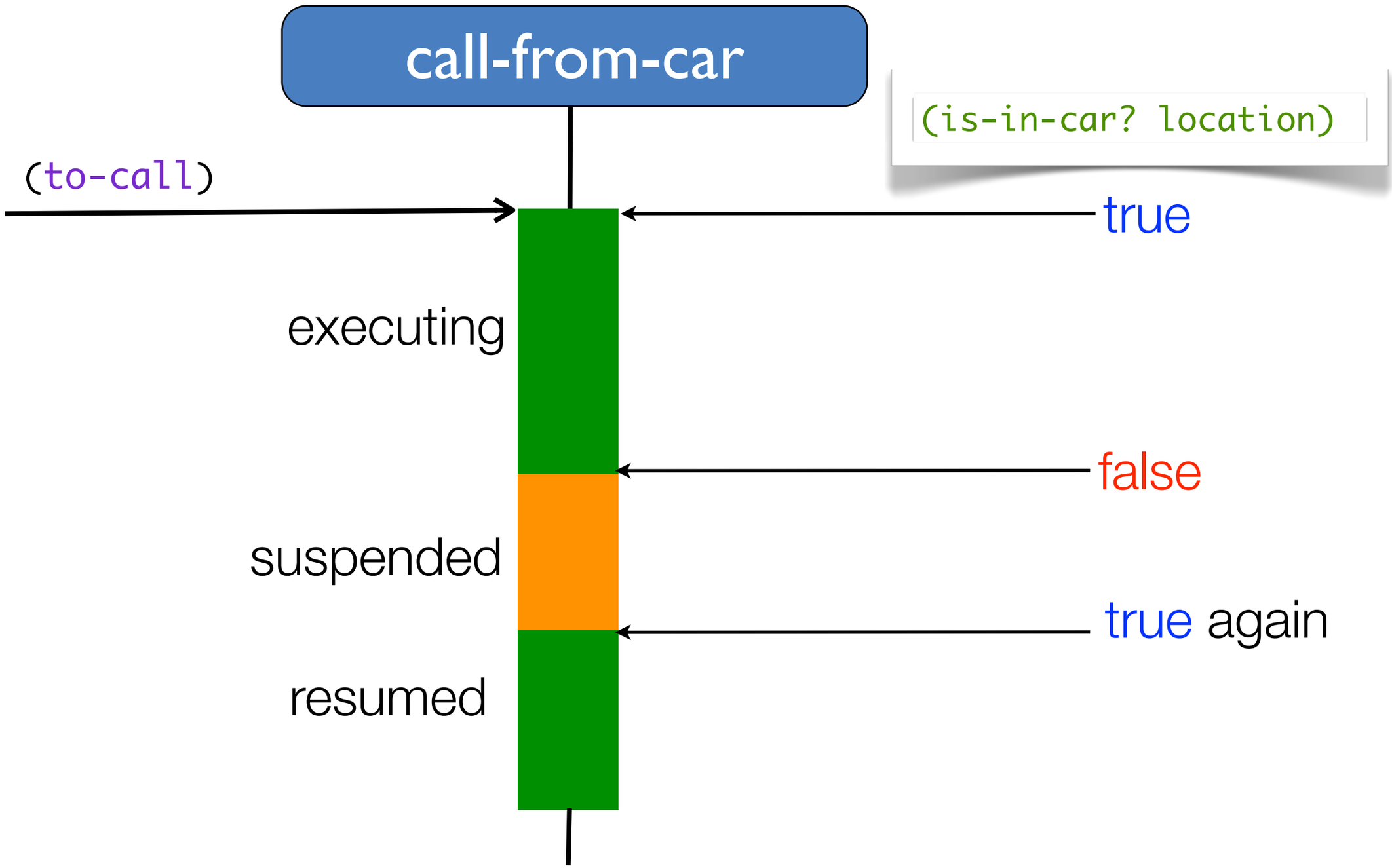
Contextual and **Reactive** Dispatching

Context sources
as reactive values





Interruptible and Resumable Executions



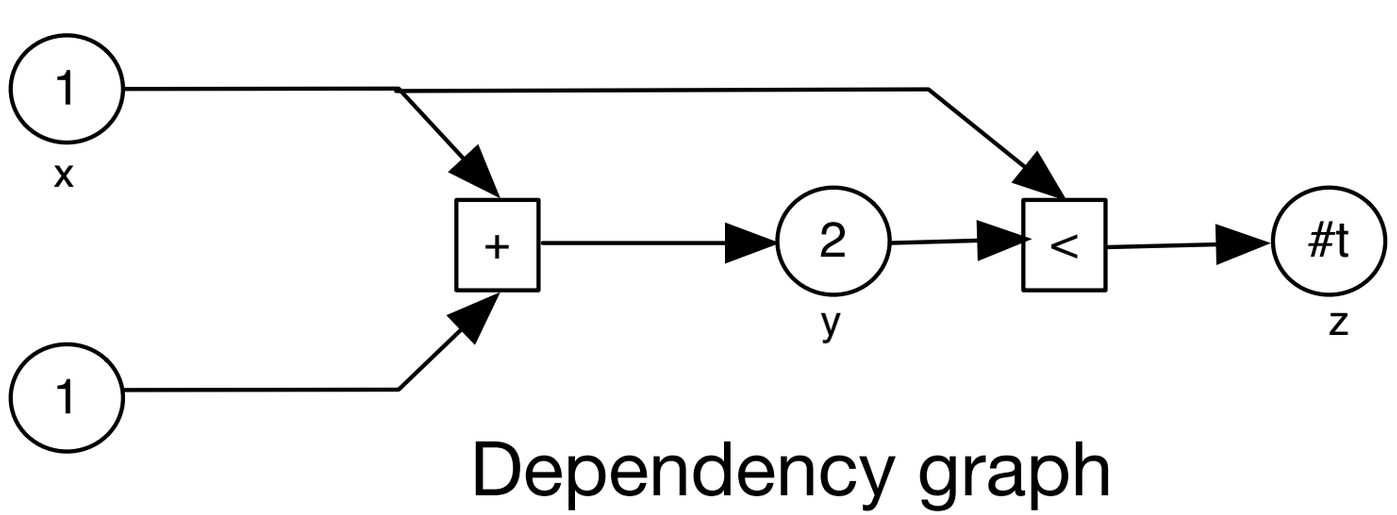
Reactive Programming in Flute

Reactive programming:

- Eliminates the use of explicit callbacks.
- Automatic propagation of changes among dependent values.

```
(define x (ctx-event 1))
(define y (+ x 1))
(define z (< x y))
```

Procedures are implicitly lifted.



Dependency graph

Bainomugisha, E. *et.al*. (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

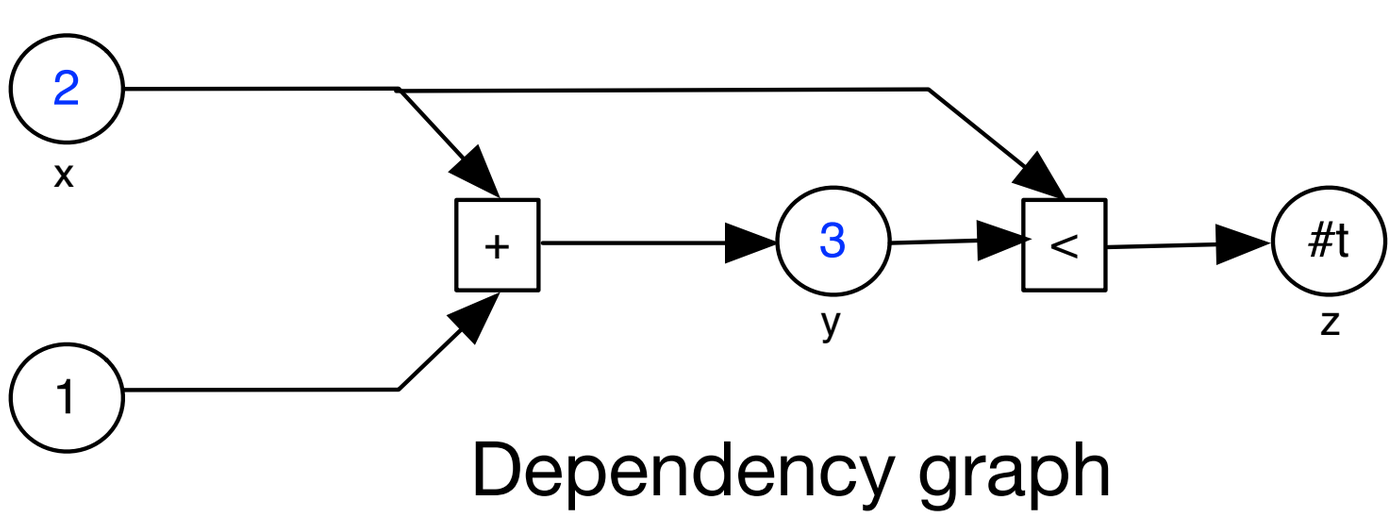
Reactive Programming in Flute

Reactive programming:

- Eliminates the use of explicit callbacks.
- Automatic propagation of changes among dependent values.

```
(define x (ctx-event 1))
(define y (+ x 1))
(define z (< x y))
```

Procedures are implicitly lifted.



