From Events to Reactions: 
A Progress Report

Tony Garnock-Jones 
tonyg@ccs.neu.edu
Northeastern University

Joint work with Matthias Felleisen and Sam Caldwell
Networked Program with coordination + internal tasks

Interactivity $\Rightarrow$ External Concurrency
Interactivity $\Rightarrow$ External Concurrency
Internal Organisation Reflects External Concurrency

Networked Program with coordination + internal tasks
Interactivity $\Rightarrow$ External Concurrency
Component startup $\Rightarrow$ interaction $\Rightarrow$ shutdown/failure
Interactivity $\Rightarrow$ External Concurrency

Component startup $\rightarrow$ interaction $\rightarrow$ shutdown/failure
Syndicate DSL by example

- Mapping events to components
- Managing conversational state
- Monitoring changes in shared state
SYNDICATE
event \times \text{state} \rightarrow [\text{action}] \times \text{state}
event × state → [action] × state

[currentScore, 3] → actor #17
[keyDown, space] → actor #42
event $\times$ state $\rightarrow$ [action] $\times$ state

Actor behaviour function
private state
unique internal ID

[currentScore, 3] $\rightarrow$ actor #17
$\rightarrow$ actor #42
event × state → [action] × state

Dataspace: assertions + provenance

cf. Linda's "Tuplespaces"
event × state → [action] × state

actor #17
actor #42
actor #94

[currentScore, 3] → actor #17
[keyDown, space] → actor #42

“I, actor #17, assert that the current score is 3.”
event × state → [action] × state

actor #17

actor #42

actor #94

[currentScore, 3] → actor #17
[keyDown, space] → actor #42

"I, actor #42, assert that the space key is currently held down."
Actions carry added and removed assertions actor $\rightarrow$ environment

- actor #17
- actor #42
- actor #94

- $\text{[currentScore, 3]} \rightarrow \text{actor #17}$
- $\text{[keyDown, space]} \rightarrow \text{actor #42}$
Events carry added and removed assertions 
environment → actor

[currentScore, 3] → actor #17
[keyDown, space] → actor #42
assert( [sprite, player, 51, 100, ] ),
assert( ?[keyDown, ★] )

[currentScore, 3] → actor #17
[keyDown, space] → actor #42
assert([sprite, player, 51, 100, 17]),
assert(?keyDown,★)
event × state → [action] × state

“I, actor #94, am interested in keeping track of assertions of the form [keyDown, ★].”
event × state → [action] × state

assert( [keyDown, space] )
event × state → [action] × state

actor #17

actor #42

actor #94

[currentScore,3] → actor #17
[keyDown,space] → actor #42
[sprite,player,51,100, ★] → actor #94
?[keyDown,★] → actor #94
event × state → [action] × state

retract([keyDown, space])

actor #17

actor #42

actor #94

[currentScore, 3] → actor #17
[keyDown, space] → actor #42
[sprite, player, 51, 100, 🌟] → actor #94
?[keyDown, ★] → actor #94
event × state → [action] × state

retract([keyDown, space])

actor #17

actor #42

actor #94

[currentScore, 3] → actor #17
[keyDown, space] → actor #42
[sprite, player, 51, 100, 🧸] → actor #94
?[keyDown, ★] → actor #94
event × state → [action] × state

actor #17
actor #42
actor #94

[currentScore, 3] → actor #17
[sprite, player, 51, 100, 🍀] → actor #94
?[keyDown, ★] → actor #94
event × state → [action] × state

retract( [keyDown, space] )

actor #17

actor #42

actor #94

[currentScore, 3] → actor #17

[sprite, player, 51, 100, player1] → actor #94

?[keyDown, ★] → actor #94
event × state → [action] × state

actor #17

actor #42

actor #94

[currentScore, 3] → actor #17

[sprite, player, 51, 100, 🐠] → actor #94

?[keyDown, ★] → actor #94
event × state → [action] × state

actor #17

[currentScore,3] → actor #17

[sprite,player,51,100, 🧙‍♀️] → actor #94

?[keyDown,★] → actor #94
event × state → [action] × state
event × state → [action] × state
event × state → [action] × state
event × state → [action] × state

[currentScore, 3] → actor #17
[sprite, player, 51, 100, 🎉] → actor #94
?[keyDown, ★] → actor #94
event × state → [action] × state
event × state → [action] × state
event × state → [action] × state

- [currentScore,3] → actor #17
- [sprite,player,51,100,()] → actor #94
- ?[keyDown,★] → actor #94

- X
- ↓ X
- ↓ ↓ X
- ↓ ↓ ↓ X
Messages are transient assertions

\[
< [\text{incrementScoreBy},3] > \\
\sim \\
\text{assert}( [\text{incrementScoreBy},3] ) \\
\text{followed by} \\
\text{retract}( [\text{incrementScoreBy},3] )
\]

(See “Coordinated Concurrent Programming in Syndicate” (ESOP 2016) for full detail of the semantics)
Syndicate Implementations

Racket macros & support library
#lang syndicate

Ohm-based translation to ECMAScript 5
Browser & node
Syndicate Implementations

Racket macros & support library
#lang syndicate

Ohm-based translation
to ECMAScript 5
Browser & node
Syndicate DSL by example

- Mapping events to components
- Managing conversational state
- Monitoring changes in shared state
Mapping events to components

controller

→ "start" checker
→ player
→ player
→ player
→ collision detector

↑ game

↑ game

?(controller-event 'start #t) → game
?(controller-event 'left ★) → game
?(clock-tick) → game
(sprite 5 'player) → game

↑ game

?(controller-event 'right ★) → player
?(clock-tick) → player
(sprite 5 'player) → player

↑ game

?(controller-event 'start #t) → game
?(controller-event 'left ★) → game
?(clock-tick) → game
(sprite 5 'player) → game
Mapping events to components

Interest in START presses at next outer dataspace

?(controller-event 'start #t) → “start” checker
?(controller-event 'left ⋆) → player
?(clock-tick) → player
?(game-piece-state ⋆ ⋆) → collision detector

?(controller-event 'start #t) → game
?(controller-event 'left ⋆) → game
?(clock-tick) → game
(sprite 5 'player) → game
Mapping events to components

