Defining Semantic Variations of Diagrammatic Languages using Behavioral Programming and Queries

EXE, October 2016, Saint Malo

Michael Bar-Sinai barsinam@cs.bgu.ac.il @michbarsinai
Computer Science Department, Ben Gurion University of the Negev, Israel
Gera Weiss
Computer Science Department, Ben Gurion University of the Negev, Israel
Assaf Marron
Faculty of Mathematics and Computer Science, Weizmann Institute of Science, Israel

We present a methodology for describing semantics of diagrammatic languages, using **queries** and **generated**, Behavioral Programming-based **code**.

We present a methodology
for describing semantics of
diagrammatic languages, using **queries**
and **generated**,
Behavioral Programming-based **code**.

*We also present a **tool** demonstrating this approach using Live Sequence Charts (LSC).*

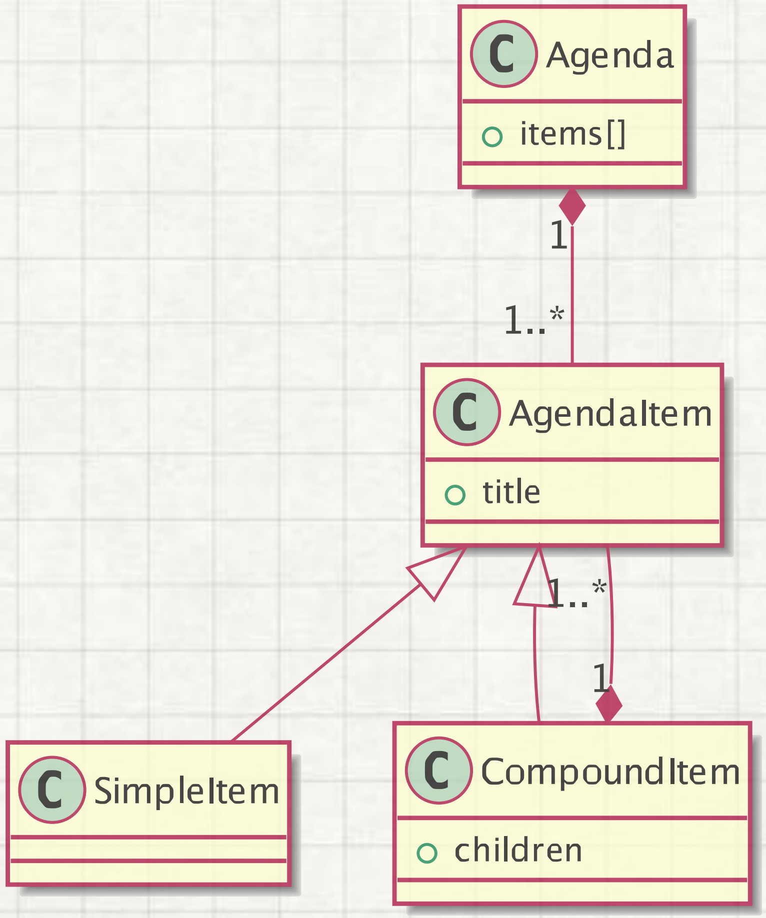
Problem Statement

- Diagrammatic programming languages hold great promise for software engineering
    - Have been holding this promise for a while
- Mostly not adopted by industry
    - Lack of approachable semantic definitions
    - Interoperation with existing tools and coding methodologies limited.

This paper focuses on languages with operations semantics; supporting languages with structural semantics is left to (near?) future work

Agenda

- Concept Overview
- Behavioral Programming 101
- Semantic Mapping to BP
- Discussion
- Case-Study: LSC
    - LSC 50
    - Visual Dictionaries
    - Implementation
    - Semantic Variations
- Related Work



Behavioral Programming 101 [1,2]

- Embedding scenario-based programming [3] concept in procedural languages
- A programming paradigm:
  compose a complex reactive system by interweaving threads of behavior (BThreads)
- An interaction protocol:
  BThreads can Request, Wait-for, and Block events
- A simple execution mechanism:
  Select event that is requested but not blocked, publish to all BThreads. Repeat.

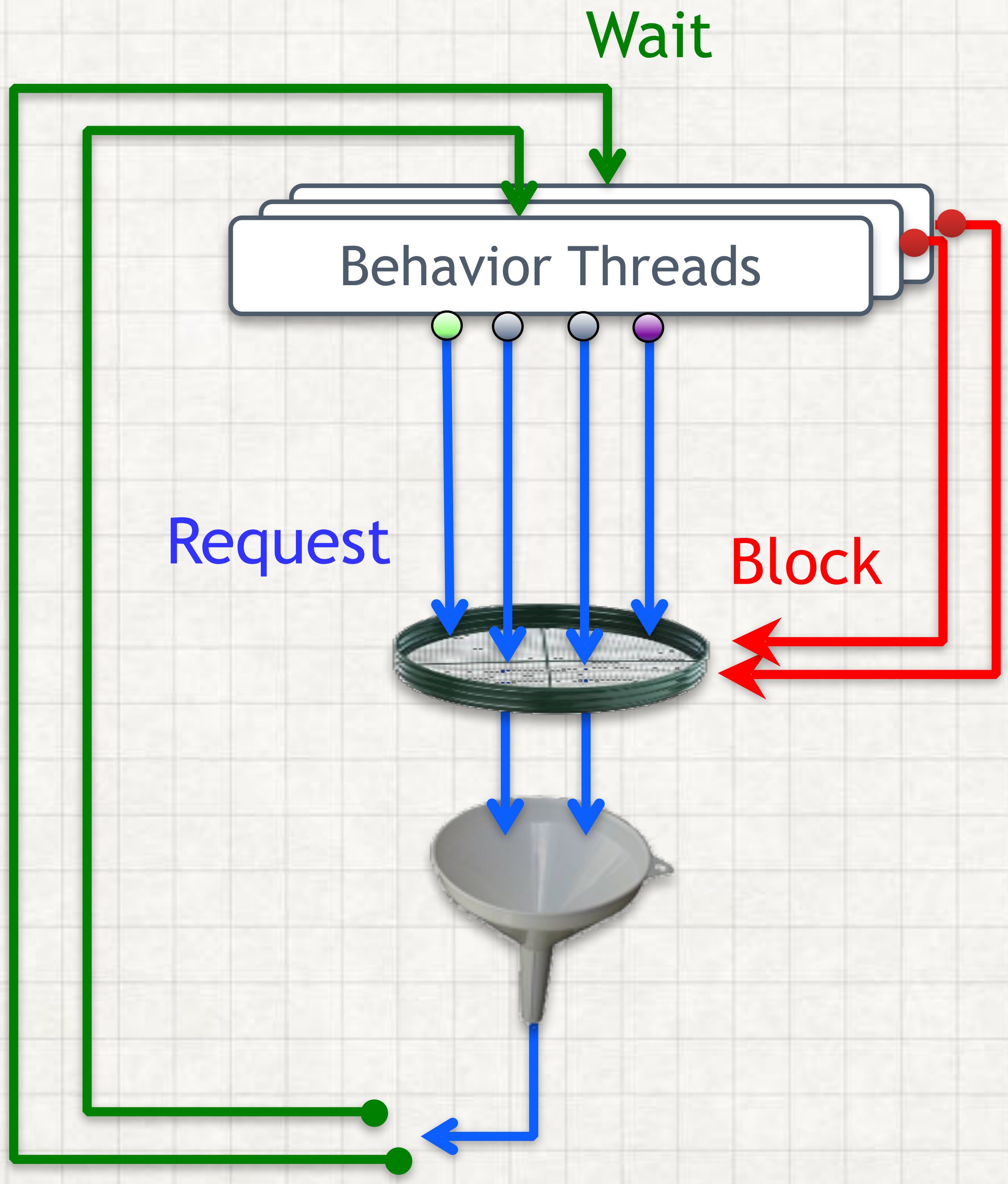
---

[1] D. Harel, A. Marron, and G. Weiss. Programming Coordinated Scenarios in Java. In Proc. 24th European Conf. on Object-Oriented Programming (ECOOP), pages 250–274, 2010.