Dependency graph

Bainomugisha, E. *et.al*. (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

Reactive Context Sources in Flute

Reactive
context sources

```
(define gps-coordinates (ctx-event))
(define location (gps->location gps-coordinates))



(CURRENT-LOCATION
 (lambda (latitude longitude)
   (update-value! gps-coordinates
                  (cons latitude longitude))))
```

Acquiring non-reactive values

from sensors.

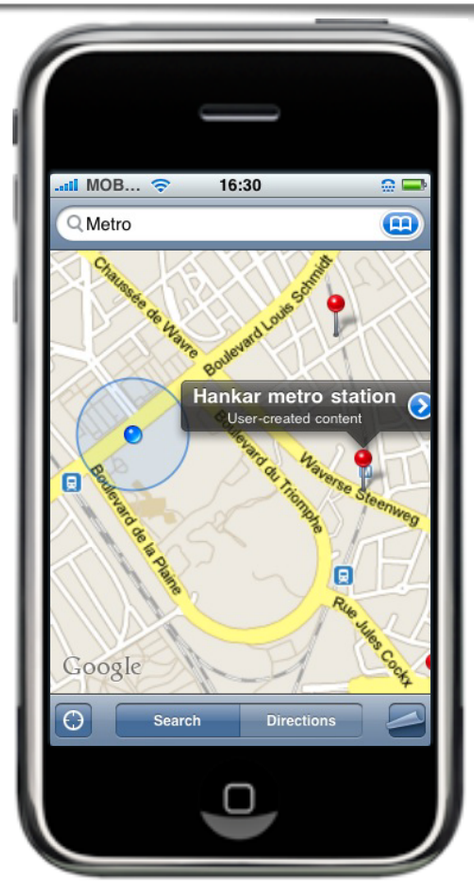
Bainomugisha, E. *et.al*. (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

Reactive Context Sources in Flute

Reactive
context sources

```
(define gps-coordinates (ctx-event))
(define location (gps->location gps-coordinates))
(define nearby-metro-station (get-metro-station location))
```

```
(show-map nearby-metro-station)
```



Bainomugisha, E. *et.al*. (2012). A Survey on Reactive Programming. *ACM Computing Surveys (to appear)*.

Procedure Modals and Modes in Flute

Procedure Modal:

```
(define to-call (modal (location)
  (define num-of-calls 0)))
```

Reactive context source

Shared variable

Procedure Mode:

```
(define call-from-car
  (mode (to-call)
    (is-in-car? location)
    (suspend resume deferred)
    (lambda ()
      (show "calling business contacts")
      (if (null? contacts-list)
          (show "business contacts list is empty")
          (let ((contact (car contacts-list)))
            (set! contacts-list (cdr contacts-list))
            (show (contact-name contact))
            (dial (phone-number contact))
            (turn-on-phonespeaker)
            (connect-to-car-speakers)
            (if (dial-next-contact? (user-response))
                (to-call))))))))
```

Modal

Context predicate

Interruption, resumption, and scoping strategies

Procedure Modals and Modes in Flute

Procedure Modal:

```
(define to-call (modal (location)
  (define num-of-calls 0)))
```

Reactive context source

Shared variable

Procedure Mode:

```
(define call-from-car
  (mode (to-call)
    (is-in-car? location)
    (suspend resume deferred)
    (lambda ()
      (show "calling business co
      (if (null? contacts-list)
          (show "business contac
          (let ((contact (car co
            (set! contacts-list
            (show (contact-name
            (dial (phone-number
            (turn-on-phonespeake
            (connect-to-car-spec
            (if (dial-next-conta
                (to-call)))))))
```

```
(define call-from-conference-room
  (mode (to-call)
    (is-in-conference-room? location)
    (suspend resume deferred)
    (lambda ()
      (show "calling co-workers contacts")
      (if (null? contacts-list)
          (show "co-workers contacts list is empty")
          (let ((contact (car contacts-list)))
            (set! contacts-list (cdr contacts-list))
            (show (contact-name contact))
            (dial (phone-number contact))
            (turn-on-phonespeaker)
            (connect-to-conference-equipment)
            (if (dial-next-contact? (user-response))
                (to-call)))))))
```

Variable Modals and Modes in Flute

Variable Modal:

```
(define contacts-list (modal (location)))
```

Variable Modes:

```
(mode (contacts-list)
  (is-in-car? location)
  (biz-contacts))
```

```
(mode (contacts-list)
  (is-in-conference-room? location)
  (co-workers-contacts))
```

Modal

Context predicate

Value

Variable Modals and Modes in Flute

```
(define call-from-car
  (mode (to-call)
    (is-in-car? location)
    (suspend resume deferred)
    (lambda ()
      (show "calling business contacts")
      (if (null? contacts-list)
          (show "business contacts list is empty")
          (let ((contact (car contacts-list)))
            (set! contacts-list (car contacts-list))
            (show (contact-name contact))
            (dial (phone-number contact))
            (turn-on-phonespeaker)
            (connect-to-car-speakers)
            (if (dial-next-contact? (user-response))
                (to-call)))))))
```

```
(define call-from-conference-room
  (mode (to-call)
    ference-room? location)
    esume deferred)

    alling co-workers contacts")
    l? contacts-list)
    w "co-workers contacts list is empty")
     ((contact (car contacts-list)))
    et! contacts-list (car contacts-list))
    how (contact-name contact))
    ial (phone-number contact))
    urn-on-phonespeaker)
    onnect-to-conference-equipment)
    o-call))))))
```

Different value depending on the current context.

Context-dependent Scoping for Modal Variables

```
(define call-from-car
  (mode (to-call)
    (is-in-car? location)
    (suspend resume deferred)
    (lambda ()
      (show "calling business contacts")
      (if (null? contacts-list)
          (show "business contacts list is empty")
          (let ((contact (car contacts-list)))
            (set! contacts-list (cdr contacts-list))
            (show (contact-name contact))
            (dial (phone-number contact))
            (turn-on-phonespeaker)
            (connect-to-car-speakers)
            (if (dial-next-contact? (user-response))
                (to-call))))))))
```

```
(define call-from-conference-room
  (mode (to-call)
    (is-in-conference-room? location)
    (suspend resume deferred)
    )
    calling co-workers contacts")
    ll? contacts-list)
    ow "co-workers contacts list is empty")
    t ((contact (car contacts-list)))
    set! contacts-list (cdr contacts-list))
    show (contact-name contact))
    dial (phone-number contact))
    turn-on-phonespeaker)
    connect-to-conference-equipment)
    f (dial-next-contact? (user-response))
        (to-call))))))))
```

Scoped assignments

Interruption and Resumption Strategies in Flute

Interruption Strategies

`abort` The execution is aborted when its context predicate is no longer satisfied.

`suspend` The execution is suspended and its execution state is saved.

