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

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

@Printer room

@Conference room

@Car
Reactive Context-aware Applications

@Office

@Printer room

@Conference room

@Car
Reactive Context-aware Applications

@Office

@Printer room

@Conference room

@Car
Reactive Context-aware Applications

@Office

@Printer room

@Conference room

@Car
Reactive Context-aware Applications

@Office

@Printer room

@Conference room

@Car

Sorry. You need to correct the fields marked below before continuing.
Reactive Context-aware Applications

@Office

@Printer room

@Conference room

to-call

to-call

@Car

co-workers

business contacts

Sorry. You need to correct the fields marked below before continuing.

Reactive Context-aware Applications

@Office

@Printer room

@Conference room

to-call

to-call

@Car

co-workers

business contacts

Sorry. You need to correct the fields marked below before continuing.
Reactive Context-aware Applications

@Office

@Printer room

@Conference room

@Car

Fundamental Characteristics:

1. **Context-constrained** executions.
2. **Sudden** interruptions.
3. **Prompt** adaptability.
(define call-from-car
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list empty")
        (let ((contact (car contacts-list)))
          (show (contact-name contact))
          (dial (phone-number contact))
          (turn-on-phone-speaker)
          (connect-to-car-speakers)
          (if (dial-next-contact? (user-response))
               (call-from-car (cdr contacts-list)))))))
Programming Reactive Context-aware Applications

(define call-from-car
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list empty")
        (let ((contact (car contacts-list)))
          (show (contact-name contact))
          (dial (phone-number contact))
          (turn-on-phone-speaker)
          (connect-to-car-speakers)
          (if (dial-next-contact? (user-response))
              (call-from-car (cdr contacts-list)))))))
define call-from-car
(lambda (contacts-list)
  (show "calling business contacts")
  (if (null? contacts-list)
      (show "business contacts list empty")
      (let ((contact (car contacts-list)))
        (show (contact-name contact))
        (dial (phone-number contact))
        (turn-on-phone-speaker)
        (connect-to-car-speakers)
        (if (dial-next-contact? (user-response))
            (call-from-car (cdr contacts-list)))))))
(define call-from-car
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list empty")
        (let ((contact (car contacts-list)))
            (show (contact-name contact))
            (dial (phone-number contact))
            (turn-on-phone-speaker)
            (connect-to-car-speakers)
            (if (dial-next-contact? (user-response))
                (call-from-car (cdr contacts-list))))))))
Programming Reactive Context-aware Applications

```scheme
(define call-from-car
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list empty")
        (let ((contact (car contacts-list)))
          (show (contact-name contact))
          (dial (phone-number contact))
          (turn-on-phone-speaker)
          (connect-to-car-speakers)
          (if (dial-next-contact? (user-response))
              (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)))))))
(define call-from-car
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list empty")
        (let (((contact (car contacts-list)))
            (show (contact-name contact))
            (dial (phone-number contact))
            (turn-on-phone-speaker)
            (connect-to-car-speakers)
            (if (dial-next-contact? (user-response))
                (call-from-car (cdr contacts-list))))))))
Programming Reactive Context-aware Applications

```
(define call-from-car
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list empty")
        (let ((contact (car contacts-list)))
          (show (contact-name contact))
          (dial (phone-number contact))
          (turn-on-phone-speaker)
          (connect-to-car-speakers)
          (if (dial-next-contact? (user-response))
              (call-from-car (cdr contacts-list)))))))
```
Programming Reactive Context-aware Applications

```scheme
(define call-from-car
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list empty")
        (let ((contact (car contacts-list)))
          (show (contact-name contact))
          (dial (phone-number contact))
          (turn-on-phone-speaker)
          (connect-to-car-speakers)
          (if (dial-next-contact? (user-response))
              (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

\[
\textbf{(if} \ (\text{is-in-car?} \ \text{location}) \\
\text{\quad (call-from-car business-contacts)})
\]
A context change can occur at any moment during a procedure execution.

Incorrect!

Not enough!
Programming Reactive Context-aware Applications

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?