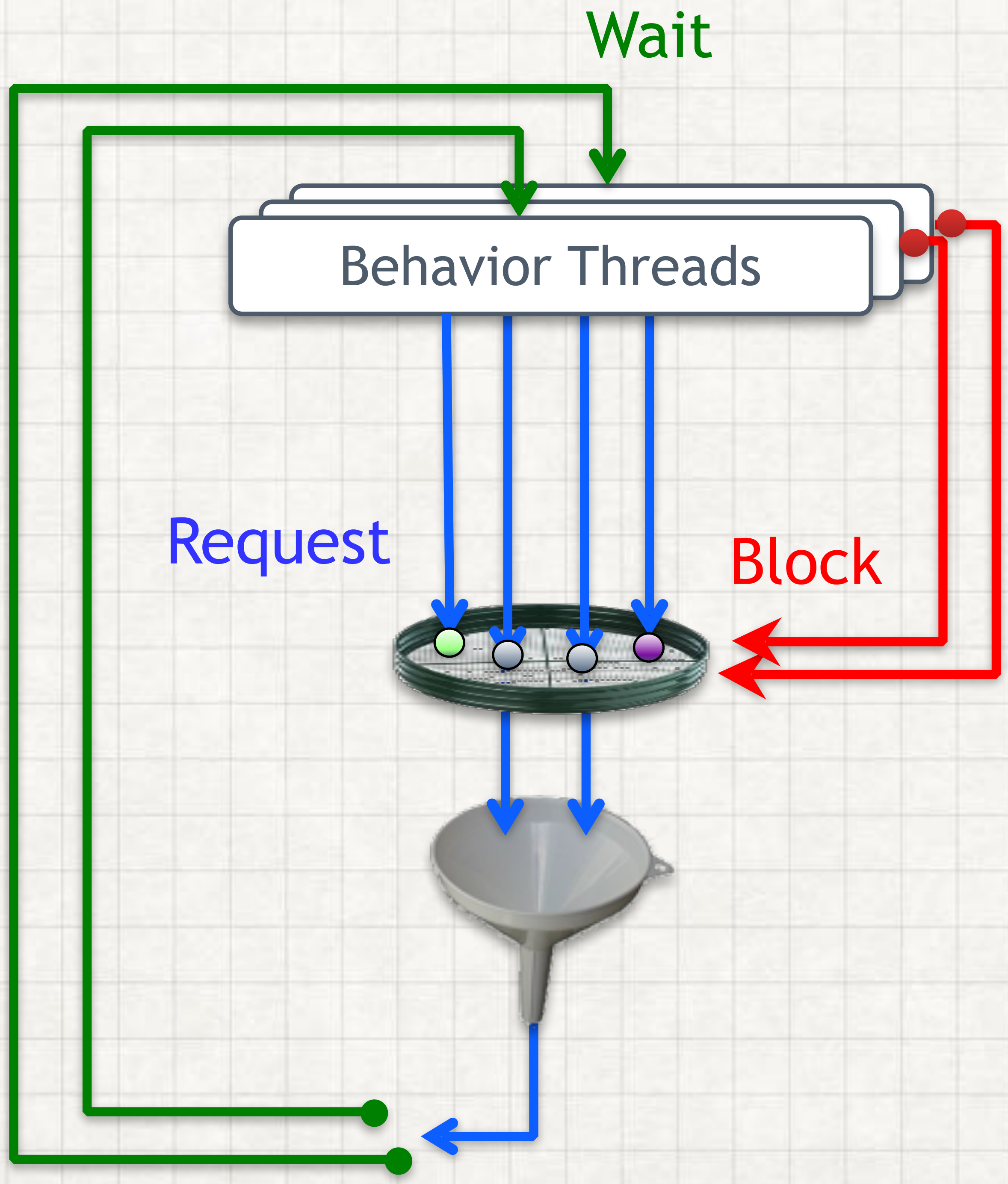
[2] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. Communications of the ACM, July 2012.

[3] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. J. on Formal Methods in System Design, 19(1):45–80, 2001.