Resumption Strategies

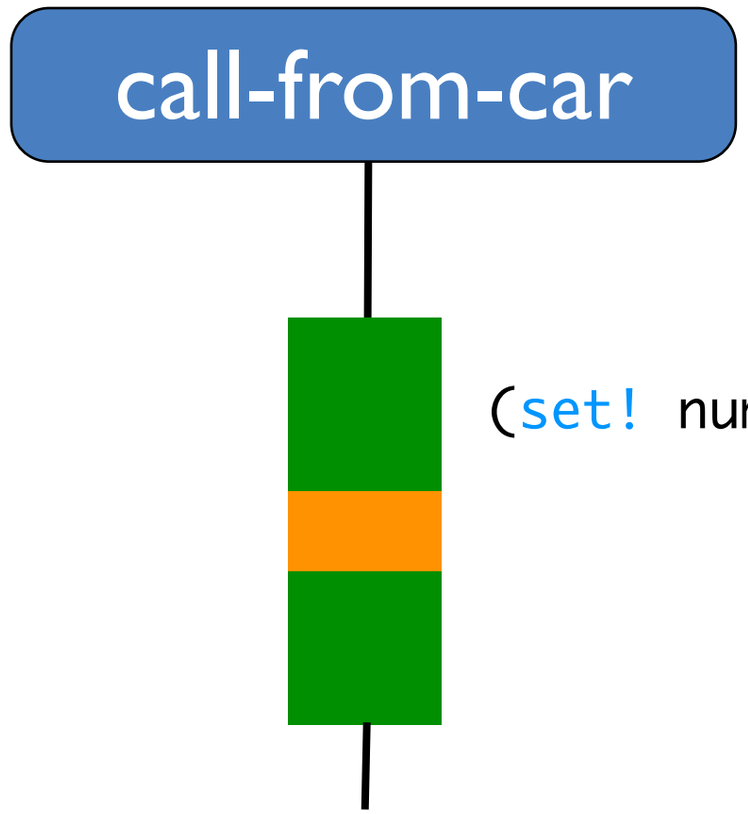
`restart` The execution is restarted from the beginning.

`resume` The execution is resumed from where it left off.

Interruption and Resumption Strategies in Flute

		Resumption Strategies	
Interruption Strategies		`restart`	`resume`
	`abort`	✔	✘
	`suspend`	✔	✔

Scoping of State Changes to Ordinary Variables in Flute



`immediate` Changes are immediately visible to other executions.

`deferred` Changes become visible to other executions on completion.

`isolated` Changes remain locally visible to the execution.

Semantics and Validation

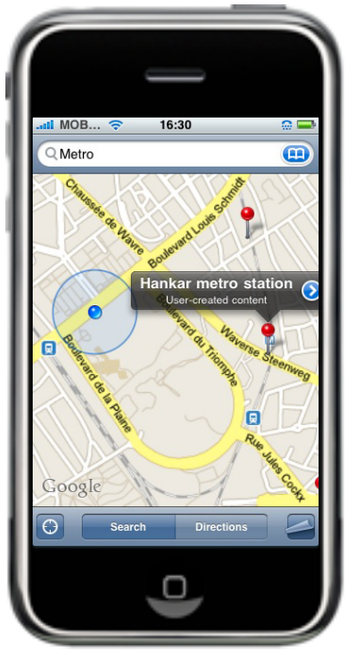
Semantics and Validation

iScheme

An Executable Semantics for Flute

Validation

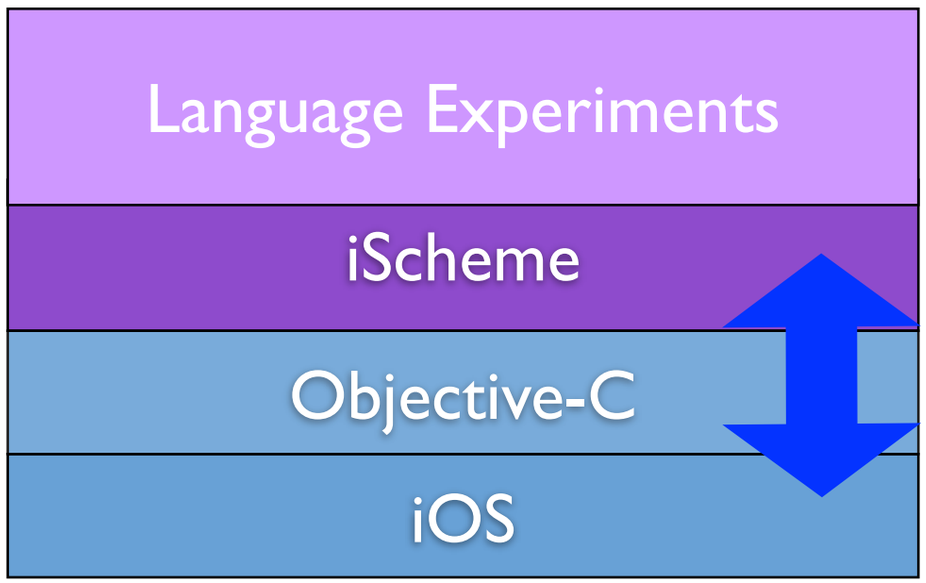
iScheme: A Mobile Language Laboratory



Access to iOS APIs (Sensors, GUI, ...)

Ambient-Oriented Programming

iScheme



Linguistic Symbiosis

Bainomugisha, E. *et.al*. (2012). Bringing Scheme Programming to the iPhone - Experience. *Software: Practice and Experience*, 42(3):331–356.

iScheme: A Mobile Language Laboratory

Language construct	Description
`OBJC-INSTANCE`	For instantiating an Objective-C class from within Scheme.
`OBJC-SEND`	Performing message sends to Objective-C instances.
`OBJC-CLASS`	Getting a reference to an Objective-C class.
`OBJC_TYPE`	A generic representation of Objective-C values in Scheme.
`SCHEME_CALL`	Calling Scheme procedures from Objective-C.
`service-type`	For creating a topic to publish or discover a procedure.
`export-service`	For exporting a procedure as a service.
`cancel-publication`	For cancelling a publication.
`when-discovered`	For discovering a procedure under a specified service type.
`cancel-subscription`	For cancelling a subscription..
`remote-send!`	For performing an asynchronous remote invocation without a return value.
`remote-send`	An asynchronous invocation with a return value.
`when-resolved`	For handling return values of remote invocations.
`due-in`	For specifying a maximum time to wait for a return value.
`catch`	For handling exceptions raised during an asynchronous invocation.

iScheme: A Mobile Language Laboratory

Language construct	Description
OBJC-INSTANCE	For instantiating an Obje Scheme.
OBJC-SEND	Performing message se s.
OBJC-CLASS	Getting a ref
OBJC_TYPE	A generic representati cheme.
SCHEME_CALL	Calling Scheme procedure
service-type	For creating a topic to publish or discover a procedure.
export-service	For exporting a procedure as a service.
cancel-publication	For cancelling a publication.
when-discovered	For discovering a procedure under a specified service type.
cancel-subscription	For cancelling a subscription..
remote-send!	For performing an asynchronous remote invocation without a return value.
remote-send	An asynchronous invocation with a return value.
when-resolved	For handling return values of remote invocations.
due-in	For specifying a maximum time to wait for a return value.
catch	For handling exceptions raised during an asynchronous invocation.

Linguistic symbiosis constructs

iScheme: A Mobile Language Laboratory

Language construct	Description
`OBJC-INSTANCE`	For instantiating an Objective-C class from within Scheme.
`OBJC-SEND`	Performing message sends to Objective-C instances.
`OBJC-CLASS`	Getting a reference to an Objective-C class.
`OBJC_TYPE`	A generic representation of Objective-C values in Scheme.
`SCHEME_CALL`	Calling Scheme procedures from Objective-C.
`service-type`	For creating a topic to publish or discover a procedure.
`export-service`	For exporting a procedure as a service.
`cancel-publication`	For cancelling a publicati
`when-discovered`	For discovering a proce e type.
`cancel-subscription`	For ca
`remote-send!`	For performing an asynch thout a return value.
`remote-send`	An asynchronous invocation with a return value.
`when-resolved`	For handling return values of remote invocations.
`due-in`	For specifying a maximum time to wait for a return value.
`catch`	For handling exceptions raised during an asynchronous invocation.