Interest in LEFT presses/releases at next dataspace
Mapping events to components

```
<table>
<thead>
<tr>
<th>Event</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;start&quot;</td>
<td>checker</td>
</tr>
<tr>
<td>?(controller-event 'start #t)</td>
<td>&quot;start&quot; checker</td>
</tr>
<tr>
<td>?(controller-event 'left ⋆)</td>
<td>player</td>
</tr>
<tr>
<td>?(clock-tick)</td>
<td>player</td>
</tr>
<tr>
<td>?(sprite 5 'player)</td>
<td>player</td>
</tr>
<tr>
<td>?(game-piece-state 'player 5)</td>
<td>player</td>
</tr>
<tr>
<td>?(controller-event 'start #t)</td>
<td>game</td>
</tr>
<tr>
<td>?(controller-event 'left ⋆)</td>
<td>game</td>
</tr>
<tr>
<td>?(clock-tick)</td>
<td>game</td>
</tr>
<tr>
<td>(sprite 5 'player)</td>
<td>game</td>
</tr>
</tbody>
</table>

Interest in clock ticks at next outer dataspace
```
Mapping events to components

controller

driver

“start” checker

player

collision detector

⇃

?(controller-event 'start #t) → “start” checker

?(controller-event 'left ★) → player

?(clock-tick) → player

?(sprite 5 'player) → player

(game-piece-state player 5) → player

?(game-piece-state) → collision detector

↑

game

Sprite published to next outer dataspace

?(controller-event 'start #t) → game

?(controller-event 'left ★) → game

?(clock-tick) → game

(sprite 5 'player) → game
Mapping events to components

controller
driver

“start”
checker

player
collision
detector

⇃
?(controller-event ’start #t)
→ “start” checker
⇃
?(controller-event ’left ★)
→ player
⇃
?(clock-tick)
→ player
⇃
(sprite 5 ’player)
→ player
⇃
(game-piece-state ’player 5)
→ player
⇃
?(game-piece-state ★ ★)
→ collision detector

↑
?(controller-event ’start #t)
→ game
⇃
?(controller-event ’left ★)
→ game
⇃
?(clock-tick)
→ game
⇃
(sprite 5 ’player)
→ game

Game-piece state published locally

↑
game
Mapping events to components

controller

checker

player

collision

detector

 проведения событий к компонентам

?(controller-event 'start #t) → "start" checker
?(controller-event 'left ★) → player
?(clock-tick) → player
(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector

↑ game

Subscription to game-piece states

?(controller-event 'start #t) → game
?(controller-event 'left ★) → game
?(clock-tick) → game
(sprite 5 'player) → game
Mapping events to components

Interest in START presses at local dataspace

? (controller-event 'start #t) → game
? (controller-event 'left ★) → game
? (clock-tick) → game
(sprite 5 'player) → game
(game-piece-state 'player 5) → player
(game-piece-state ★ ★) → collision detector
Mapping events to components

```
?(controller-event 'start #t) → "start" checker
?(controller-event 'left ⭐) → player
?(clock-tick) → player
?(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ⭐ ⭐) → collision detector
```

```
?(controller-event 'start #t) → game
?(controller-event 'left ⭐) → game
?(clock-tick) → game
?(sprite 5 'player) → game
```

Interest in LEFT presses/releases at local dataspace
Mapping events to components

Controller driver

"Start" checker

Player

Collision detector

…

\( ?(\text{controller-event 'start} \ #t) \rightarrow \text{"start" checker} \)
\( ?(\text{controller-event 'left} \ ★) \rightarrow \text{player} \)
\( ?(\text{clock-tick}) \rightarrow \text{player} \)
\( ?(\text{sprite 5 'player}) \rightarrow \text{player} \)
\( ?(\text{game-piece-state 'player 5}) \rightarrow \text{player} \)
\( ?(\text{game-piece-state} \ ★ \ ★) \rightarrow \text{collision detector} \)

Interest in clock ticks at local dataspace

\( ?(\text{controller-event 'start} \ #t) \rightarrow \text{game} \)
\( ?(\text{controller-event 'left} \ ★) \rightarrow \text{game} \)
\( ?(\text{clock-tick}) \rightarrow \text{game} \)
\( ?(\text{sprite 5 'player}) \rightarrow \text{game} \)
Mapping events to components

- Controller
- Driver
- "Start" checker
- Player
- Collision detector

 Assertion of sprite position at local dataspace

```
(defvar game (game))
(defvar player (sprite 5 'player))
(defvar player-state (game-piece-state 'player 5))
(defvar collision (game-piece-state 'player))
```

```
(destructuring-bind (event) (controller-event) (game)
 (case event
  ('start #t) ("start" checker)
  ('left *) (player)
  ('clock-tick) (player)
  (sprite 5 'player) (player)
  (game-piece-state 'player 5) (player)
  (game-piece-state 'player) (collision detector)
  (clock-tick) (game)
  (sprite 5 'player) (game)
  (default) (error))
```
Mapping events to components

```
…

(controller-event 'start #t) → "start" checker
(controller-event 'left ★) → player
(clock-tick) → player
(sprite 5 'player) → player
(game-piece-state 'player 5) → player
(game-piece-state ★ ★) → collision detector
```

```
(game-piece-state ★ ★) → collision detector
```

```
controller
driver
```

```
…
```

```
(car-deck) → game
card-rendering → game
…
```

```
(game-piece-state ★ ★) → collision detector
```

```
…
```

```
(controller-event 'start #t) → game
(controller-event 'left ★) → game
(clock-tick) → game
(sprite 5 'player) → game
```

```
…
```

```
…
```
Mapping events to components

- **Pressing the START key should terminate the game**

  ```lisp
  (controller-event 'start #t) → "start" checker
  (controller-event 'left ★) → player
  (clock-tick) → player
  (sprite 5 'player) → player
  (game-piece-state 'player 5) → player
  (game-piece-state ★ ★) → collision detector
  ```

- **Game**

  ```lisp
  (sprite 5 'player) → game
  ```
Mapping events to components:

< (controller-event 'start #t) >

- (controller-event 'start #t) → “start” checker
- (controller-event 'left ★) → player
- (clock-tick) → player
- (sprite 5 'player) → player
- (game-piece-state 'player 5) → player
- (game-piece-state ★ ★) → collision detector

↑ game
Mapping events to components

< (controller-event 'start #t) >

- ?(controller-event 'start #t) → “start” checker
- ?(controller-event 'left ⋄) → player
- ?(clock-tick) → player
- (sprite 5 'player) → player
- ?(game-piece-state 'player ⋄ ⋄) → collision detector