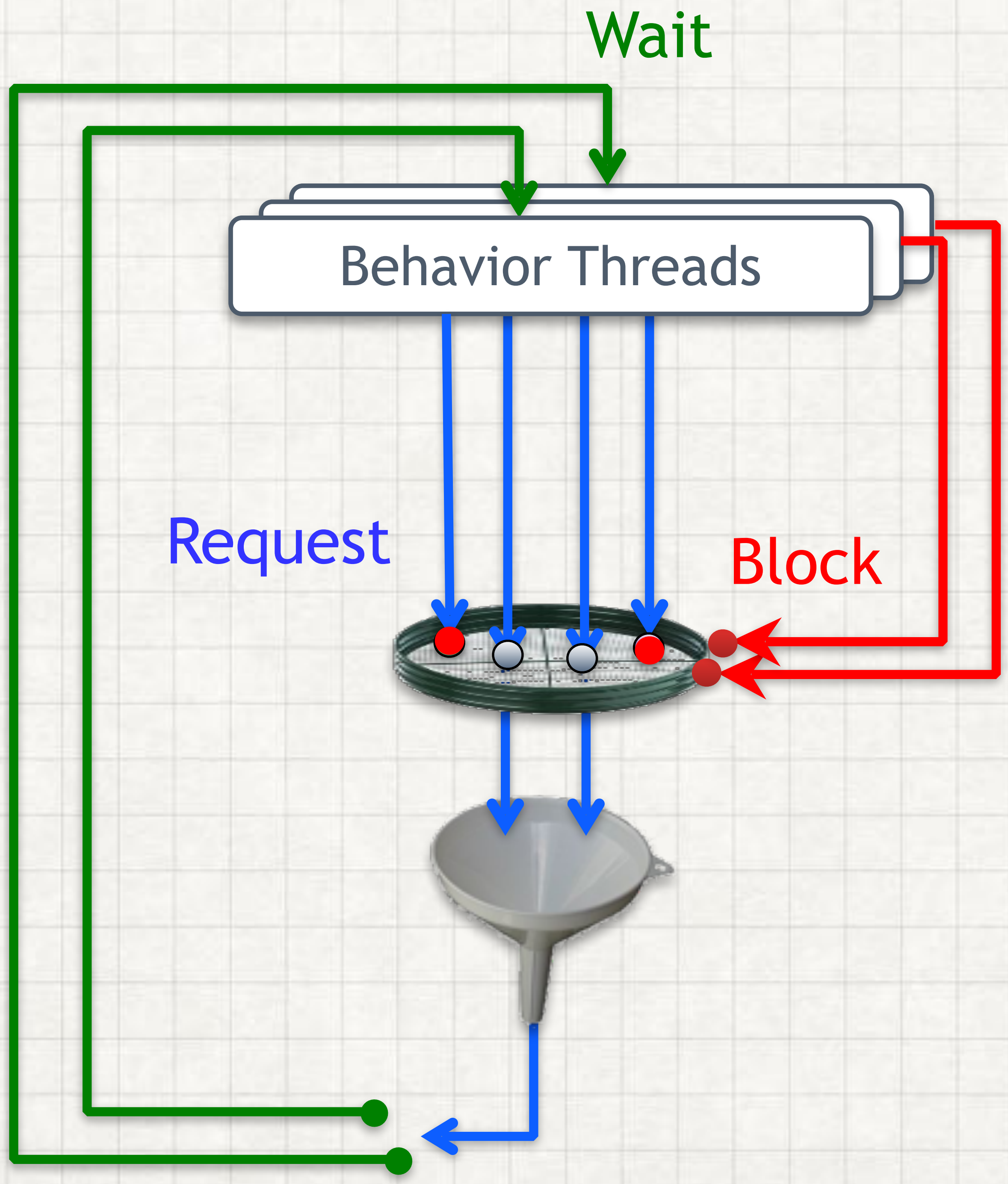
The BP Execution Cycle



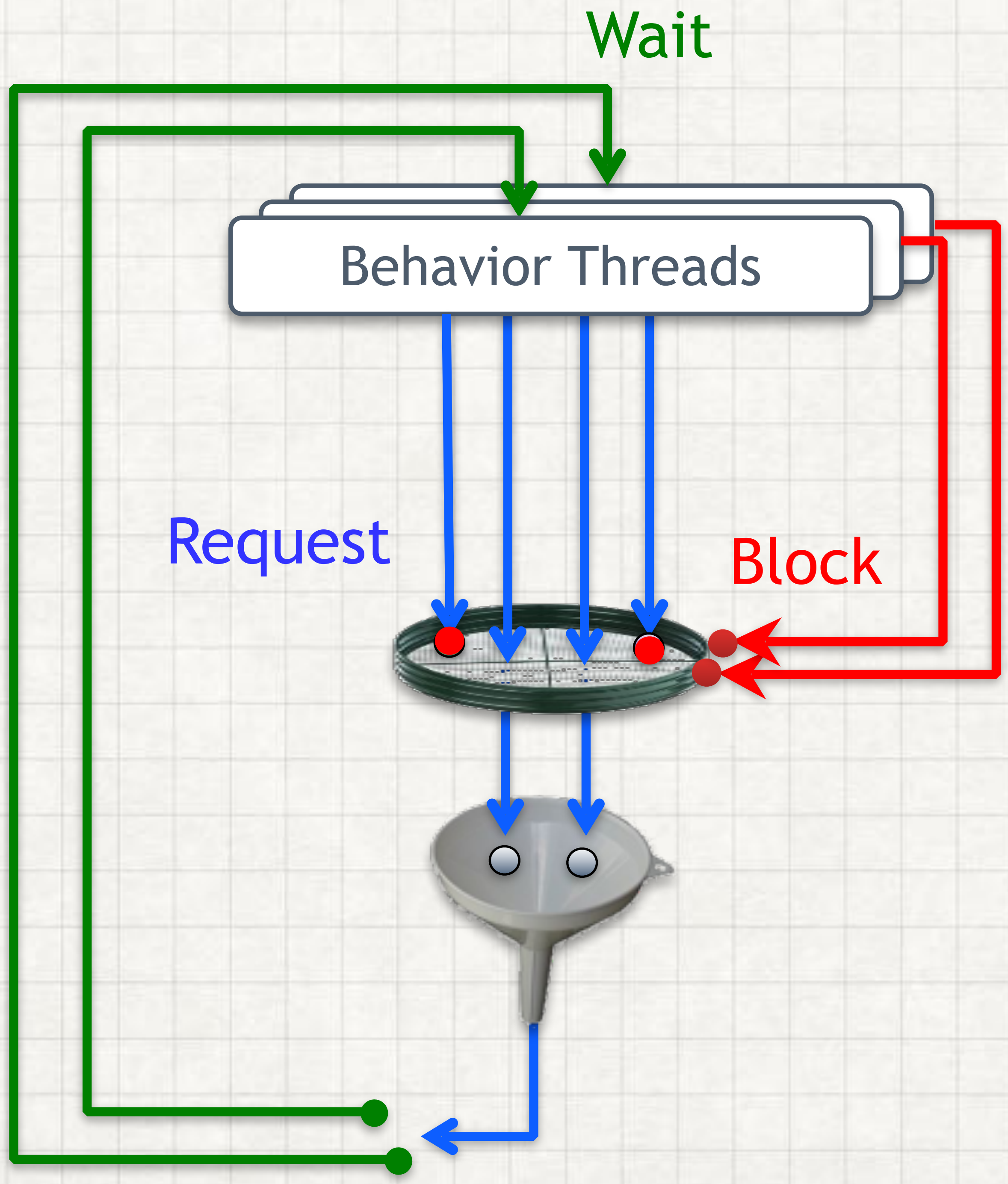
The BP Execution Cycle



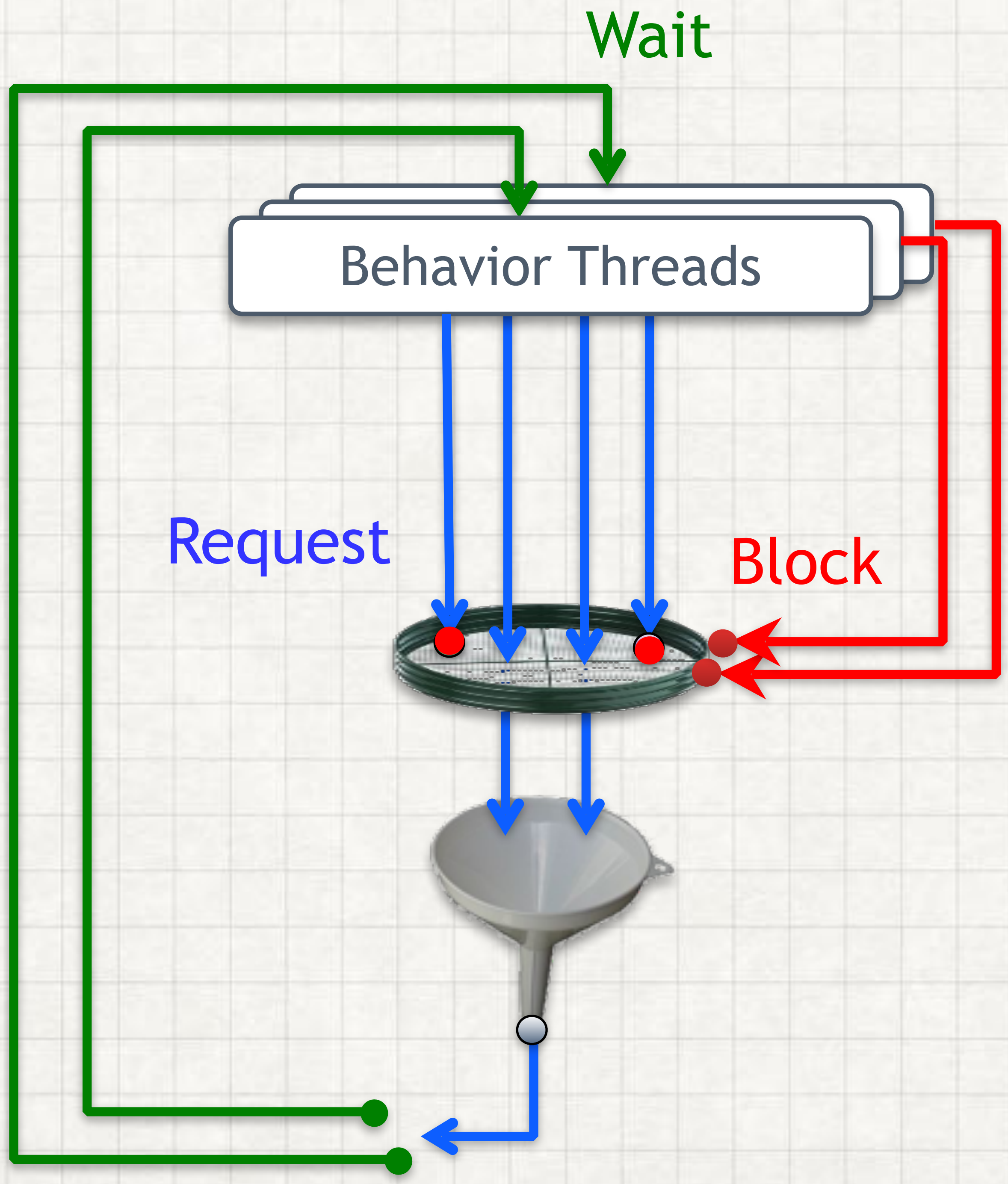
The BP Execution Cycle



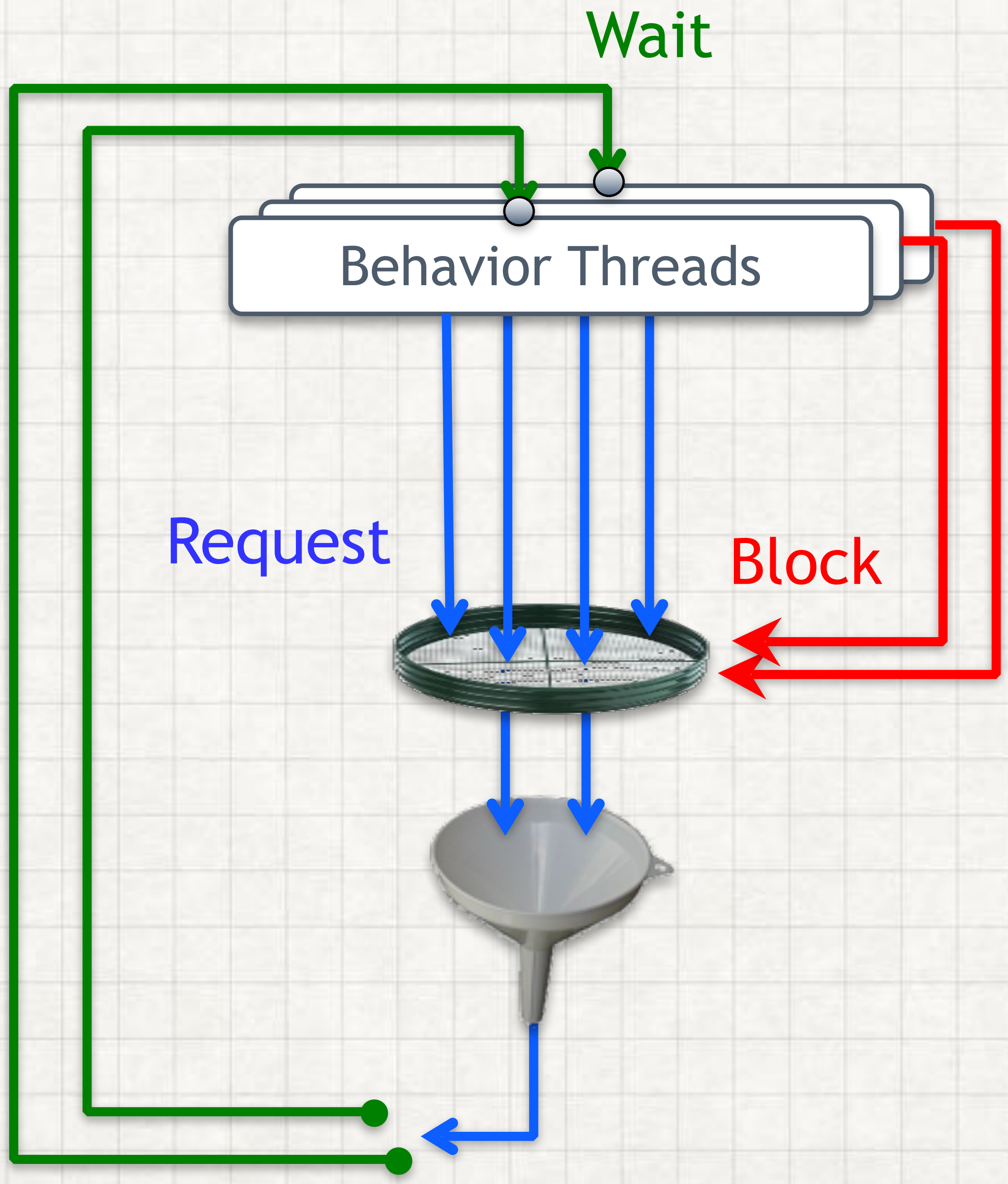
The BP Execution Cycle



The BP Execution Cycle