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

```scheme
(define to-call
  (lambda (contacts-list)
    (show "calling business contacts")
    (if (null? contacts-list)
        (show "business contacts list is empty")
        (let ((contact (car contacts-list)))
          (show (contact-name contact))
          (dial (phone-number contact))
          (turn-on-phone-speaker)
          (connect-to-car-speakers)
          (if (dial-next-contact? (user-response))
              (to-call (cdr contacts-list))))))))

(define to-call
  (lambda (contacts-list)
    (show "calling co-workers contacts")
    (if (null? contacts-list)
        (show "co-workers contacts list is empty")
        (let ((contact (car contacts-list)))
          (show (contact-name contact))
          (dial (phone-number contact))
          (turn-on-phone-speaker)
          (connect-to-conference-equipment)
          (if (dial-next-contact? (user-response))
              (to-call (cdr contacts-list))))))))
```

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

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

<table>
<thead>
<tr>
<th></th>
<th>Chained Context Reactions</th>
<th>Context-dependent Interruptions</th>
<th>Context-dependent Resumptions</th>
<th>Contextual Dispatch</th>
<th>Reactive Dispatch</th>
<th>Reactive Scope Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-Oriented Programming (COP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContextL and other Layer-based Languages</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Lambic</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Ambience</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Contextual Values</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Functional Reactive Programming (FRP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrTime and its Siblings</td>
<td>✅ behaviours and events</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Reactive SML</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced Control Flow Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>First-class Continuations</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Threads (Cooperative)</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td><strong>Other Programming Language Facilities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guards</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Assertions</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Invariants</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>
# A Survey of the State of the Art

<table>
<thead>
<tr>
<th></th>
<th>Chained Context Reactions</th>
<th>Context-dependent Interruptions</th>
<th>Context-dependent Resumptions</th>
<th>Contextual Dispatch</th>
<th>Reactive Dispatch</th>
<th>Reactive Scope Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-Oriented Programming (COP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContextL and other Layer-based Languages</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Lambic</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Ambience</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Contextual Values</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Functional Reactive Programming (FRP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrTime and its Siblings</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Reactive SML</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Advanced Control Flow Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>First-class Continuations</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Threads (Cooperative)</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Other Programming Language Facilities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guards</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Assertions</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Invariants</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
A Survey of the State of the Art

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

<table>
<thead>
<tr>
<th></th>
<th>Chained Context Reactions</th>
<th>Context-dependent Interruptions</th>
<th>Context-dependent Resumptions</th>
<th>Contextual Dispatch</th>
<th>Reactive Dispatch</th>
<th>Reactive Scope Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-Oriented Programming (COP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContextL and other Layer-based Languages</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Lambic</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Ambience</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Contextual Values</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓ variable</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>Functional Reactive Programming (FRP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrTime and its Siblings</td>
<td>✓ behaviours and events</td>
<td>×</td>
<td>×</td>
<td>✓ same evaluation</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Reactive SML</td>
<td>×</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td><strong>Advanced Control Flow Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td>×</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>First-class Continuations</td>
<td>×</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Threads (Cooperative)</td>
<td>×</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>Other Programming Language Facilities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guards</td>
<td>×</td>
<td>✓ blocking at the beginning</td>
<td>✓ from the beginning</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Assertions</td>
<td>×</td>
<td>✓ explicit</td>
<td>× abort</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Invariants</td>
<td>×</td>
<td>✓ only at the beginning and the end</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
# A Survey of the State of the Art

<table>
<thead>
<tr>
<th></th>
<th>Chained Context Reactions</th>
<th>Context-dependent Interruptions</th>
<th>Context-dependent Resumptions</th>
<th>Contextual Dispatch</th>
<th>Reactive Dispatch</th>
<th>Reactive Scope Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-Oriented Programming (COP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContextL and other Layer-based Languages</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Lambic</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Ambience</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Contextual Values</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓ variable</td>
<td>✓</td>
<td>✓ scoped construct</td>
</tr>
<tr>
<td><strong>Functional Reactive Programming (FRP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrTime and its Siblings</td>
<td>✓ behaviours and events</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Reactive SML</td>
<td>✗</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Advanced Control Flow Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td>✗</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>First-class Continuations</td>
<td>✗</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Threads (Cooperative)</td>
<td>✗</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Other Programming Language Facilities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guards</td>
<td>✗</td>
<td>✓ blocking at the beginning</td>
<td>✓ from the beginning</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Assertions</td>
<td>✗</td>
<td>✓ explicit</td>
<td>✗ abort</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Invariants</td>
<td>✗</td>
<td>✓ only at the beginning and the end</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
## A Survey of the State of the Art