Distribution constructs

Semantics and Validation

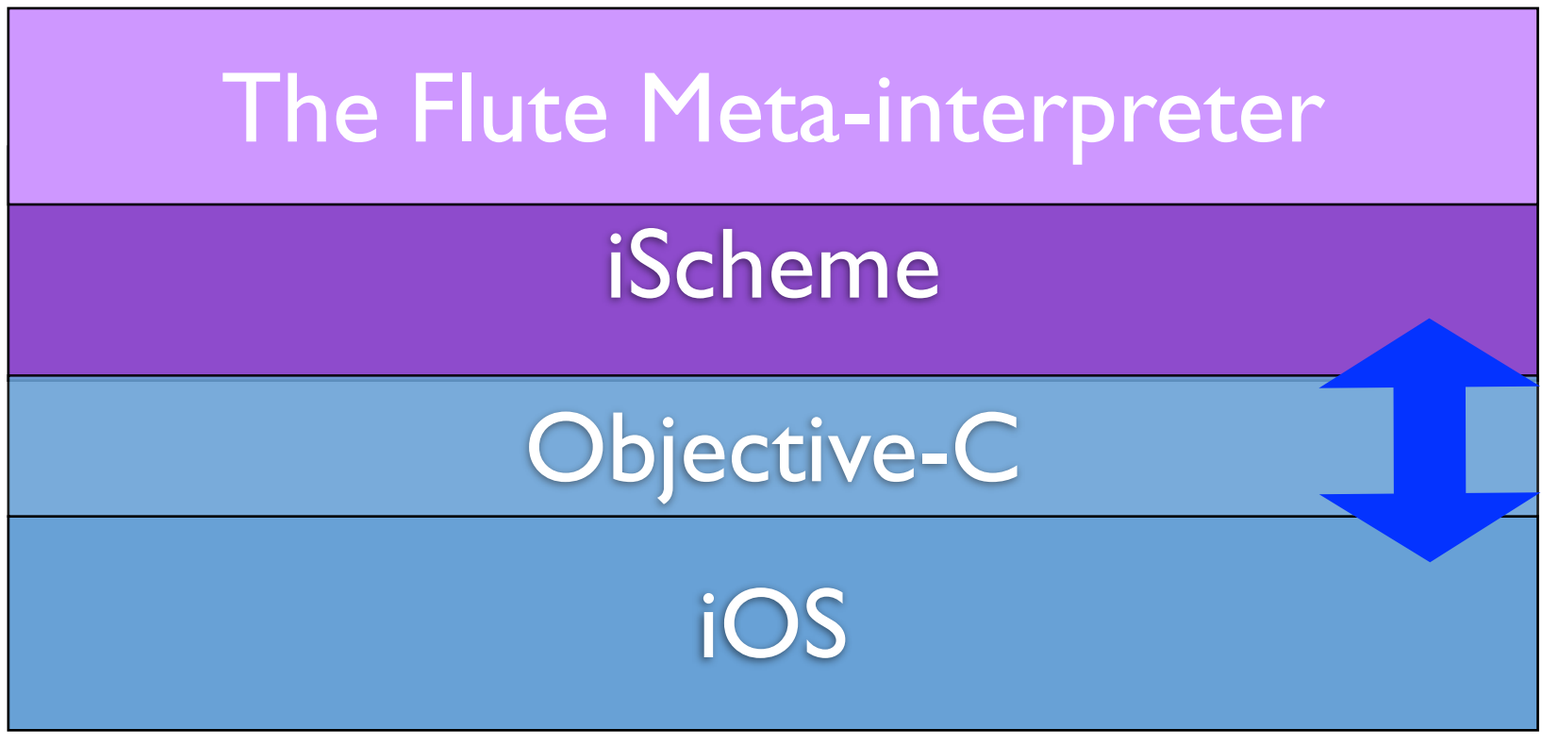
Semantics and Validation

An Executable Semantics for Flute

Validation

An Executable Semantics for Flute

- Flute is implemented as a meta-interpreter on top of iScheme.
- Context sources: GPS, proximity sensor, accelerometer on the iOS.



Bainomugisha, E. *et.al*. (2012). Bringing Scheme Programming to the iPhone - Experience. *Software: Practice and Experience*, 42(3):331–356.

An Executable Semantics for Flute

Mode eval

```
(define (eval-mode modal-expr pred-expr value-expr . exprs)
  (lambda (continue env tailcall)
    (define modal-binding (assoc (car modal-expr) env))

    (define (continue-after-value-expr variable-value env-after-value-expr)
      (define modal (cdr modal-binding))
      (let* ((params      '())
             (modal-envt (modal-env modal))
             (mode-envt  (append modal-envt env))
             (mode-proc  (make-proc params (list value-expr) mode-envt))
             (mode       (make-mode pred-expr mode-proc mode-envt)))
        (add-mode! modal mode)
        (continue mode-proc env)))

    (define (continue-after-config mode-config env-after-value-expr)
      (define (mode-params exprs)
        (car (cdr (car exprs))))
      (define (mode-exprs exprs)
        (cdr (cdr (car exprs))))
      (define modal (cdr modal-binding))
      (let* ((params       (mode-params exprs))
             (body-exprs   (mode-exprs exprs))
             (event-sources (modal-event-sources modal))
             (modal-envt   (modal-env modal))
             (mode-envt    (append modal-envt env))
             (mode-proc    (make-c-proc params
                                        (cons event-sources pred-expr)
                                        mode-config body-exprs mode-envt))
             (mode         (make-mode pred-expr mode-proc mode-envt)))
        (add-mode! modal mode)
        (continue mode-proc env)))

    (if (pair? exprs)
        (eval (car value-expr) continue-after-config  env #f)
        (eval value-expr continue-after-value-expr  env #f))))
```

Contextual and reactive dispatch eval

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
 (define head                (car body-exprs))
 (define tail                (cdr body-exprs))
 (define context-sources     (car pred-expr))
 (define context-pred        (cdr pred-expr))

 (define (continue-with-seq value env-after-seq)
   (eval-seq pred-expr config tail args continue env-after-seq tailcall))

 (define (continue-after-context-pred boolean env-after-pred)
   (if (equal? boolean #f)
       (begin
         (save-execution resume-evaluation context-sources)
         (continue interrupted env-after-pred))
       (if (null? tail)
           (eval head continue env-after-pred tailcall)
           (eval head continue-with-seq env-after-pred #f))))

 (eval context-pred continue-after-context-pred env #f))
```

Procedure's body eval

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
 (define head                (car body-exprs))
 (define tail                (cdr body-exprs))
 (define context-sources     (car pred-expr))
 (define context-pred        (cdr pred-expr))

 (define (continue-with-seq value env-after-seq)
   (eval-seq pred-expr config tail args continue env-after-seq tailcall))

 (define (continue-after-context-pred boolean env-after-pred)
   (if (equal? boolean #f)
       (begin
         (save-execution resume-evaluation context-sources)
         (continue interrupted env-after-pred))
       (if (null? tail)
           (eval head continue env-after-pred tailcall)
           (eval head continue-with-seq env-after-pred #f))))

 (eval context-pred continue-after-context-pred env #f))
```

Modal eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Main eval

```
(define (eval expr continue env tailcall)
  (cond
    ((symbol? expr)
     (eval-variable expr continue env))
    ((pair? expr)
     (let ((operator (car expr))
           (operands (cdr expr)))
       ((apply
         (case operator
           ((begin)            eval-begin       )
           ((define)           eval-define      )
           ((if)               eval-if          )
           ((lambda)           eval-lambda      )
           ((quote)            eval-quote       )
           ((set!)             eval-set!        )
           ((while)            eval-while       )
           ((let)              eval-let         )
           ((c-let)            eval-c-let       )
           ((let*)             eval-let*        )
           ((continuous)       eval-continuous  )
           ((c-set!)           eval-c-set!      )
           ((c-ref)            eval-c-ref       )
           ((modal)            eval-modal       )
           ((mode)             eval-mode        )
           ((OBJC-SEND)        eval-objc-send   )
           ((OBJC-CLASS)       eval-objc-class  )
           ((OBJC-INSTANCE)    eval-objc-instance)
           (else               (eval-application operator)))
         operands) continue env tailcall)))
    (else (continue expr env))))
```

Reactive values eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Modal invocation eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Variable modal access eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Scoped assignments eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

An Executable Semantics for Flute

Mode eval

```
(define (eval-mode modal-expr pred-expr value-expr . exprs)
  (lambda (continue env tailcall)
    (define modal-binding (assoc (car modal-expr) env))

    (define (continue-after-value-expr variable-value env-after-value-expr)
      (define modal (cdr modal-binding))
      (let* ((params      '())
             (modal-envt (modal-env modal))
             (mode-envt  (append modal-envt env))
             (mode-proc  (make-proc params (list value-expr) mode-envt))
             (mode       (make-mode pred-expr mode-proc mode-envt)))
        (add-mode! modal mode)
        (continue mode-proc env)))

    (define (continue-after-config mode-config env-after-value-expr)
      (define (mode-params exprs)
        (car (cdr (car exprs))))
      (define (mode-exprs exprs)
        (cdr (cdr (car exprs))))
      (define modal (cdr modal-binding))
      (let* ((params        (mode-params exprs))
             (body-exprs    (mode-exprs exprs))
             (event-sources (modal-event-sources modal))
             (modal-envt    (modal-env modal))
             (mode-envt     (append modal-envt env))
             (mode-proc     (make-c-proc params
                                         (cons event-sources pred-expr)
                                         mode-config body-exprs mode-envt))
             (mode          (make-mode pred-expr mode-proc mode-envt)))
        (add-mode! modal mode)
        (continue mode-proc env)))

    (if (pair? exprs)
        (eval (car value-expr) continue-after-config  env #f)
        (eval value-expr continue-after-value-expr  env #f))))
```