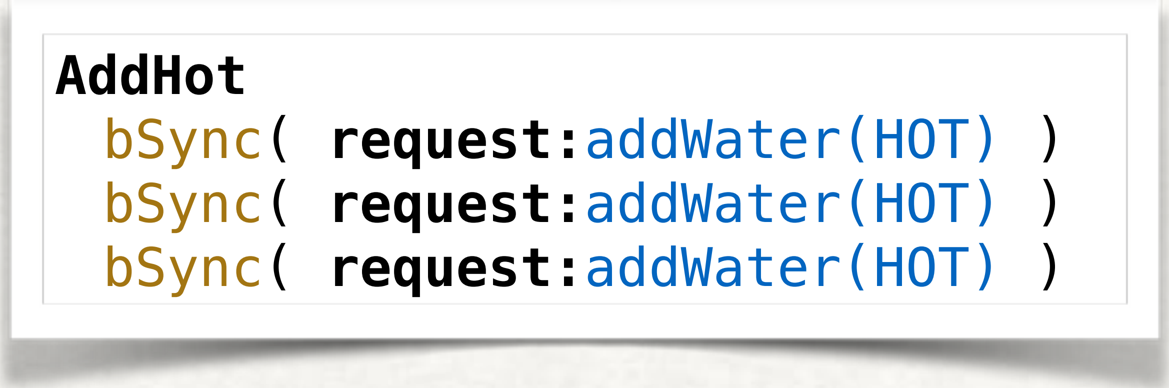


The BP Execution Cycle

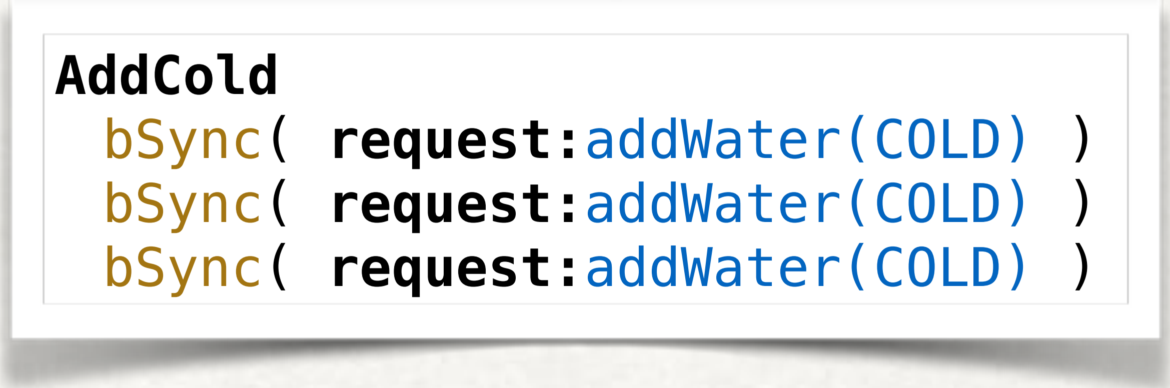


BP in action

Example: Fill a Bath



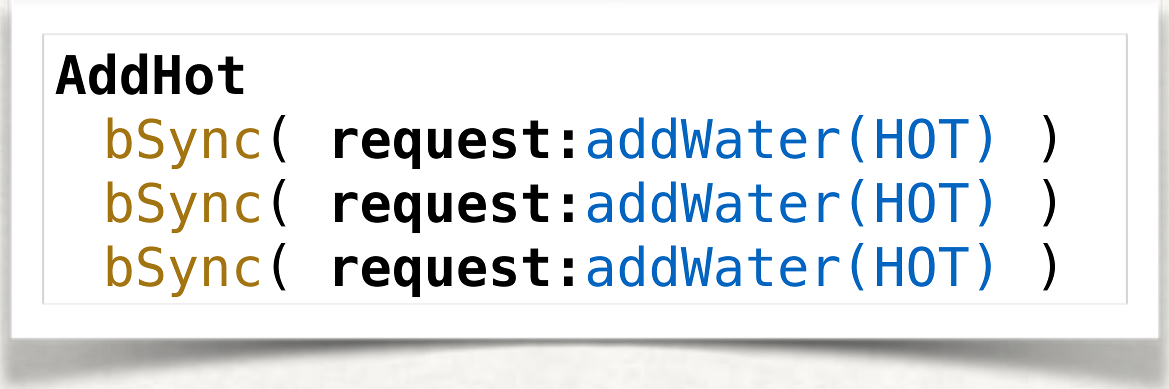
```
AddHot
  bSync( request:addWater(HOT) )
  bSync( request:addWater(HOT) )
  bSync( request:addWater(HOT) )
```