↑ game

- ?(controller-event 'start #t) → game
- ?(controller-event 'left ⋄) → game
- ?(clock-tick) → game
- (sprite 5 'player) → game
< (controller-event 'start #t) >

- ?(controller-event 'start #t) → “start” checker
- ?(controller-event 'left ★) → player
- ?(clock-tick) → player
- (sprite 5 'player) → player
- (game-piece-state 'player 5) → player
- ?(game-piece-state ★ ★) → collision detector

↑ game

↓(controller-event 'start #t) → game
↓(controller-event 'left ★) → game
↓(clock-tick) → game
(sprite 5 'player) → game
Mapping events to components

- **Controller**
  - `(?controller-event 'start #t)` → "start" checker
  - `(?controller-event 'left ✫)` → player
  - `(?clock-tick)` → player
  - `?(sprite 5 'player)` → player
  - `(game-piece-state 'player 5)` → player
  - `?(game-piece-state ✫ ✫)` → collision detector

- **Game**
  - `(?controller-event 'start #t)` → game
  - `(?controller-event 'left ✫)` → game
  - `(?clock-tick)` → game
  - `?(sprite 5 'player)` → game

- **Quit-dataspace**
Mapping events to components

- **controller**
  - "start" checker
  - player
  - collision detector

- **driver**
  - ?(controller-event 'start #t) → "start" checker
  - ?(controller-event 'left ⋅) → player
  - ?(clock-tick) → player
  - (sprite 5 'player) → player
  - (game-piece-state 'player 5) → player
  - ?(game-piece-state ⋅ ⋅) → collision detector

- **game**
  - ?(controller-event 'start #t) → game
  - ?(controller-event 'left ⋅) → game
  - ?(clock-tick) → game
  - (sprite 5 'player) → game
Mapping events to components

controller driver

?(controller event 'start #t)  game
?(controller event 'left ★)  game
?(clock tick)  game
(sprite 5 'player)  game
Mapping events to components

ccontroller
driver
(spawn-dataspace (spawn-start-button-monitor)
  (spawn-player)
  (spawn-collision-detection)
  ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
         (match-event evt
           [(message (at-meta
                         (controller-event 'start #t)))
            (transition state (quit-dataspace)))])))
  (void)
  (sub (controller-event 'start #t)
        #:meta-level 1))
(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
    (match-event evt
      [(message (at-meta
                   (controller-event 'start #t))
        (transition state (quit-dataspace)))]))
  (void)
  (sub (controller-event 'start #t)
    #:meta-level 1) ))
(spawn-dataspace (spawn-start-button-monitor)
 (spawn-player)
 (spawn-collision-detection)
 ...)

(define (spawn-start-button-monitor)
 (spawn (lambda (evt state)
         (match-event evt
           [(message (at-meta
                        (controller-event 'start #t)))
            (transition state (quit-dataspace)))]))
 (void)
 (sub (controller-event 'start #t)
       #:meta-level 1))
(spawn-dataspace (spawn-start-button-monitor)
  (spawn-player)
  (spawn-collision-detection)
  ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
      (match-event evt
        [(message (at-meta
                  (controller-event 'start #t)))
          (transition state (quit-dataspace)))]))
  (void)
  (sub (controller-event 'start #t)
    #:meta-level 1))
(spawn-dataspace (spawn-start-button-monitor) 
  (spawn-player) 
  (spawn-collision-detection) 
  ...)

(define (spawn-start-button-monitor) 
  (spawn (lambda (evt state) 
    (match-event evt 
      [(message (at-meta 
        (controller-event 'start #t))) 
       (transition state (quit-dataspace))]))) 
  (void) 
  (sub (controller-event 'start #t) 
    #:meta-level 1))
(spawn-dataspace (spawn-start-button-monitor)
  (spawn-player)
  (spawn-collision-detection)
...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
    (match-event evt
      [(message (at-meta
        (controller-event 'start #t)))
       (transition state (quit-dataspace)))]))
  (void)
  (sub (controller-event 'start #t)
    #:meta-level 1))
(spawn-dataspace (spawn-start-button-monitor)
 (spawn-player)
 (spawn-collision-detection)
 ...)

(define (spawn-start-button-monitor)
 (spawn (lambda (evt state)
 (match-event evt
  [(message (at-meta
    (controller-event 'start #t)))
   (transition state (quit-dataspace)))]))
 (void)
 (sub (controller-event 'start #t)
   #:meta-level 1))
(spawn-dataspace (spawn-start-button-monitor)
 (spawn-player)
 (spawn-collision-detection)
 ...)

(define (spawn-start-button-monitor)
 (spawn (lambda (evt state)
    (match-event evt
     [(message (at-meta
      [(controller-event 'start #t))]
      (transition state (quit-dataspace)))]))
 (void)
 (sub (controller-event 'start #t)
    #:meta-level 1))
)
(spawn-dataspace (spawn-start-button-monitor)
 (spawn-player)
 (spawn-collision-detection)
 ...)

(define (spawn-start-button-monitor)
 (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                          (controller-event 'start #t)))
              (transition state (quit-dataspace))])]
             (void)
             (sub (controller-event 'start #t)
                  #:meta-level 1))  ))
(spawn-dataspace (spawn-start-button-monitor) (spawn-player) (spawn-collision-detection) ...)

(define (spawn-start-button-monitor)
  (spawn (lambda (evt state)
           (match-event evt
             [(message (at-meta
                         (controller-event 'start #t)))
              (transition state (quit-dataspace)))]))
  (void)
  (sub (controller-event 'start #t
                          #:meta-level 1)
        ))

Dataspace lifetime not syntactically apparent
(spawn-dataspace (spawn-start-button-monitor)
 (spawn-player)
 (spawn-collision-detection)
 ...)

(define (spawn-start-button-monitor)
 (spawn (lambda (evt state)
           (match-event evt
              [(message (at-meta
                            (controller-event 'start #t)))
                (transition state (quit-dataspace)))]))
 (void)
 (sub (controller-event 'start #t)
       #:meta-level 1))
(spawn-dataspace (spawn-start-button-monitor)
(spawn-player)
(spawn-collision-detection)
...)

(define (spawn-start-button-monitor)
(spawn (lambda (evt state)
  (match-event evt
    [(message (at-meta
      (controller-event 'start #t)))
     (transition state (quit-dataspace)))]))
(void)
(sub (controller-event 'start #t)
  #:meta-level 1))

2× repetition of pattern
(spawn-dataspace (spawn-start-button-monitor)
 (spawn-player)
 (spawn-collision-detection)
 ...)