Contextual and reactive dispatch eval

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
  (define head                (car body-exprs))
  (define tail                (cdr body-exprs))
  (define context-sources     (car pred-expr))
  (define context-pred        (cdr pred-expr))

  (define (continue-with-seq value env-after-seq)
    (eval-seq pred-expr config tail args continue env-after-seq tailcall))

  (define (continue-after-context-pred boolean env-after-pred)
    (if (equal? boolean #f)
        (begin
          (save-execution resume-evaluation context-sources)
          (continue interrupted env-after-pred))
        (if (null? tail)
            (eval head continue env-after-pred tailcall)
            (eval head continue-with-seq env-after-pred #f))))

  (eval context-pred continue-after-context-pred env #f))
```

Procedure's body eval

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
  (define head                (car body-exprs))
  (define tail                (cdr body-exprs))
  (define context-sources     (car pred-expr))
  (define context-pred        (cdr pred-expr))

  (define (continue-with-seq value env-after-seq)
    (eval-seq pred-expr config tail args continue env-after-seq tailcall))

  (define (continue-after-context-pred boolean env-after-pred)
    (if (equal? boolean #f)
        (begin
          (save-execution resume-evaluation context-sources)
          (continue interrupted env-after-pred))
        (if (null? tail)
            (eval head continue env-after-pred tailcall)
            (eval head continue-with-seq env-after-pred #f))))

  (eval context-pred continue-after-context-pred env #f))
```

Modal eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params        (car exprs))
               (args          (map lookup params))
               (event-sources args)
               (modal-envt    (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Main eval

```
(define (eval expr continue env tailcall)
  (cond
    ((symbol? expr)
     (eval-variable expr continue env))
    ((pair? expr)
     (let ((operator (car expr))
           (operands (cdr expr)))
       ((apply
         (case operator
           ((begin)            eval-begin       )
           ((define)           eval-define      )
           ((if)               eval-if          )
           ((lambda)           eval-lambda      )
           ((quote)            eval-quote       )
           ((set!)             eval-set!        )
           ((while)            eval-while       )
           ((let)              eval-let         )
           ((c-let)            eval-c-let       )
           ((let*)             eval-let*        )
           ((continuous)       eval-continuous  )
           ((c-set!)           eval-c-set!      )
           ((c-ref)            eval-c-ref       )
           ((modal)            eval-modal       )
           ((mode)             eval-mode        )
           ((OBJC-SEND)        eval-objc-send   )
           ((OBJC-CLASS)       eval-objc-class  )
           ((OBJC-INSTANCE)    eval-objc-instance)
           (else               (eval-application operator)))
         operands) continue env tailcall)))
    (else (continue expr env))))
```

Reactive values eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params        (car exprs))
               (args          (map lookup params))
               (event-sources args)
               (modal-envt    (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Modal invocation eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params        (car exprs))
               (args          (map lookup params))
               (event-sources args)
               (modal-envt    (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Variable modal access eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params        (car exprs))
               (args          (map lookup params))
               (event-sources args)
               (modal-envt    (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Scoped assignments eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params        (car exprs))
               (args          (map lookup params))
               (event-sources args)
               (modal-envt    (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
              (continue mode-proc env))
              (if (< true-count true-count-end)
                  (continue #f env)
                  (error "Ambiguous predicates for procedure modal" true-count)))
      (let* ((head-pred       (car preds))
             (tail-preds      (cdr preds))
             (head-mode       (car modes))
             (tail-modes      (cdr modes))
             (mode-envt       (mode-env head-mode))
             (remaining-modes (cons head-mode modes-after-dispatch))
             (head-mode-proc  (mode-proc head-mode)))
        (define (continue-after-pred value env)
          (if value
              (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
              (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
        (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
              (continue mode-proc env))
              (if (< true-count true-count-end)
                  (continue #f env)
                  (error "Ambiguous predicates for procedure modal" true-count)))
      (let* ((head-pred       (car preds))
             (tail-preds      (cdr preds))
             (head-mode       (car modes))
             (tail-modes      (cdr modes))
             (mode-envt       (mode-env head-mode))
             (remaining-modes (cons head-mode modes-after-dispatch))
             (head-mode-proc  (mode-proc head-mode)))
        (define (continue-after-pred value env)
          (if value
              (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
              (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
        (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes       (modal-modes modal))
  (define preds       (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
              (continue mode-proc env))
              (if (< true-count true-count-end)
                  (continue #f env)
                  (error "Ambiguous predicates for procedure modal" true-count)))
        (let* ((head-pred       (car preds))
               (tail-preds      (cdr preds))
               (head-mode       (car modes))
               (tail-modes      (cdr modes))
               (mode-envt       (mode-env head-mode))
               (remaining-modes (cons head-mode modes-after-dispatch))
               (head-mode-proc  (mode-proc head-mode)))
          (define (continue-after-pred value env)
            (if value
                (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
                (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
          (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
              (continue mode-proc env))
              (if (< true-count true-count-end)
                  (continue #f env)
                  (error "Ambiguous predicates for procedure modal" true-count)))
        (let* ((head-pred       (car preds))
               (tail-preds      (cdr preds))
               (head-mode       (car modes))
               (tail-modes      (cdr modes))
               (mode-envt       (mode-env head-mode))
               (remaining-modes (cons head-mode modes-after-dispatch))
               (head-mode-proc  (mode-proc head-mode)))
          (define (continue-after-pred value env)
            (if value
                (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
                (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
          (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
              (continue mode-proc env))
              (if (< true-count true-count-end)
                  (continue #f env)
                  (error "Ambiguous predicates for procedure modal" true-count)))
        (let* ((head-pred       (car preds))
               (tail-preds      (cdr preds))
               (head-mode       (car modes))
               (tail-modes      (cdr modes))
               (mode-envt       (mode-env head-mode))
               (remaining-modes (cons head-mode modes-after-dispatch))
               (head-mode-proc  (mode-proc head-mode)))
          (define (continue-after-pred value env)
            (if value
                (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
                (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
          (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f  ()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
              (continue mode-proc env))
              (if (< true-count true-count-end)
                  (continue #f env)
                  (error "Ambiguous predicates for procedure modal" true-count)))
        (let* ((head-pred       (car preds))
               (tail-preds      (cdr preds))
               (head-mode       (car modes))
               (tail-modes      (cdr modes))
               (mode-envt       (mode-env head-mode))
               (remaining-modes (cons head-mode modes-after-dispatch))
               (head-mode-proc  (mode-proc head-mode)))
          (define (continue-after-pred value env)
            (if value
                (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
                (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
          (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
            (continue mode-proc env))
            (if (< true-count true-count-end)
                (continue #f env)
                (error "Ambiguous predicates for procedure modal" true-count)))
        (let* ((head-pred       (car preds))
               (tail-preds      (cdr preds))
               (head-mode       (car modes))
               (tail-modes      (cdr modes))
               (mode-envt       (mode-env head-mode))
               (remaining-modes (cons head-mode modes-after-dispatch))
               (head-mode-proc  (mode-proc head-mode)))
          (define (continue-after-pred value env)
            (if value
                (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
                (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
          (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (define (iterate preds modes true-count mode-proc modes-after-dispatch)
    (if (null? preds)
        (if (= true-count true-count-end)
            (continue mode-proc env))
            (if (< true-count true-count-end)
                (continue #f env)
                (error "Ambiguous predicates for procedure modal" true-count)))
        (let* ((head-pred       (car preds))
               (tail-preds      (cdr preds))
               (head-mode       (car modes))
               (tail-modes      (cdr modes))
               (mode-envt       (mode-env head-mode))
               (remaining-modes (cons head-mode modes-after-dispatch))
               (head-mode-proc  (mode-proc head-mode)))
          (define (continue-after-pred value env)
            (if value
                (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
                (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
          (eval head-pred continue-after-pred  mode-envt #f))))
  (iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (defin
    (if
```

```
(if (ormap event? context-sources)
    (let* ((dispatcher (ctx-event))
           (resumption-point
            (lambda ()
              (eval-modal-dispatch modal continue-after-dispatch env))))
      (set-thunk! dispatcher resumption-point)
      (for-each
       (lambda (context-source)
         (if (event? context-source)
             (register dispatcher context-source)))
       (listify context-sources))))
```