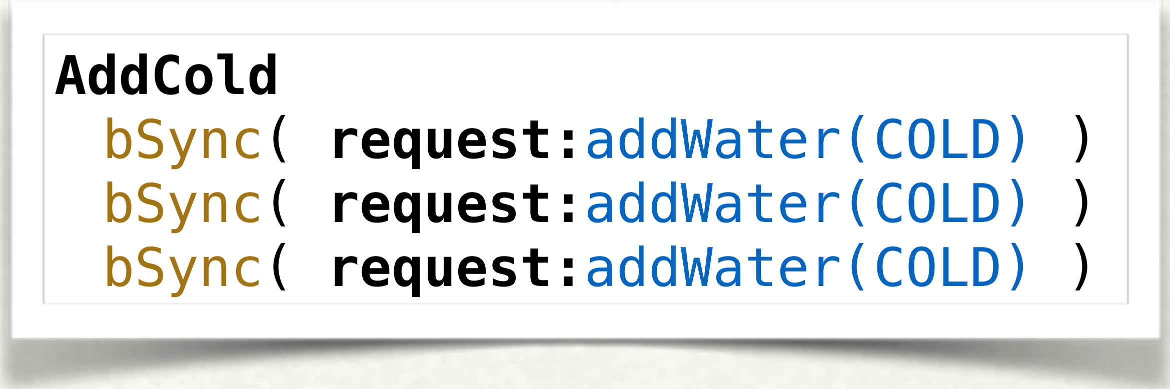
```
AddCold
  bSync( request:addWater(COLD) )
  bSync( request:addWater(COLD) )
  bSync( request:addWater(COLD) )
```

BP in action

Example: Fill a Bath



```
AddHot
  bSync( request:addWater(HOT) )
  bSync( request:addWater(HOT) )
  bSync( request:addWater(HOT) )
```



```
AddCold
  bSync( request:addWater(COLD) )
  bSync( request:addWater(COLD) )
  bSync( request:addWater(COLD) )
```

End result: Bath filled with lukewarm water.

Intermediate results: May vary

BP in action

Example: Fill a Bath

```
MaintainSaneTemp
  for ( i=1..3 ) {
    bSync( waitFor:addWater(_), block:addWater(HOT) )
    bSync( waitFor:addWater(_), block:addWater(COLD) )
  }
```

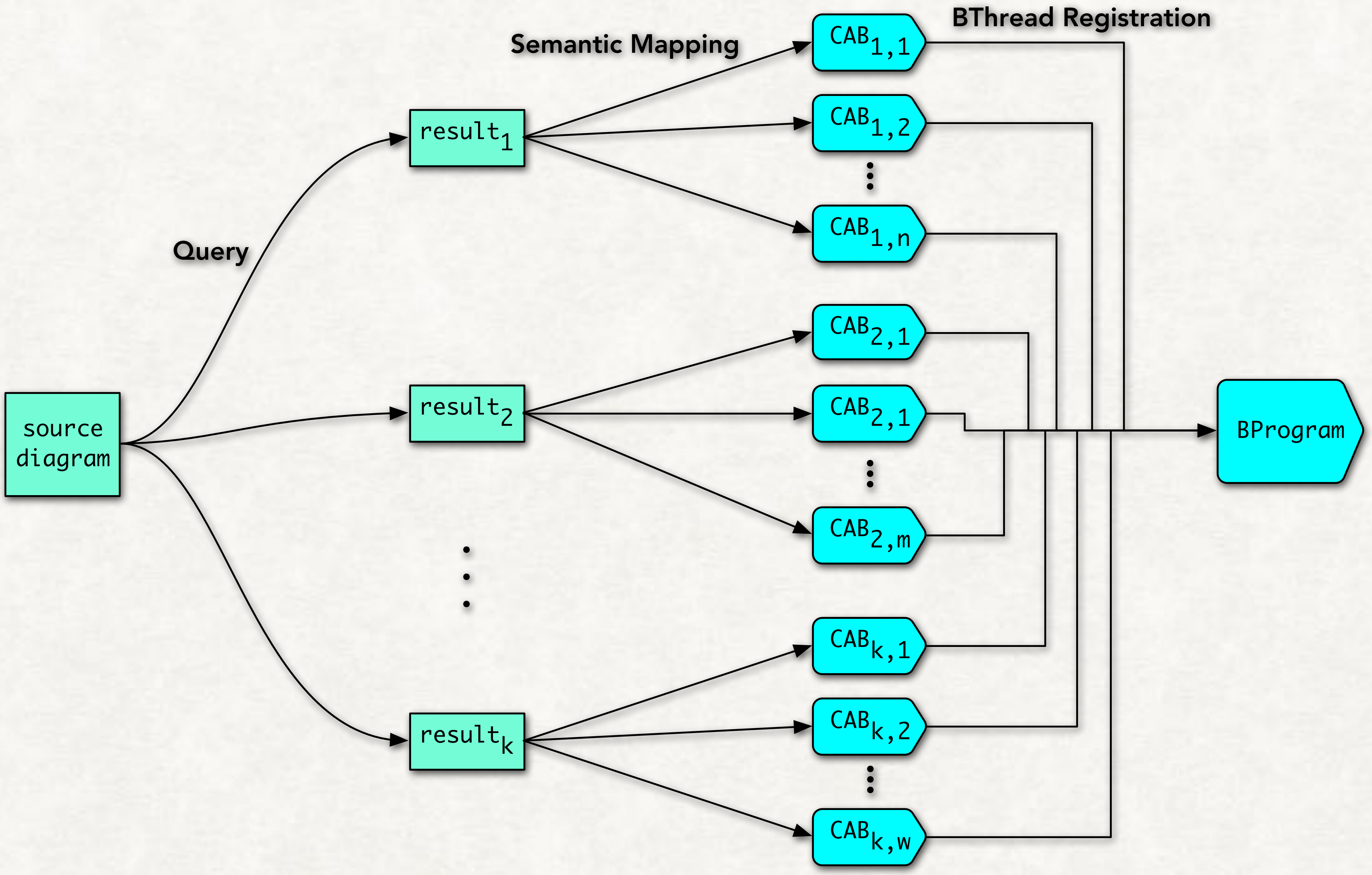
```
AddHot
  bSync( request:addWater(HOT) )
  bSync( request:addWater(HOT) )
  bSync( request:addWater(HOT) )
```

```
AddCold
  bSync( request:addWater(COLD) )
  bSync( request:addWater(COLD) )
  bSync( request:addWater(COLD) )
```