(define (spawn-start-button-monitor)
 (spawn (lambda (evt state)
 (match-event evt
 [(message (at-meta
 (controller-event 'start #t)))
 (transition state (quit-dataspace))]]))
 (void)
 (sub (controller-event 'start #t)
 #:meta-level 1))
(spawn-dataspace (spawn-start-button-monitor)
 (spawn-player)
 (spawn-collision-detection)
 ...)

(define (spawn-start-button-monitor)
 (spawn (lambda (evt state)
 (match-event evt
  [(message (at-meta
    (controller-event 'start #t)))
   (transition state (quit-dataspace))]]))
 (void)
 (sub (controller-event 'start #t)
   #:meta-level 1))

2× repetition of metalevel, in two styles
(dataspace (spawn-player)
  (spawn-collision-detection)
...
(u...
(dataspace (spawn-player)
  (spawn-collision-detection)
  ...
  (until (message (controller-event 'start #t)
    #:meta-level 1)))

Dataspace termination near dataspace startup
(dataspace (spawn-player)
  (spawn-collision-detection)
  ...
  (until (message (controller-event 'start #t)
                   #:meta-level 1)))

subscription/message pattern written once
(dataspace (spawn-player)
  (spawn-collision-detection)
  ...
  (until (message (controller-event 'start #t)
               #:meta-level 1)))

metalevel number written once, in one style
Syndicate DSL by example

- Mapping events to components
- Managing conversational state
- Monitoring changes in shared state
Managing conversational state

controller driver

“start” checker

player

collision detector

...(omitted edges)

(down) ?(controller-event 'start #t) → “start” checker
(down) ?(controller-event 'left ★) → player

(down) (sprite 5 'player) → player
(down) (game-piece-state 'player 5) → player
(down) ?(game-piece-state ★ ★) → collision detector

↑game

?(controller-event 'start #t) → level
?(controller-event 'left ★) → level
Three jobs:
- watch state of left-arrow
- listen to clock-tick while arrow pressed
- maintain sprite & game-piece-state
Managing conversational state

< (controller-event 'left #t) >
Managing conversational state

< (controller-event 'left #t) >

```plaintext
rouch
p
collision
detector
	

↑ game

?@ (controller-event 'start #t) → "start" checker
?@ (controller-event 'left ⬤) → player

↓(sprite 5 'player) → player
(game-piece-state 'player ⬤) → player
?@ (game-piece-state ⬤ ⬤) → collision detector

?@ (controller-event 'start #t) → level
?@ (controller-event 'left ⬤) → level
```
Managing conversational state

```
< (controller-event 'left #t) >
```

```
↥ (controller-event 'start #t) ➔ "start" checker
↥ (controller-event 'left ★) ➔ player

↥ (sprite 5 'player) ➔ player
_PIXEL
↥ (game-piece-state 'player 5) ➔ player
↥ (?(game-piece-state ★ ★)) ➔ collision detector

↥ game

?(controller-event 'start #t) ➔ level
?(controller-event 'left ★) ➔ level
```
assert( \(\downarrow\)(clock-tick) )

```
verte
(controller-event 'start #t) \rightarrow "start" checker

verte
(controller-event 'left ✽) \rightarrow player

verte
(sprite 5 'player) \rightarrow player

verte
(game-piece-state 'player 5) \rightarrow player

verte
?(game-piece-state ✽ ✽) \rightarrow collision detector
```

```
verte
?(controller-event 'start #t) \rightarrow level

verte
?(controller-event 'left ✽) \rightarrow level
```
Managing conversational state

assert( \( ?(\text{clock-tick}) \) )
Managing conversational state

```
assert( ?(clock-tick) )
```

```
?•(controller-event 'start #t) → "start" checker
?•(controller-event 'left ★) → player
?•(clock-tick) → player
?•(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?•(game-piece-state ★ ★) → collision detector
```

```
?•(controller-event 'start #t) → level
?•(controller-event 'left ★) → level
```

```
↑game
```

Managing conversational state

class controller

"start" checker

(player)

collision detector

...
Managing conversational state

```
<table>
<thead>
<tr>
<th>controller driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>?(controller-event 'start #t) → level</td>
</tr>
<tr>
<td>?(controller-event 'left ★) → level</td>
</tr>
<tr>
<td>?(clock-tick) → level</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>„start“ checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>?(controller-event 'start #t) → „start“ checker</td>
</tr>
<tr>
<td>?(controller-event 'left ★) → player</td>
</tr>
<tr>
<td>?(clock-tick) → player</td>
</tr>
<tr>
<td>?(sprite 5 'player) → player</td>
</tr>
<tr>
<td>(game-piece-state 'player 5) → player</td>
</tr>
<tr>
<td>?(game-piece-state ★ ★) → collision detector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>player</th>
</tr>
</thead>
<tbody>
<tr>
<td>?(clock-tick) → player</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>collision detector</th>
</tr>
</thead>
<tbody>
<tr>
<td>?(game-piece-state ★ ★) → collision detector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>↑ game</th>
</tr>
</thead>
<tbody>
<tr>
<td>?(controller-event 'start #t) → level</td>
</tr>
<tr>
<td>?(controller-event 'left ★) → level</td>
</tr>
<tr>
<td>?(clock-tick) → level</td>
</tr>
</tbody>
</table>
```
Managing conversational state

< (controller-event 'left #f) >
Managing conversational state

< (controller-event 'left #f) >

controller driver

"start" checker

player

collision detector

↑ game

?(controller-event 'start #t) → "start" checker
?(controller-event 'left ▼) → player
?(clock-tick) → player
?(sprite 5 'player) → player
?(game-piece-state 'player *) → player
?(game-piece-state ★ *) → collision detector

?(controller-event 'start #t) → level
?(controller-event 'left ★) → level
?(clock-tick) → level
Managing conversational state

Controller

Driver

"Start" checker

Player

Collision detector

...(omitted)