```
          (head-mode-proc  (mode-proc head-mode)))
      (define (continue-after-pred value env)
        (if value
            (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
            (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
      (eval head-pred continue-after-pred  mode-envt #f))))
(iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (if (ormap event? context-sources)
      (let* ((dispatcher (ctx-event))
             (resumption-point
              (lambda ()
                (eval-modal-dispatch modal continue-after-dispatch env))))
        (set-thunk! dispatcher resumption-point)
        (for-each
         (lambda (context-source)
           (if (event? context-source)
               (register dispatcher context-source)))
         (listify context-sources))))

      (head-mode-proc  (mode-proc head-mode)))
    (define (continue-after-pred value env)
      (if value
          (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
          (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
    (eval head-pred continue-after-pred  mode-envt #f))))
(iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (defin
    (if
```

```
(if (ormap event? context-sources)
    (let* ((dispatcher (ctx-event))
           (resumption-point
            (lambda ()
              (eval-modal-dispatch modal continue-after-dispatch env))))
      (set-thunk! dispatcher resumption-point)
      (for-each
       (lambda (context-source)
         (if (event? context-source)
             (register dispatcher context-source)))
       (listify context-sources))))
```

```
          (head-mode-proc  (mode-proc head-mode)))
      (define (continue-after-pred value env)
        (if value
            (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
            (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
      (eval head-pred continue-after-pred  mode-envt #f))))
(iterate preds modes true-count-start #f '()))
```

Contextual and Reactive Dispatch Evaluation

```
(define (eval-modal-dispatch modal continue env)
  (define modes      (modal-modes modal))
  (define preds      (modal-preds modal))
  (define true-count-end   1)
  (define true-count-start 0)

  (defin
    (if
```

```
(if (ormap event? context-sources)
    (let* ((dispatcher (ctx-event))
           (resumption-point
            (lambda ()
              (eval-modal-dispatch modal continue-after-dispatch env))))
      (set-thunk! dispatcher resumption-point)
      (for-each
       (lambda (context-source)
         (if (event? context-source)
             (register dispatcher context-source)))
       (listify context-sources))))
```

```
          (head-mode-proc  (mode-proc head-mode)))
      (define (continue-after-pred value env)
        (if value
            (iterate tail-preds  tail-modes (+ true-count 1) head-mode-proc modes-after-dispatch)
            (iterate tail-preds  tail-modes true-count mode-proc remaining-modes)))
      (eval head-pred continue-after-pred  mode-envt #f))))
(iterate preds modes true-count-start #f '()))
```

An Executable Semantics for Flute

Mode eval

```
(define (eval-mode modal-expr pred-expr value-expr . exprs)
  (lambda (continue env tailcall)
    (define modal-binding (assoc (car modal-expr) env))

    (define (continue-after-value-expr variable-value env-after-value-expr)
      (define modal (cdr modal-binding))
      (let* ((params      '())
             (modal-envt (modal-env modal))
             (mode-envt  (append modal-envt env))
             (mode-proc  (make-proc params (list value-expr) mode-envt))
             (mode       (make-mode pred-expr mode-proc mode-envt)))
        (add-mode! modal mode)
        (continue mode-proc env)))

    (define (continue-after-config mode-config env-after-value-expr)
      (define (mode-params exprs)
        (car (cdr (car exprs))))
      (define (mode-exprs exprs)
        (cdr (cdr (car exprs))))
      (define modal (cdr modal-binding))
      (let* ((params       (mode-params exprs))
             (body-exprs   (mode-exprs exprs))
             (event-sources (modal-event-sources modal))
             (modal-envt   (modal-env modal))
             (mode-envt    (append modal-envt env))
             (mode-proc    (make-c-proc params
                                        (cons event-sources pred-expr)
                                        mode-config body-exprs mode-envt))
             (mode         (make-mode pred-expr mode-proc mode-envt)))
        (add-mode! modal mode)
        (continue mode-proc env)))

    (if (pair? exprs)
        (eval (car value-expr) continue-after-config  env #f)
        (eval value-expr continue-after-value-expr  env #f))))
```

Contextual and reactive dispatch eval

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
  (define head                (car body-exprs))
  (define tail                (cdr body-exprs))
  (define context-sources     (car pred-expr))
  (define context-pred        (cdr pred-expr))

  (define (continue-with-seq value env-after-seq)
    (eval-seq pred-expr config tail args continue env-after-seq tailcall))

  (define (continue-after-context-pred boolean env-after-pred)
    (if (equal? boolean #f)
        (begin
          (save-execution resume-evaluation context-sources)
          (continue interrupted env-after-pred))
        (if (null? tail)
            (eval head continue env-after-pred tailcall)
            (eval head continue-with-seq env-after-pred #f))))

  (eval context-pred continue-after-context-pred env #f))
```

Procedure's body eval

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
  (define head                (car body-exprs))
  (define tail                (cdr body-exprs))
  (define context-sources     (car pred-expr))
  (define context-pred        (cdr pred-expr))

  (define (continue-with-seq value env-after-seq)
    (eval-seq pred-expr config tail args continue env-after-seq tailcall))

  (define (continue-after-context-pred boolean env-after-pred)
    (if (equal? boolean #f)
        (begin
          (save-execution resume-evaluation context-sources)
          (continue interrupted env-after-pred))
        (if (null? tail)
            (eval head continue env-after-pred tailcall)
            (eval head continue-with-seq env-after-pred #f))))

  (eval context-pred continue-after-context-pred env #f))
```

Modal eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Main eval

```
(define (eval expr continue env tailcall)
  (cond
    ((symbol? expr)
     (eval-variable expr continue env))
    ((pair? expr)
     (let ((operator (car expr))
           (operands (cdr expr)))
       ((apply
         (case operator
           ((begin)            eval-begin       )
           ((define)           eval-define      )
           ((if)               eval-if          )
           ((lambda)           eval-lambda      )
           ((quote)            eval-quote       )
           ((set!)             eval-set!        )
           ((while)            eval-while       )
           ((let)              eval-let         )
           ((c-let)            eval-c-let       )
           ((let*)             eval-let*        )
           ((continuous)       eval-continuous  )
           ((c-set!)           eval-c-set!      )
           ((c-ref)            eval-c-ref       )
           ((modal)            eval-modal       )
           ((mode)             eval-mode        )
           ((OBJC-SEND)        eval-objc-send   )
           ((OBJC-CLASS)       eval-objc-class  )
           ((OBJC-INSTANCE)    eval-objc-instance)
           (else               (eval-application operator)))
         operands) continue env tailcall)))
    (else (continue expr env))))
```

Reactive values eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Modal invocation eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Variable modal access eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Scoped assignments eval

```
(define (eval-modal . exprs)
  (lambda (continue env tailcall)
    (define empty-envt '())
    (define (lookup variable)
      (define binding (assoc variable env))
      (if (pair? binding)
          (cdr binding)
          (meta-level-eval variable (interaction-env))))

    (if (pair? exprs)
        (let* ((params       (car exprs))
               (args         (map lookup params))
               (event-sources args)
               (modal-envt   (bind-params params args empty-envt)))
          (define (continue-after-modal-exprs value env-after-modal-exprs)
            (continue (make-modal event-sources env-after-modal-exprs) env))
          (if (pair? (cdr exprs))
              (eval-seq (cdr exprs)
                        continue-after-modal-exprs
                        modal-envt tailcall)
              (continue-after-modal-exprs '()  modal-envt)))
        (continue (make-modal '() empty-envt) env))))
```

Interruptible Evaluation of a Procedure's Body

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
 (define context-sources      (car pred-expr))
 (define context-pred         (cdr pred-expr))
 (define head                 (car body-exprs))
 (define tail                 (cdr body-exprs))

 (define (continue-with-seq value env-after-seq)
   (eval-seq pred-expr config tail args continue env-after-seq tailcall))

(define (continue-after-context-pred boolean env-after-pred)
  (if (equal? boolean #f)
      (begin
        (save-execution resume-evaluation context-sources)
        (continue interrupted env-after-pred))
      (if (null? tail)
          (eval head continue env-after-pred tailcall)
          (eval head continue-with-seq env-after-pred #f))))

(eval context-pred continue-after-context-pred env #f))
```

Interruptible Evaluation of a Procedure's Body

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
 (define context-sources      (car pred-expr))
 (define context-pred         (cdr pred-expr))
 (define head                 (car body-exprs))
 (define tail                 (cdr body-exprs))

 (define (continue-with-seq value env-after-seq)
   (eval-seq pred-expr config tail args continue env-after-seq tailcall))