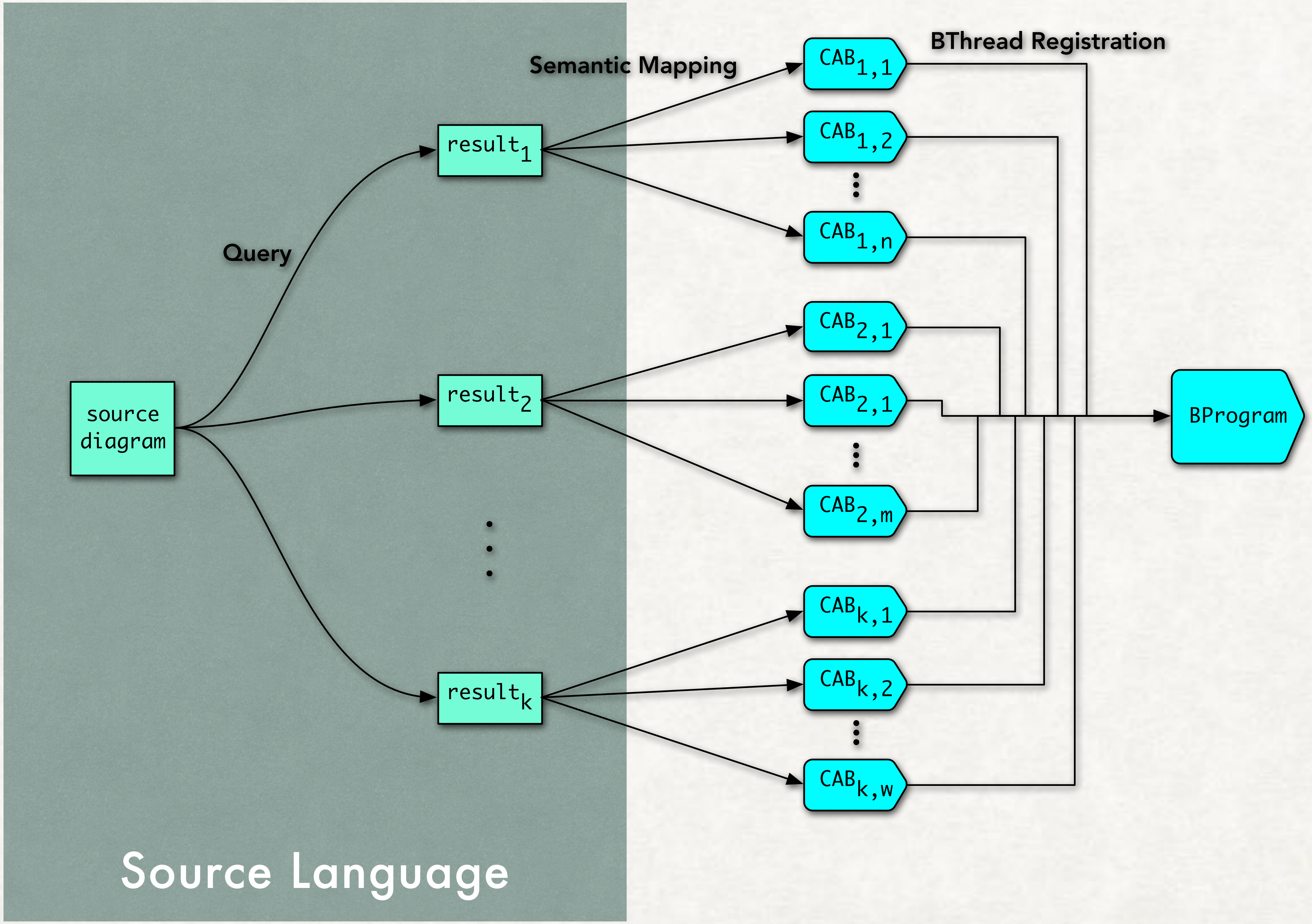
End result: Bath filled with lukewarm water.