<↓(controller-event 'left #f)>
Managing conversational state

```
(⇒ (controller-event 'start #t) → "start" checker
⇒ (controller-event 'left ★) → player
⇒ (clock-tick) → player
⇒ (sprite 5 'player) → player
⇒ (game-piece-state 'player 5) → player
⇒ (game-piece-state ★ ★) → collision detector
⇒ (controller-event 'start #t) → level
⇒ (controller-event 'left ★) → level
⇒ (clock-tick) → level
⇒ (controller-event 'left ★) → level
⇒ (controller-event 'left ★) → level
⇒ (controller-event 'left ★) → level
↑ game

↑game
```

(retract( ↑?(clock-tick) ) )
Managing conversational state

controller driver

"start" checker
collision detector
player

↓?(controller-event 'start #t) → "start" checker
↓?(controller-event 'left ★) → player
↑?(clock tick)
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ★ ★) → collision detector

↑game

?(controller-event 'start #t) → level
?(controller-event 'left ★) → level
?(clock-tick) → level
Managing conversational state

```prolog
retract( ?(clock-tick) )
```

```
<table>
<thead>
<tr>
<th>Event</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>(controller-event 'start #t)</td>
<td>&quot;start&quot; checker</td>
</tr>
<tr>
<td>(controller-event 'left ★)</td>
<td>player</td>
</tr>
<tr>
<td>(sprite 5 'player)</td>
<td>player</td>
</tr>
<tr>
<td>(game-piece-state 'player 5)</td>
<td>player</td>
</tr>
<tr>
<td>?(game-piece-state ★ ★)</td>
<td>collision detector</td>
</tr>
<tr>
<td>?(controller-event 'start #t)</td>
<td>level</td>
</tr>
<tr>
<td>?(controller-event 'left ★)</td>
<td>level</td>
</tr>
<tr>
<td>?(clock-tick)</td>
<td>level</td>
</tr>
</tbody>
</table>
```
Managing conversational state

controller
driver

“start”
checker

player

collision
detector

?(controller-event ’start #t) → “start” checker
?(controller-event ’left ⋄) → player

?(sprite 5 ’player) → player
(game-piece-state ’player 5) → player
?(game-piece-state ⋄ ⋄) → collision detector

game

?(controller-event ’start #t) → level
?(controller-event ’left ⋄) → level
?(clock tick) level
Managing conversational state

```
⇃(controller-event 'start #t) → "start" checker
⇃(controller-event 'left ★) → player

⇃(sprite 5 'player) → player
(game-piece-state 'player 5) → player
⇃(game-piece-state ★ ★) → collision detector

↑game

⇃(controller-event 'start #t) → level
⇃(controller-event 'left ★) → level
```
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
             (match-event evt
               [(message (at-meta (controller-event 'left pressed?)))
                (transition (struct-copy player-state state
                              [left-down? pressed?])
                             '()))
               [(message (at-meta (clock-tick)))
                (define new-state
                 (if (player-state-left-down? state)
                     (struct-copy player-state state
                                   [position (- (player-state-position state) 1)])
                     state))
                (define new-pos (player-state-position new-state))
                (transition new-state
                 (patch-seq (retract (sprite ? ?) #:meta-level 1)
                            (assert (sprite new-pos 'player) #:meta-level 1)
                            (retract (game-piece-state ? ?))
                            (assert (game-piece-state 'player new-pos))))]
             )
           )
       )
       (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                  (sub (clock-tick) #:meta-level 1)
                  (assert (sprite initial-pos 'player) #:meta-level 1)
                  (assert (game-piece-state 'player initial-pos))))
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
    (match-event evt
      [(message (at-meta (controller-event 'left pressed?)))
        (transition (struct-copy player-state state
                   [left-down? pressed?])
          '()))
      [(message (at-meta (clock-tick)))
        (define new-state
          (if (player-state-left-down? state)
            (struct-copy player-state state
                       [position (- (player-state-position state) 1)])
            state))
        (define new-pos (player-state-position new-state))
        (transition new-state
          (patch-seq (retract (sprite ? ?) #:meta-level 1)
          (assert (sprite new-pos 'player) #:meta-level 1)
          (retract (game-piece-state ? ?))
          (assert (game-piece-state 'player new-pos))))]
    initial-state
    (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
    (sub (clock-tick) #:meta-level 1)
    (assert (sprite initial-pos 'player) #:meta-level 1)
    (assert (game-piece-state 'player initial-pos))))
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
            (match-event evt
              [(message (at-meta (controller-event 'left pressed?)))
               (transition (struct-copy player-state state
                                 [left-down? pressed?]')())]
              [(message (at-meta (clock-tick)))
               (define new-state
                 (if (player-state-left-down? state)
                     (struct-copy player-state state
                                   [position (- (player-state-position state) 1)]) state))
               (define new-pos (player-state-position new-state))
               (transition new-state
                 (patch-seq (retract (sprite ? ?) #:meta-level 1)
                             (assert (sprite new-pos 'player) #:meta-level 1)
                             (retract (game-piece-state ? ?))
                             (assert (game-piece-state 'player new-pos)))));
               (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                         (sub (clock-tick) #:meta-level 1)
                         (assert (sprite initial-pos 'player) #:meta-level 1)
                         (assert (game-piece-state 'player initial-pos))))))
  initial-state)
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
          (match-event evt
            [(message (at-meta (controller-event 'left pressed?)))
             (transition (struct-copy player-state state
                           [left-down? pressed?])
                           '())]
            [(message (at-meta (clock-tick)))
             (define new-state
               (if (player-state-left-down? state)
                   (struct-copy player-state state
                                 [position (- (player-state-position state) 1)])
                   state))
             (define new-pos (player-state-position new-state))
             (transition new-state
               (patch-seq (retract (sprite ? ?) #:meta-level 1)
                           (assert (sprite new-pos 'player) #:meta-level 1)
                           (retract (game-piece-state ? ?))
                           (assert (game-piece-state 'player new-pos)))))
             initial-state
             (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                         (sub (clock-tick) #:meta-level 1)
                         (assert (sprite initial-pos 'player) #:meta-level 1)
                         (assert (game-piece-state 'player initial-pos))))))))
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
    (match-event evt
      [(message (at-meta (controller-event 'left pressed?)))
        (transition (struct-copy player-state state
                      [left-down? pressed?]
                    '()))
      [(message (at-meta (clock-tick)))
        (define new-state
          (if (player-state-left-down? state)
            (struct-copy player-state state
                          [position (- (player-state-position state) 1)]
                        state))
        (define new-pos (player-state-position new-state))
        (transition new-state
          (patch-seq (retract (sprite ? ?) #:meta-level 1)
            (assert (sprite new-pos 'player) #:meta-level 1)
            (retract (game-piece-state ? ?))
            (assert (game-piece-state 'player new-pos)))
          initial-state
          (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                        (sub (clock-tick) #:meta-level 1)
                        (assert (sprite initial-pos 'player) #:meta-level 1)
                        (assert (game-piece-state 'player initial-pos))))))
)
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
      (match-event evt
        [(message (at-meta (controller-event 'left pressed?)))
          (transition (struct-copy player-state state
            [left-down? pressed?]')())
        [(message (at-meta (clock-tick)))
          (define new-state
            (if (player-state-left-down? state)
              (struct-copy player-state state
                [position (- (player-state-position state state) 1)]) state))
          (define new-pos (player-state-position new-state))
          (transition new-state
            (patch-seq (retract (sprite ? ?) #:meta-level 1)
              (assert (sprite new-pos 'player) #:meta-level 1)
              (retract (game-piece-state ? ?))
              (assert (game-piece-state 'player new-pos))))]])))