 (define (continue-after-context-pred boolean env-after-pred)
   (if (equal? boolean #f)
       (begin
         (save-execution resume-evaluation context-sources)
         (continue interrupted env-after-pred))
       (if (null? tail)
           (eval head continue env-after-pred tailcall)
           (eval head continue-with-seq env-after-pred #f))))

(eval context-pred continue-after-context-pred env #f))
```

Interruptible Evaluation of a Procedure's Body

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
 (define context-sources      (car pred-expr))
 (define context-pred         (cdr pred-expr))
 (define head                 (car body-exprs))
 (define tail                 (cdr body-exprs))

 (define (continue-with-seq value env-after-seq)
   (eval-seq pred-expr config tail args continue env-after-seq tailcall))

(define (continue-after-context-pred boolean env-after-pred)
  (if (equal? boolean #f)
      (begin
        (save-execution resume-evaluation context-sources)
        (continue interrupted env-after-pred))
      (if (null? tail)
          (eval head continue env-after-pred tailcall)
          (eval head continue-with-seq env-after-pred #f))))

(eval context-pred continue-after-context-pred env #f))
```

Interruptible Evaluation of a Procedure's Body

```
(define (eval-seq pred-expr config body-exprs args continue env tailcall)
 (define context-sources      (car pred-expr))
 (define context-pred         (cdr pred-expr))
 (define head                 (car body-exprs))
 (define tail                 (cdr body-exprs))

 (define (continue-with-seq value env-after-seq)
   (eval-seq pred-expr config tail args continue env-after-seq tailcall))

(define (continue-after-context-pred boolean env-after-pred)
  (if (equal? boolean #f)
      (begin
        (save-execution resume-evaluation context-sources)
        (continue interrupted env-after-pred))
      (if (null? tail)
          (eval head continue env-after-pred tailcall)
          (eval head continue-with-seq env-after-pred #f))))

(eval context-pred continue-after-context-pred env #f))
```

Semantics and Validation

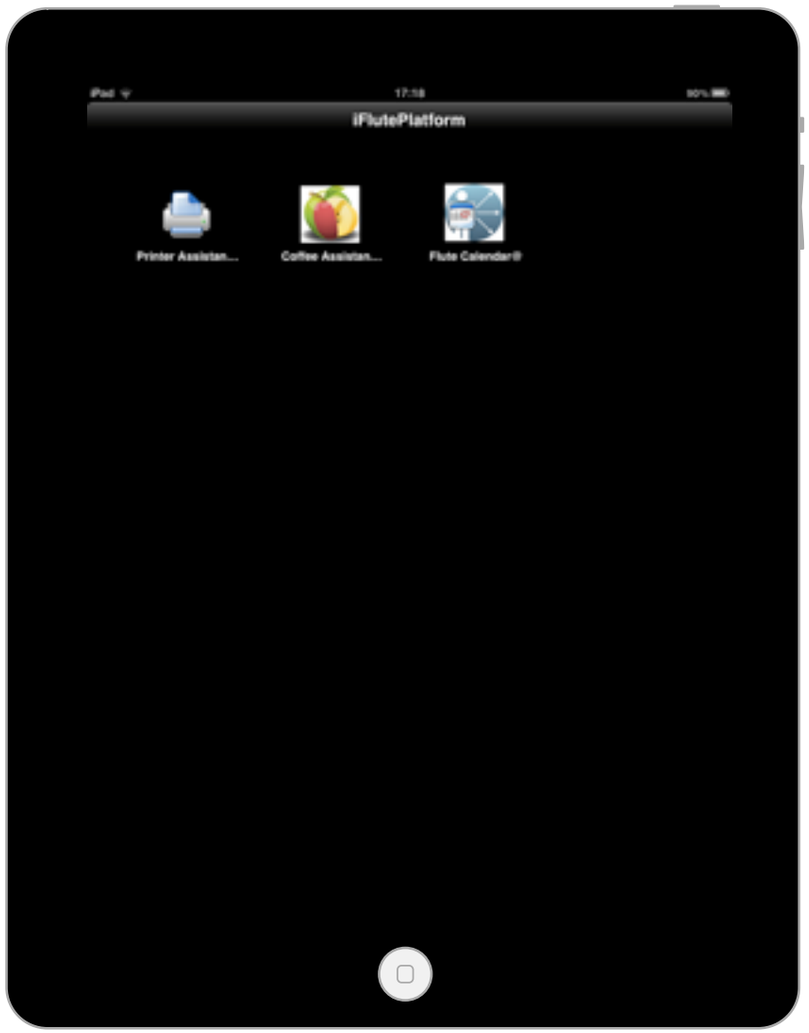
Semantics and Validation

iScheme

An Executable Semantics for Flute

Validation

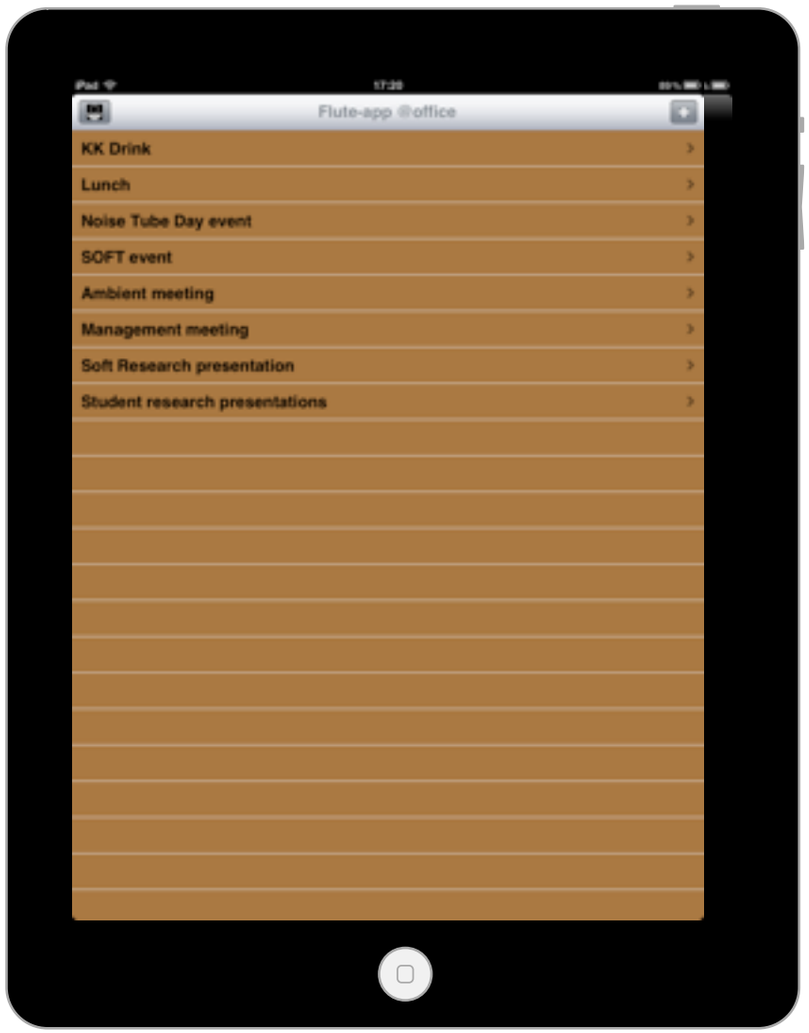
Validation: The iFlute Mobile Platform



Example apps on the iFlute mobile platform

1. *Kalenda:* a context-aware calendar assistant.
2. *Pulinta:* a context-aware printer assistant.
3. *Tasiki*: a context-aware task assistant.

Validation: The iFlute Mobile Platform



Example apps on the iFlute mobile platform

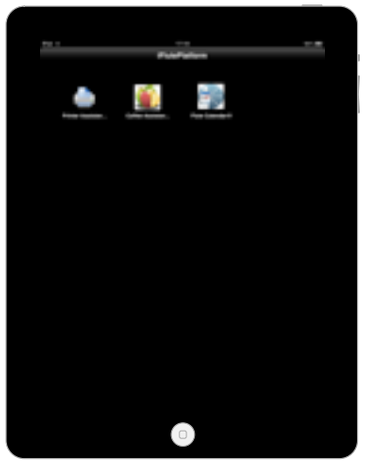
1. *Kalenda:* a context-aware calendar assistant.
2. *Pulinta:* a context-aware printer assistant.
3. *Tasiki*: a context-aware task assistant.