Intermediate results: May vary

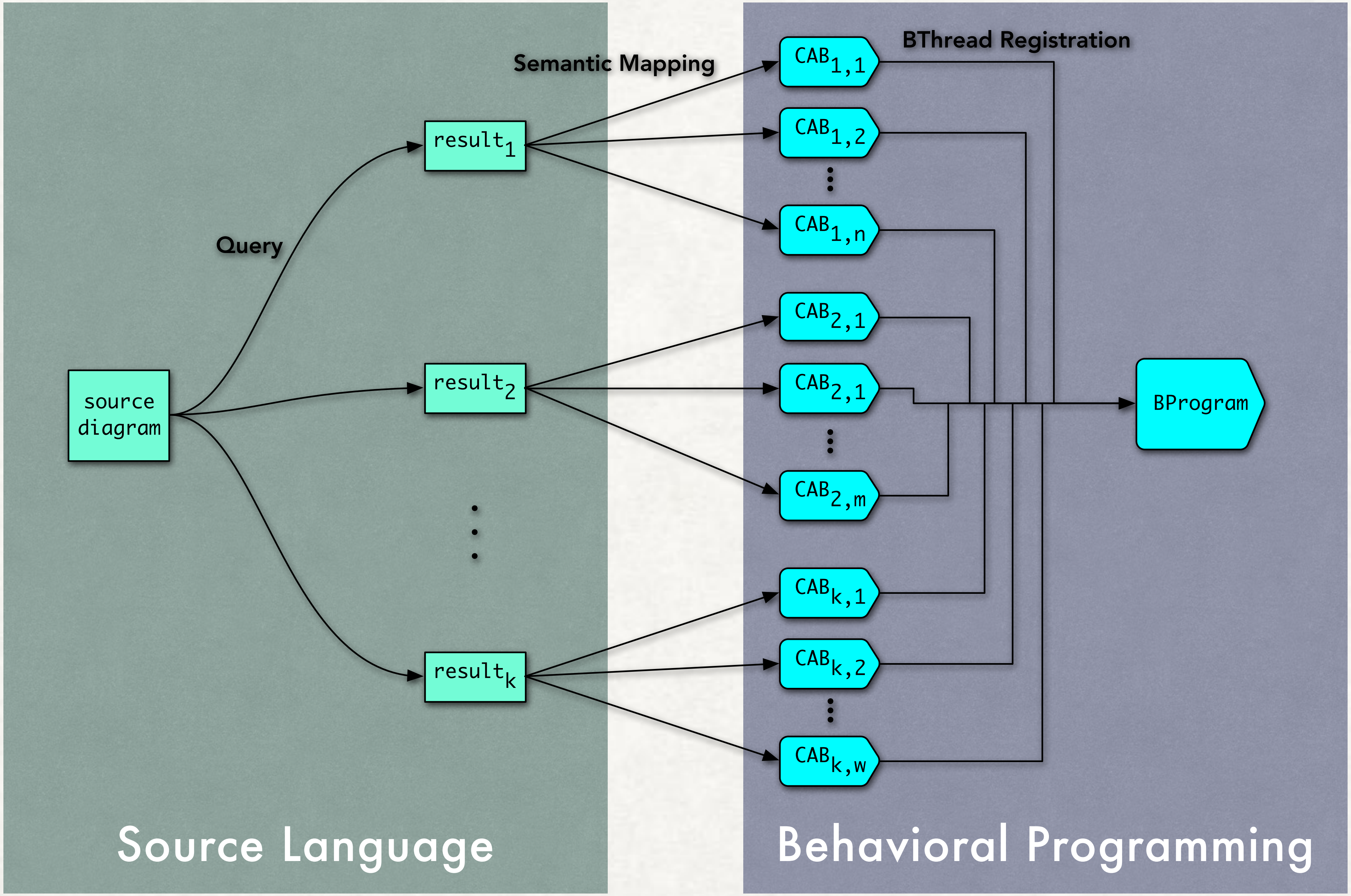
Concept Overview: Semantic Mapping



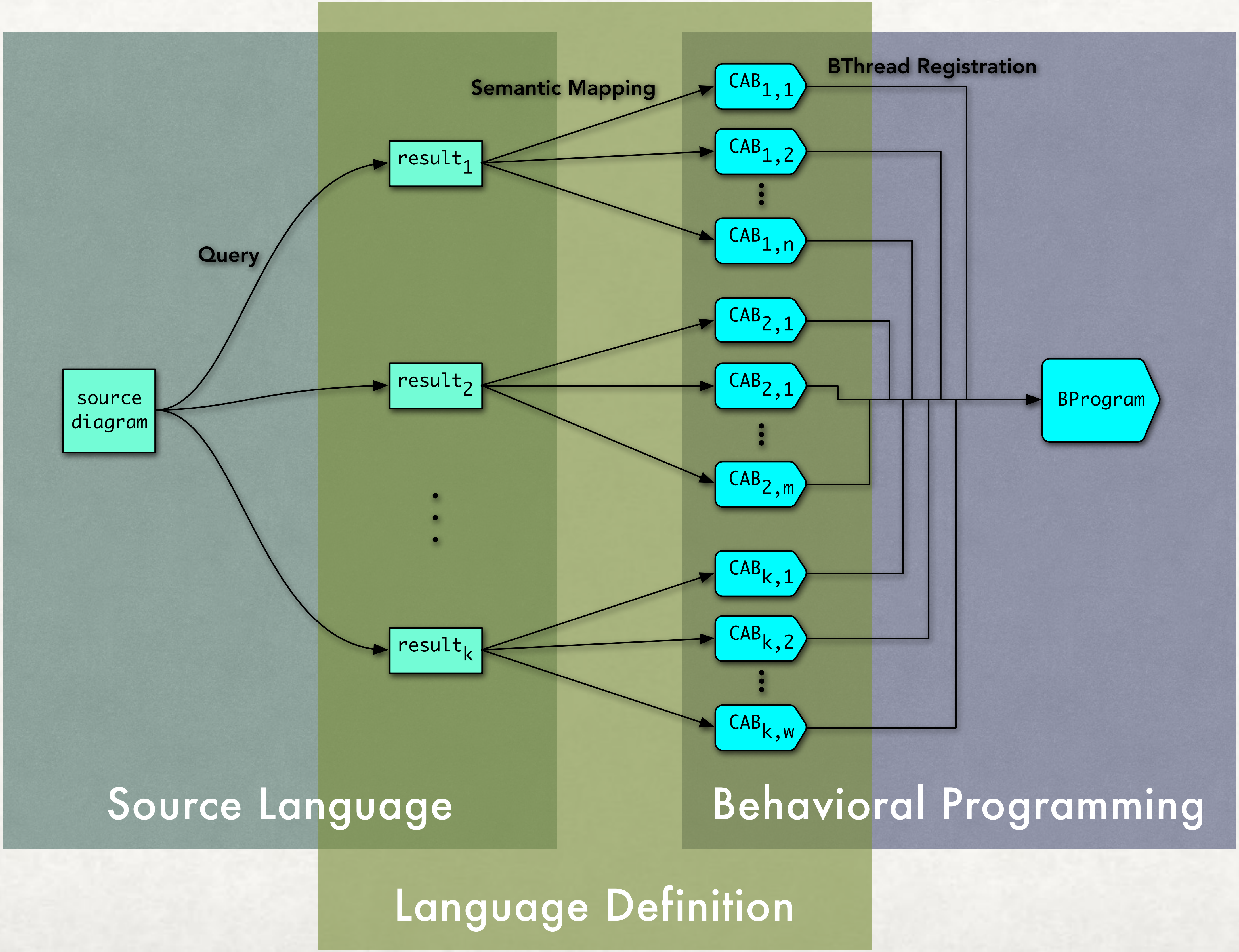
Concept Overview: Semantic Mapping



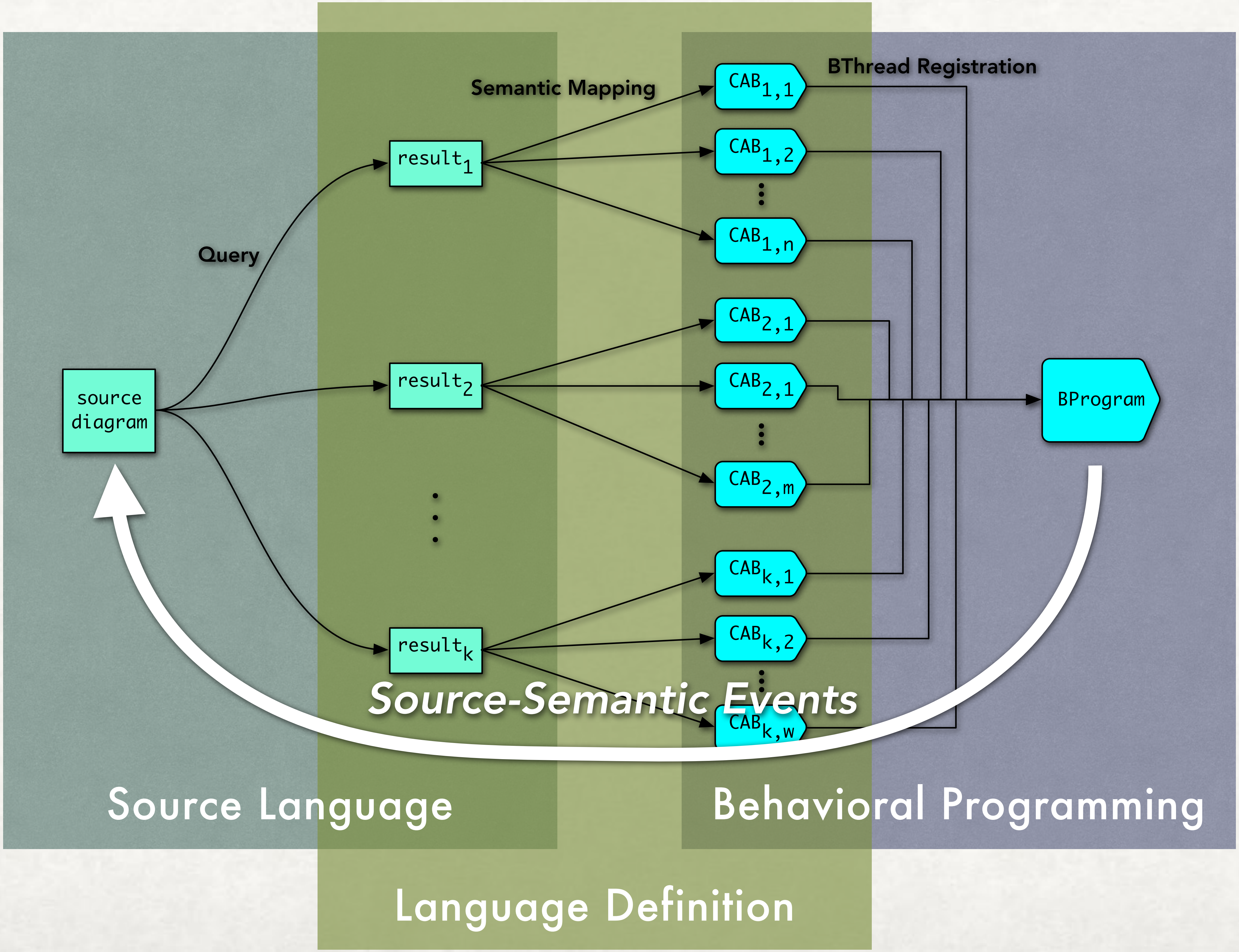
Concept Overview: Semantic Mapping



Concept Overview: Semantic Mapping



Concept Overview: Semantic Mapping



Proposed Methodology in Action

Define Language *L*:

- **Decide** what are the syntactic building blocks of programs in the language (e.g. squares and ovals, connected by green arrows or red arrows).
- **Define** a set of queries over programs in L.
- For each query, **define** a set of *BThread templates*, parametrized by query results.
    - Each BThread templates defines some of the executable semantics of its parameters. Together, the templates define the full executable semantic of the construct returned from the query.

Write Programs *P* in *L*:

- **Draw** the program's diagrams
- **Load** L's language definition into an execution tool.
- **Load** the diagrams into the tool.
- **Press** the *Run* button.
- The tool runs the queries in L's definition against *P*'s diagrams. It then uses the BThread templates and the query results to generate a BP-program, and executes it.

So, Is this a Good Idea?