initial-state
  (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
    (sub (clock-tick) #:meta-level 1)
    (assert (sprite initial-pos 'player) #:meta-level 1)
    (assert (game-piece-state 'player initial-pos))))
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
    (match-event evt
      [(message (at-meta (controller-event 'left pressed?)))
        (transition (struct-copy player-state state
                       [left-down? pressed?] '()))]
      [(message (at-meta (clock-tick)))
        (define new-state
          (if (player-state-left-down? state)
            (struct-copy player-state state
                           [position (- (player-state-position state) 1)])
            state))
        (define new-pos (player-state-position new-state))
        (transition new-state
          (patch-seq (retract (sprite ? ?) #:meta-level 1)
          (assert (sprite new-pos 'player) #:meta-level 1)
          (retract (game-piece-state ? ?))
          (assert (game-piece-state 'player new-pos)))))))))

(initial-state
  (patch-seq (sub (controller-event 'left?) #:meta-level 1)
    (sub (clock-tick) #:meta-level 1)
    (assert (sprite initial-pos 'player) #:meta-level 1)
    (assert (game-piece-state 'player initial-pos))))
(struct player-state (position left-down?))

(define (spawn-player)
  (define initial-pos 5)
  (define initial-state (player-state initial-pos #f))
  (spawn (lambda (evt state)
    (match-event evt
      [((message (at-meta (controller-event 'left pressed?))))
        (transition (struct-copy player-state state
                             [left-down? pressed?])
                     '()))
      [((message (at-meta (clock-tick))))
        (define new-state
          (if (player-state-left-down? state)
              (struct-copy player-state state
                            [position (- (player-state-position state state) 1)])
              state))
        (define new-pos (player-state-position new-state))
        (transition new-state
          (patch-seq (retract (sprite ? ?) #:meta-level 1)
                      (assert (sprite new-pos 'player) #:meta-level 1)
                      (retract (game-piece-state ? ?))
                      (assert (game-piece-state 'player new-pos)))
          initial-state
          (patch-seq (sub (controller-event 'left ?) #:meta-level 1)
                      (sub (clock-tick) #:meta-level 1)
                      (assert (sprite initial-pos 'player) #:meta-level 1)
                      (assert (game-piece-state 'player initial-pos)))))))
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position))
    (on (message (controller-event 'left #t)
      #:meta-level 1)
      (until (message (controller-event 'left #f)
        #:meta-level 1))
      (on (message (clock-tick) #:meta-level 1)
        (send! move-left)))))
  (on (message move-left)
    (- position 1)))))
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position))
    (on (message (controller-event 'left #t)
      #:meta-level 1)
      (until (message (controller-event 'left #f)
        #:meta-level 1)
        (on (message (clock-tick) #:meta-level 1)
          (send! move-left)))))
    (on (message move-left)
      (- position 1)))))
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)])
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position))
    (on (message (controller-event 'left #t) #:meta-level 1)
      (until (message (controller-event 'left #f) #:meta-level 1)
        (on (message (clock-tick) #:meta-level 1)
          (send! move-left)))))
    (on (message move-left)
      (- position 1)))
  )
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
      (assert (sprite position 'player) #:meta-level 1)
      (assert (game-piece-state 'player position))
      (on (message (controller-event 'left #t)
                   #:meta-level 1)
          (until (message (controller-event 'left #f)
                         #:meta-level 1))
          (on (message (clock-tick) #:meta-level 1)
               (send! move-left))))
  (on (message move-left)
       (- position 1)))))
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position)))
  (on (message (controller-event 'left #t)
      #:meta-level 1)
    (until (message (controller-event 'left #f)
      #:meta-level 1)
    (on (message (clock-tick) #:meta-level 1)
      (send! move-left))))
  (on (message move-left)
    (- position 1)))))
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)])
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position))
    (on (message (controller-event 'left #t)
                   #:meta-level 1)
      (until (message (controller-event 'left #f)
                     #:meta-level 1)
        (on (message (clock-tick) #:meta-level 1)
          (send! move-left)))
    (on (message move-left)
      (- position 1)))
  ))

Substate continues to apply until termination event triggered
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
      (assert (sprite position 'player) #:meta-level 1)
      (assert (game-piece-state 'player position))
      (on (message (controller-event 'left #t)
          #:meta-level 1)
        (until (message (controller-event 'left #f)
            #:meta-level 1)
          (on (message (clock-tick) #:meta-level 1)
            (send! move-left))))
      (on (message move-left)
        (- position 1)))))

Substate continues to apply until termination event triggered
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position))
    (on (message (controller-event 'left #t)
      #:meta-level 1)
      (until (message (controller-event 'left #f)
        #:meta-level 1)
        (on (message (clock-tick) #:meta-level 1)
          (send! move-left)))))
    (on (message move-left)
      (- position 1)))))
(define (spawn-player)
  (define move-left (gensym))
  (actor (forever #:collect [(position 5)]
         (assert (sprite position 'player) #:meta-level 1)
         (assert (game-piece-state 'player position)))
  (on (message (controller-event 'left #t)
             #:meta-level 1)
       (until (message (controller-event 'left #f)
                      #:meta-level 1)
               (on (message (clock-tick) #:meta-level 1)
                    (send! move-left)))))
  (on (message move-left)
       (- position 1)))))
(define (spawn-player)
  (define position 5)
  (actor (forever
    (assert (sprite position 'player) #:meta-level 1)
    (assert (game-piece-state 'player position))
    (on (message (controller-event 'left #t)
                  #:meta-level 1)
      (until (message (controller-event 'left #f)
                      #:meta-level 1)
        (on (message (clock-tick) #:meta-level 1)
          (set! position (- position 1))))))))
(define (spawn-player)
    (define position 5)
    (actor (forever
        (assert (sprite position 'player) #:meta-level 1)
        (assert (game-piece-state 'player position))
        (on (message (controller-event 'left #t)
                        #:meta-level 1)
            (until (message (controller-event 'left #f)
                            #:meta-level 1)
                (on (message (clock-tick) #:meta-level 1)
                    (set! position (+ position 1)))))))))
Syndicate DSL by example