<table>
<thead>
<tr>
<th></th>
<th>Chained Context Reactions</th>
<th>Context-dependent Interruptions</th>
<th>Context-dependent Resumptions</th>
<th>Contextual Dispatch</th>
<th>Reactive Dispatch</th>
<th>Reactive Scope Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-Oriented Programming (COP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContextL and other Layer-based Languages</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Lambic</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Ambience</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Contextual Values</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✓ variable</td>
<td>✓ scoped construct</td>
<td></td>
</tr>
<tr>
<td><strong>Functional Reactive Programming (FRP) Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrTime and its Siblings</td>
<td>✓ behaviours and events</td>
<td>X</td>
<td>X</td>
<td>✓ explicit</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Reactive SML</td>
<td>X</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Advanced Control Flow Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td>X</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>First-class Continuations</td>
<td>X</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Threads (Cooperative)</td>
<td>X</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>✓ explicit</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Other Programming Language Facilities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guards</td>
<td>X</td>
<td>✓ blocking at the beginning</td>
<td>✓ from the beginning</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Assertions</td>
<td>X</td>
<td>✓ explicit</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Invariants</td>
<td>X</td>
<td>✓ only at the beginning and the end</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
There is a Need for a New COP Paradigm

<table>
<thead>
<tr>
<th></th>
<th>Chained Context Reactions</th>
<th>Context-dependent Interruptions</th>
<th>Context-dependent Resumptions</th>
<th>Contextual Dispatch</th>
<th>Reactive Dispatch</th>
<th>Reactive Scope Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-Oriented Programming Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContextL and other Layer-based Languages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ambience</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contextual Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Functional Reactive Programming Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrTime and its Siblings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reactive SML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Advanced Control Flow Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First-class Continuations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threads (Cooperative)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Other Programming Language Facilities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assertions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reactive Programming  
+ Context-Oriented Programming  
+ Implicit Interruptions

Reactive Method Dispatch for Context-Oriented Programming

ACM Computing Surveys (to appear)
There is a Need for a New COP Paradigm

<table>
<thead>
<tr>
<th>Category</th>
<th>Chained Context Reactions</th>
<th>Context-dependent Interruptions</th>
<th>Context-dependent Resumptions</th>
<th>Contextual Dispatch</th>
<th>Reactive Dispatch</th>
<th>Reactive Scope Management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-Oriented Programming Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContextL and other Layer-based Languages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ambience</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contextual Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Functional Reactive Languages</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reactive SML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Advanced Control Flow Constructs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First-class Continuations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threads (Cooperative)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Other Programming Language Facilities</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guards</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assertions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reactive Programming + Context-Oriented Programming + Implicit Interruptions

Reactive Method Dispatch for Context-Oriented Programming

Reactive Method Dispatch for Context-Oriented Programming
## The ICoDE Model: Design Space

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

Reactive Method Dispatch for Context-Oriented Programming
The ICoDE Model in Flute

Context sources as reactive values

to-call modal

call-from-car mode

call-from-conference-room mode

is-in-car?
is-in-conference-room?
Contextual and Reactive Dispatching

Context sources as reactive values

(to-call) → to-call modal

is-in-car?
call-from-car mode

is-in-conference-room?
call-from-conference-room mode
Contextual and Reactive Dispatching

Context sources as reactive values

(to-call) -> to-call modal ->

- is-in-car?
- call-from-car mode
- is-in-conference-room?
- call-from-conference-room mode
Contextual and **Reactive** Dispatching

**Context sources as reactive values**

- to-call **modal**
- **call-from-car** mode
- **call-from-conference-room** mode

**Questions:**
- is-in-car?
- is-in-conference-room?
Interruptible and Resumable Executions

(call-from-car)

(is-in-car? location)

(to-call)

executing

suspended

resumed

true

false

true again
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.

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.

Reactive Context Sources in Flute

```
(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.

Reactive Context Sources in Flute

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

(show-map nearby-metro-station)
**Procedure Modal:**

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

**Procedure Mode:**

```lisp
(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)))))))
```

**Reactive Context Source:**
- Modal
- Context predicate
- Interruption, resumption, and scoping strategies

**Shared Variable:**
- num-of-calls

**Procedure Modals and Modes in Flute**
Procedure Modals and Modes in Flute

**Procedure Modal:**
```
(define to-call (modal (location)
  (define num-of-calls 0)))
```

**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)))))))
```

```
(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))
```
Variable Modals and Modes in Flute

(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)
          (to-call))))))

Different value depending on the current context.
Scoped assignments

(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))))))))

(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))))))
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

<table>
<thead>
<tr>
<th>Interruption Strategies</th>
<th>Resumption Strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>abort</strong></td>
<td>restart</td>
</tr>
<tr>
<td><strong>suspend</strong></td>
<td></td>
</tr>
</tbody>
</table>
Scoping of State Changes to Ordinary Variables in Flute

(call-from-car)

(set! num-of-calls (+ num-of-calls 1))

- **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

iScheme

An Executable Semantics for Flute

Validation
iScheme: A Mobile Language Laboratory

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

Ambient-Oriented Programming

Language Experiments

iScheme

Objective-C

iOS

Linguistic Symbiosis

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

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

---

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 '()))
```
(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 '()))
(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)))

(continue after-dispatch #f '())))
Contextual and Reactive Dispatch Evaluation