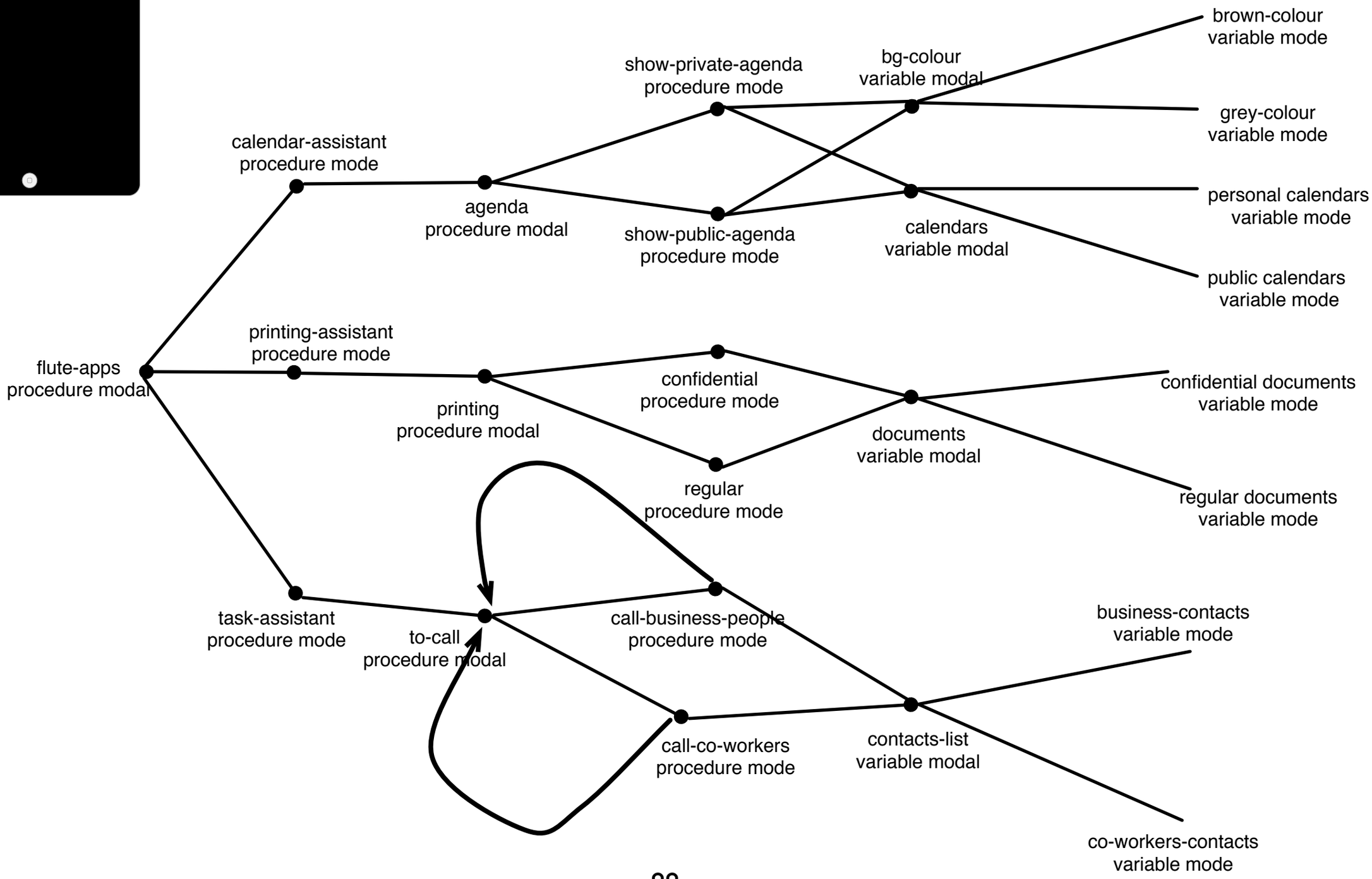
Validation: The iFlute Mobile Platform

Example apps on the iFlute mobile platform

1. *Kalenda:* a context-aware calendar assistant.
2. *Pulinta:* a context-aware printer assistant.
3. *Tasiki*: a context-aware task assistant.

Validation: The iFlute Mobile Platform





Evaluation

Language Construct	Satisfied Requirement
`ctx-event`	✓ Chained context reactions
`mode`	✓ Contextual dispatch
`modal`	✓ Reactive dispatch

`suspend`
`abort`
`resume`
`restart`
	✓ Context-dependent interruptions
	✓ Context-dependent resumptions

`deferred`
`immediate`
`isolated`
	✓ Reactive scope management

Evaluation

Language Construct	Satisfied Requirement
`ctx-event`	✓ Chained context reactions
`mode`	✓ Contextual dispatch
`modal`	✓ Reactive dispatch

`suspend`
`abort`
`resume`
`restart`
	✓ Context-dependent interruptions
	✓ Context-dependent resumptions

`deferred`
`immediate`
`isolated`
	✓ Reactive scope management

Evaluation

Language Construct	Satisfied Requirement
`ctx-event`	✓ Chained context reactions
`mode`	✓ Contextual dispatch
`modal`	✓ Reactive dispatch

`suspend`
`abort`
`resume`
`restart`
	✓ Context-dependent interruptions
	✓ Context-dependent resumptions

`deferred`
`immediate`
`isolated`
	✓ Reactive scope management

Evaluation

Language Construct	Satisfied Requirement
`ctx-event`	✓ Chained context reactions
`mode`	✓ Contextual dispatch
`modal`	✓ Reactive dispatch

`suspend`
`abort`
`resume`
`restart`
	✓ Context-dependent interruptions
	✓ Context-dependent resumptions

`deferred`
`immediate`
`isolated`
	✓ Reactive scope management

Evaluation

Language Construct	Satisfied Requirement
`ctx-event`	✓ Chained context reactions
`mode`	✓ Contextual dispatch
`modal`	✓ Reactive dispatch

`suspend`
`abort`
`resume`
`restart`
	✓ Context-dependent interruptions
	✓ Context-dependent resumptions

`deferred`
`immediate`
`isolated`
	✓ Reactive scope management

Limitations and Future Research Directions

- **Garbage collection of suspended executions.**
- **Flute programming language for the real world.**
- Ambiguous context predicates.
- Evaluation overhead.
- Data structure mutations for variable modals.
- Distributed interruptions.

Summarising the Contributions

1. The ICoDE model:
   - **Interruptible** and **resumable** executions.
   - Representation of context as **reactive** values.
   - **Contextual** and **reactive** dispatch.
   - **Reactive scope** management.

2. iScheme: a mobile language experimentation laboratory.

3. The Flute programming language and its executable semantics.

4. The iFlute Platform: a proof-of-concept mobile application platform for reactive context-aware applications.

Summarising the Contributions

1. The ICoDE model:
   - **Interruptible** and **resumable** executions.
   - Representation of context as **reactive**
   - **Contextual** and **reactive** dis
   - **Reactive scope** mana

2. iScheme: perimentation laboratory.

3. g language and its executable
   ute Platform: a proof-of-concept mobile application platform for reactive context-aware applications.

Reactive Method Dispatch for Context-Oriented Programming

Supporting Publications

1. **Journal**: E. Bainomugisha, A. Lombide Carreton, T. Van Cutsem, S. Mostinckx, W. De Meuter. A Survey on Reactive Programming. To appear in *ACM Computing Surveys, 2012.*

2. **Conference**: E. Bainomugisha, J. Vallejos, C. De Roover, A. Lombide Carreton, W. De Meuter. Interruptible Context-dependent Executions: A Fresh Look at Programming Context-aware Applications. In *Proceedings of the ACM SPLASH/Onward! '12*, 2012.

3. **Journal**: E. Bainomugisha, J. Vallejos, E. Gonzalez Boix, P. Costanza, T. D'Hondt, W. De Meuter. Bringing Scheme Programming to the iPhone - Experience. In *Software: Practice and Experience*, 2012.

4. **Conference**: E. Bainomugisha, P. Koosha, J. Vallejos, Y. Berbers, W. De Meuter. Flexub: Dynamic Subscriptions for Publish/Subscribe Systems in MANETs. In *12th IFIP International Conference on Distributed Applications and Interoperable Systems*, Lecture Notes in Computer Science, 2012.

5. **Book chapter:** E. Bainomugisha, A. Cádiz, P. Costanza, W. De Meuter, S. Gonzalez, K. Mens, J. Vallejos, T. Van Cutsem. Language Engineering for Mobile Software. In *Handbook of Research on Mobile Software Engineering: Design Implementation and Emergent Applications*, IGI Global, 2011.