✓ Mapping events to components
✓ Managing conversational state
• Monitoring changes in shared state
Monitoring changes in shared state

(clock driver)

“start” checker

player

collision detector

 gameTime

?(controller-event 'start #t) → “start” checker
?(controller-event 'left ⋄) → player
?(clock-tick) → player
?(sprite 5 'player) → player
(game-piece-state 'player 5) → player
?(game-piece-state ⋄ ⋄) → collision detector

↑game

?(controller-event 'start #t) → level
?(controller-event 'left ⋄) → level
?(clock-tick) → level
Monitoring changes in shared state

“start” checker  →  checker
⇃
?(controller-event 'start #t)  →  "start" checker
⇃
?(controller-event 'left ★)  →  player
⇃
?(clock-tick)  →  player
⇃
(sprite 5 'player)  →  player
⇃
(game-piece-state 'player 5)  →  player
⇃
?(game-piece-state ★ ★)  →  collision detector
⇃
(game-piece-state ★ ★)  →  collision detector
⇃
?(controller-event 'start #t)  →  level
⇃
?(controller-event 'left ★)  →  level
⇃
?(clock-tick)  →  level
⇃

Must maintain an index over game-piece-state records asserted by other actors
Monitoring changes in shared state

< (clock-tick) >

(clock-driver)

“start” checker

player

collision detector

↓(? (controller-event 'start #t)) → "start" checker
↓(? (controller-event 'left ⋆)) → player
↓(? (clock-tick)) → player
↓(sprite 5 'player) → player
(game-piece-state 'player 5) → player
(? (game-piece-state ⋆ ⋆)) → collision detector

↑game

↑(controller-event 'start #t) → level
(? (controller-event 'left ⋆)) → level
(? (clock-tick)) → level
Monitoring changes in shared state

< (clock-tick) >

```
;;; clock

(clock-tick)

;;; player

(player)

;;; "start" checker

("start" checker)

;;; collision detector

(collision detector)

;;; game

(game)

;;; level

(level)
```

```

;;; controller-event

(controller-event 'start #t)

(controller-event 'left ★)

(clock-tick)

(sprite 5 'player)

(game-piece-state 'player)

(game-piece-state ★ ★)

?(controller-event 'start #t) → "start" checker

?(controller-event 'left ★) → player

?(clock-tick) → player

?(sprite 5 'player) → player

?(game-piece-state 'player) → player

?(game-piece-state ★ ★) → collision detector

?(controller-event 'start #t) → level

?(controller-event 'left ★) → level

?(clock-tick) → level
```
Monitoring changes in shared state

- `锺 clock` driver
- `锺 “start” checker` player
- `锺 collision detector` ...

锺 \(\downarrow\) (controller-event ‘start #t) → “start” checker
锺 \(\downarrow\) (controller-event ‘left ⋆) → player
锺 \(\downarrow\) (clock-tick) → player
锺 \(\downarrow\) (sprite 5 ‘player) → player
锺 (game-piece-state ‘player 5) → player
锺 ?(game-piece-state ⋆ ⋆) → collision detector

锺 \(\uparrow\) game

锺 ?(controller-event ‘start #t) → level
锺 ?(controller-event ‘left ⋆) → level
锺 ?(clock-tick) → level
retract( (sprite ★ ★) ),
assert( (sprite 4 'player) ),
retract( (game-piece-state ★ ★) ),
assert( (game-piece-state 'player 4) )
retract( (sprite ★ ★) ),
assert( (sprite 4 'player) ),
retract( (game-piece-state ★ ★) ),
assert( (game-piece-state 'player 4) )
retract( (sprite 5 'player) )
assert( (sprite 4 'player) )
retract( (game-piece-state 'player 5) ),
assert( (game-piece-state 'player 4) )

```
⇃(controller-event 'start #t) → "start" checker
⇃(controller-event 'left ★) → player
⇃(clock-tick) → player
⇃(sprite 4 'player) → player
⇃(game-piece-state 'player 4) → player
⇃(game-piece-state ★ ★) → collision detector
```

```
⇃(controller-event 'start #t) → level
⇃(controller-event 'left ★) → level
⇃(clock-tick) → level
```

```
↑game
```
Monitoring changes in shared state

- "start" checker
- player
- collision detector

Clock driver

```
锺?(controller-event 'start #t) → "start" checker
锺?(controller-event 'left ⭐) → player
锺?(clock-tick) → player
锺(sprite 4 'player) → player
锺(game-piece-state 'player 4) → player
锺?(game-piece-state ⋆ ⋆) → collision detector
```

```
锺?(controller-event 'start #t) → level
锺?(controller-event 'left ⭐) → level
锺?(clock-tick) → level
```