```scheme
(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 '()))
```
(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 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 '()))
```
(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))))

(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))))

(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 '())

(continue true-count-start #f '())
(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)))))

(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))))
(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 '()))

(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))))
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 '())
```

(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))))
An Executable Semantics for Flute
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

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

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

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

\texttt{(eval context-pred continue-after-context-pred env \texttt{#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

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.


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

<table>
<thead>
<tr>
<th>Language Construct</th>
<th>Satisfied Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx-event</td>
<td>✓ Chained context reactions</td>
</tr>
<tr>
<td>mode</td>
<td>✓ Contextual dispatch</td>
</tr>
<tr>
<td>modal</td>
<td>✓ Reactive dispatch</td>
</tr>
<tr>
<td>suspend</td>
<td>✓ Context-dependent interruptions</td>
</tr>
<tr>
<td>abort</td>
<td>✓ Context-dependent resumptions</td>
</tr>
<tr>
<td>resume</td>
<td>✓ Reactive scope management</td>
</tr>
<tr>
<td>restart</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td></td>
</tr>
<tr>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>isolated</td>
<td></td>
</tr>
</tbody>
</table>
## Evaluation

<table>
<thead>
<tr>
<th>Language Construct</th>
<th>Satisfied Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx-event</td>
<td>✓ Chained context reactions</td>
</tr>
<tr>
<td></td>
<td>✓ Contextual dispatch</td>
</tr>
<tr>
<td></td>
<td>✓ Reactive dispatch</td>
</tr>
<tr>
<td>mode</td>
<td></td>
</tr>
<tr>
<td>modal</td>
<td></td>
</tr>
<tr>
<td>suspend</td>
<td>✓ Context-dependent interruptions</td>
</tr>
<tr>
<td>abort</td>
<td></td>
</tr>
<tr>
<td>resume</td>
<td>✓ Context-dependent resumptions</td>
</tr>
<tr>
<td>restart</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td>✓ Reactive scope management</td>
</tr>
<tr>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>isolated</td>
<td></td>
</tr>
</tbody>
</table>
## Evaluation

<table>
<thead>
<tr>
<th>Language Construct</th>
<th>Satisfied Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx-event</td>
<td>✓ Chained context reactions</td>
</tr>
<tr>
<td>mode</td>
<td>✓ Contextual dispatch</td>
</tr>
<tr>
<td>modal</td>
<td>✓ Reactive dispatch</td>
</tr>
<tr>
<td>suspend</td>
<td>✓ Context-dependent interruptions</td>
</tr>
<tr>
<td>abort</td>
<td>✓ Context-dependent resumptions</td>
</tr>
<tr>
<td>resume</td>
<td>✓ Reactive scope management</td>
</tr>
<tr>
<td>restart</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td></td>
</tr>
<tr>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>isolated</td>
<td></td>
</tr>
</tbody>
</table>
## Evaluation

<table>
<thead>
<tr>
<th>Language Construct</th>
<th>Satisfied Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx-event</td>
<td>✓ Chained context reactions</td>
</tr>
<tr>
<td>mode</td>
<td>✓ Contextual dispatch</td>
</tr>
<tr>
<td>modal</td>
<td>✓ Reactive dispatch</td>
</tr>
<tr>
<td>suspend, abort, resume, restart</td>
<td>✓ Context-dependent interruptions</td>
</tr>
<tr>
<td>deferred, immediate, isolated</td>
<td>✓ Context-dependent resumptions</td>
</tr>
<tr>
<td></td>
<td>✓ Reactive scope management</td>
</tr>
</tbody>
</table>
### Evaluation

<table>
<thead>
<tr>
<th>Language Construct</th>
<th>Satisfied Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx-event</td>
<td>✓ Chained context reactions</td>
</tr>
<tr>
<td>mode</td>
<td>✓ Contextual dispatch</td>
</tr>
<tr>
<td>modal</td>
<td>✓ Reactive dispatch</td>
</tr>
<tr>
<td>suspend</td>
<td>✓ Context-dependent interruptions</td>
</tr>
<tr>
<td>abort</td>
<td></td>
</tr>
<tr>
<td>resume</td>
<td></td>
</tr>
<tr>
<td>restart</td>
<td></td>
</tr>
<tr>
<td>deferred</td>
<td>✓ Reactive scope management</td>
</tr>
<tr>
<td>immediate</td>
<td></td>
</tr>
<tr>
<td>isolated</td>
<td></td>
</tr>
</tbody>
</table>
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 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.
Supporting Publications