Game
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
    (match-event evt
      [(? patch? p)
        (define p0 (collision-detection-state-pieces state))
        (define p1 (for-trie/fold
          [(pieces p0)]
          [((game-piece-state $id _) (patch-removed p))]
          (hash-remove pieces id))]
        (define p2 (for-trie/fold [(pieces p1)]
          [((? piece (game-piece-state _ _)) (patch-added p))]
          (hash-set pieces (game-piece-state-id piece) piece))]
        (transition (struct-copy collision-detection-state state
          [pieces p2])
          '()))))
  (collision-detection-state (hash))
  (sub (game-piece-state ? ?))))
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
    (match-event evt
      [(? patch? p)
        (define p0 (collision-detection-state-pieces state))
        (define p1 (for-trie/fold
          [(pieces p0)]
          [(((game-piece-state $id _) (patch-removed p)))]
          (hash-remove pieces id)))]
      (define p2 (for-trie/fold [(pieces p1)]
        [((( $ piece (game-piece-state _ _)) (patch-added p)))]
        (hash-set pieces (game-piece-state-id piece) piece)))]
      (transition (struct-copy collision-detection-state state
        [pieces p2])
        '())))))
  (collision-detection-state (hash))
  (sub (game-piece-state ? ?))))
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                            [(pieces p0)]
                            [((game-piece-state $id _) (patch-removed p))]
                            (hash-remove pieces id))]
              (define p2 (for-trie/fold [(pieces p1)]
                            [((? piece (game-piece-state _ _)) (patch-added p))]
                            (hash-set pieces (game-piece-state-id piece) piece)))
              (transition (struct-copy collision-detection-state state
                             [pieces p2])
                            '())))
           (collision-detection-state (hash))
           (sub (game-piece-state ? ?))))))
Patch events describe whole sets of added and removed assertions, but programmers think about single assertions.
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
    (match-event evt
      [(? patch? p)
        (define p0 (collision-detection-state-pieces state))
        (define p1 (for-trie/fold
                     [(pieces p0)]
                     [(((game-piece-state $id _) (patch-removed p)))]
                     (hash-remove pieces id)))
        (define p2 (for-trie/fold [(pieces p1)]
                     [((($ piece (game-piece-state _ _)) (patch-added p))] (hash-set pieces (game-piece-state-id piece) piece)))
        (transition (struct-copy collision-detection-state state
                         [pieces p2])
                    '()))))
  (collision-detection-state (hash))
  (sub (game-piece-state ? ?))))
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
             (match-event evt
               [(? patch? p)
                (define p0 (collision-detection-state-pieces state))
                (define p1 (for-trie/fold
                               [(pieces p0)]
                               [(((game-piece-state $id _) (patch-removed p)))]
                               (hash-remove pieces id))]
                (define p2 (for-trie/fold [(pieces p1)]
                               [((( $ piece (game-piece-state _ _)) (patch-added p))]
                               (hash-set pieces (game-piece-state-id piece) piece))
                (transition (struct-copy collision-detection-state state
                               [pieces p2])
                             '()))))
             (collision-detection-state (hash))
             (sub (game-piece-state ? ?))))))
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
    (match-event evt
      [(? patch? p)
        (define p0 (collision-detection-state-pieces state))
        (define p1 (for-trie/fold
          [(pieces p0)]
          [((game-piece-state $id _) (patch-removed p))]
          (hash-remove pieces id))]
        (define p2 (for-trie/fold [(pieces p1)]
          [(($ piece (game-piece-state _ _)) (patch-added p))]`
          (hash-set pieces (game-piece-state-id piece) piece))]
        (transition (struct-copy collision-detection-state state
          [pieces p2])
          '())))))
  (collision-detection-state (hash))
  (sub (game-piece-state ? ?))))
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
    (match-event evt
      [(? patch? p)
       (define p0 (collision-detection-state-pieces state))
       (define p1 (for-trie/fold
          [(pieces p0)]
          [((game-piece-state $id _) (patch-removed p))]
          (hash-remove pieces id))]
      (define p2 (for-trie/fold [(pieces p1)]
        [(( $ piece (game-piece-state _ _)) (patch-added p))]
        (hash-set pieces (game-piece-state-id piece) piece)))
      (transition (struct-copy collision-detection-state state [pieces p2])
        '()))))
  (collision-detection-state (hash))
  (sub (game-piece-state ? ?))))
(struct collision-detection-state (pieces))

(define spawn-collision-detection
  (spawn (lambda (evt state)
           (match-event evt
             [(? patch? p)
              (define p0 (collision-detection-state-pieces state))
              (define p1 (for-trie/fold
                          [(pieces p0)]
                          [((game-piece-state $id _) (patch-removed p))]
                          (hash-remove pieces id)))]
              (define p2 (for-trie/fold [(pieces p1)]
                          [((? piece (game-piece-state _ _)) (patch-added p))]
                          (hash-set pieces (game-piece-state-id piece) piece)))]
              (transition (struct-copy collision-detection-state state
                          [pieces p2])))
            '()))))

(collision-detection-state (hash))
(sub (game-piece-state ? ?))))
(struct collision-detection-state (pieces))

(define (spawn-collision-detection)
  (spawn (lambda (evt state)
    (match-event evt
      [(? patch? p)
        (define p0 (for-trie/fold [(pieces p0)])
          (((game-piece-state $id _) (patch-removed p)))
            (hash-remove pieces id))
        (define p1 (for-trie/fold [(pieces p1)])
          ((($ piece (game-piece-state _ _)) (patch-added p)))
            (hash-set pieces (game-piece-state-id piece) piece)))
      (transition (struct-copy collision-detection-state state
            [pieces p2])
          '()))))
  (collision-detection-state (hash))
  (sub (game-piece-state ? ?))))
(define (spawn-collision-detection)
  (actor
    (forever #:collect [(pieces (hash))]
      (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
      (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece))))
(define (spawn-collision-detection)
  (actor
    (forever #$collect [(pieces (hash))]
      (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
      (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece))))
(define (spawn-collision-detection)
  (actor
    (forever #:collect [(pieces (hash))])
    (on (retracted (game-piece-state $id _))
      (hash-remove pieces id))
    (on (asserted ($ piece (game-piece-state _ _)))
      (hash-set pieces (game-piece-state-id piece) piece))))
(define (spawn-collision-detection)
  (actor
    (forever #:collect [(pieces (hash))]
      (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
      (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece))))))
(define (spawn-collision-detection)
  (actor
    (forever #:collect [(pieces (hash))]
      (on (retracted (game-piece-state $id _))
        (hash-remove pieces id))
      (on (asserted ($ piece (game-piece-state _ _)))
        (hash-set pieces (game-piece-state-id piece) piece))))

(actor
  (forever
    (query [pieces (hash id piece); “group-by”
      ($ piece (game-piece-state $id _))]
      (on (changed pieces) ...
          ...))))
Syndicate DSL by example

✓ Mapping events to components
✓ Managing conversational state
✓ Monitoring changes in shared state
Strong space savings in most places
syn·di·cate
a language for interactive programs

Progress Report on DSL Design

Repeated idioms → Language features

Future work:
- Improved state sharing with substates
- “Queries” (e.g. “group-by”)
- Non-naive compilation strategy
- Better technique for naming metalevels
- More evaluations & case studies

http://syndicate-lang.org/