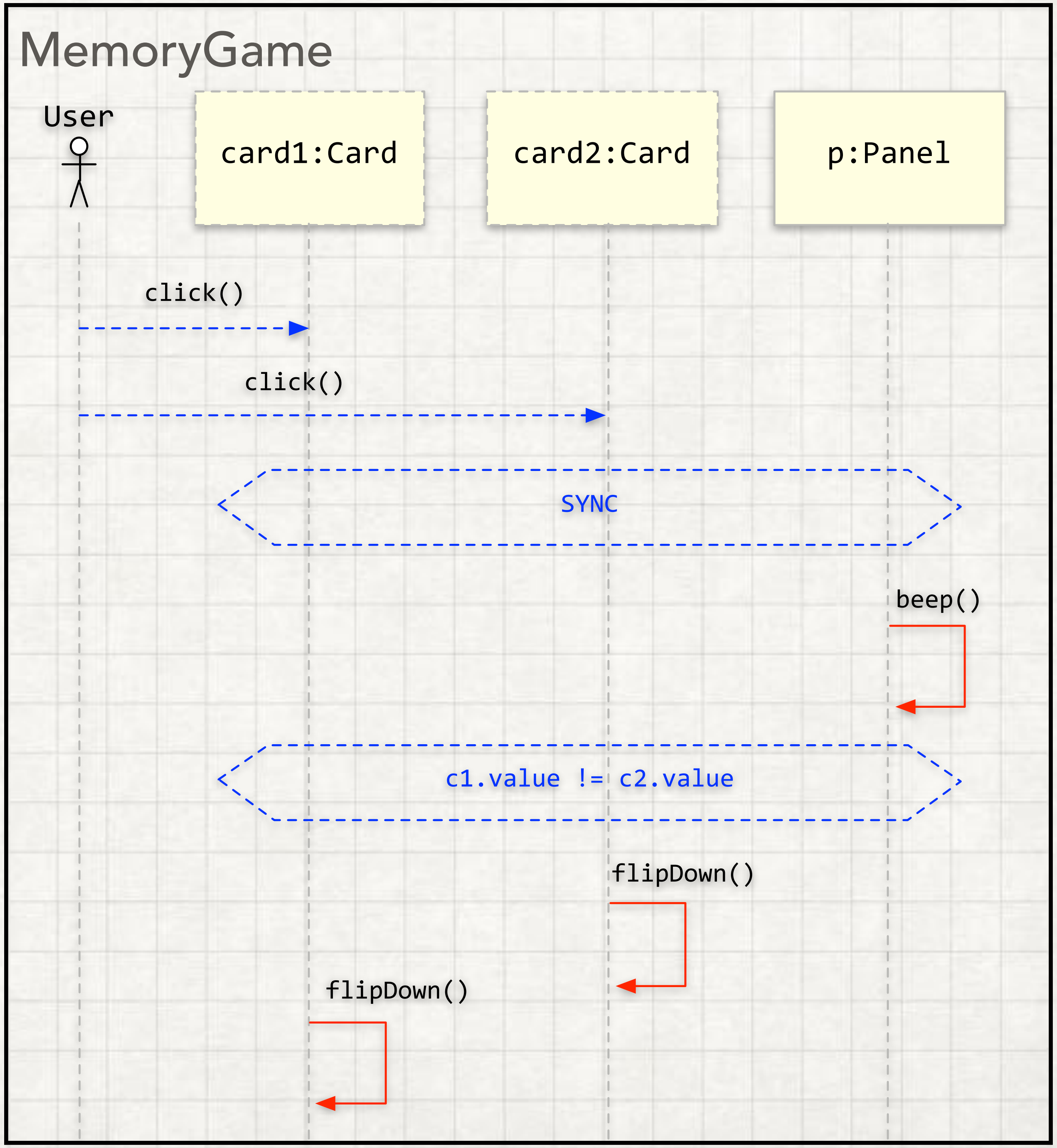
- Reader needs to remember multiple CABs for understanding a single construct
- BP not widely known
- Too many BThreads might cause performance issues

- Generated definitions are:
  - Formal
  - Accessible
  - Executable
  - Verifiable (e.g. BPMC)
- Mix and match semantics and constructs by adding/removing relevant queries and mappers only.
- Use BP as common execution environment for heterogeneous specifications

Case Study:
Semantic Variations of
Live Sequence Charts

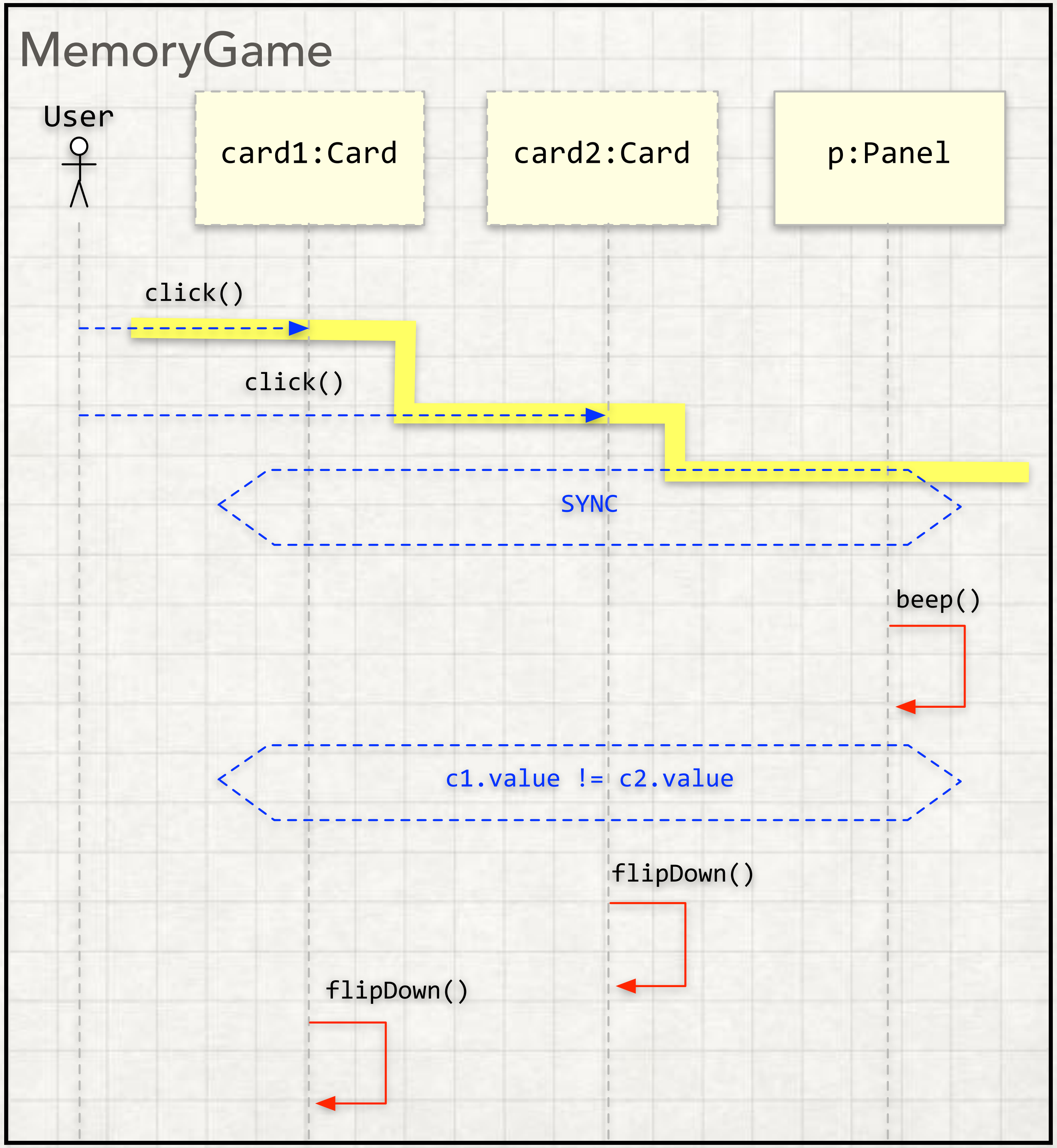
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



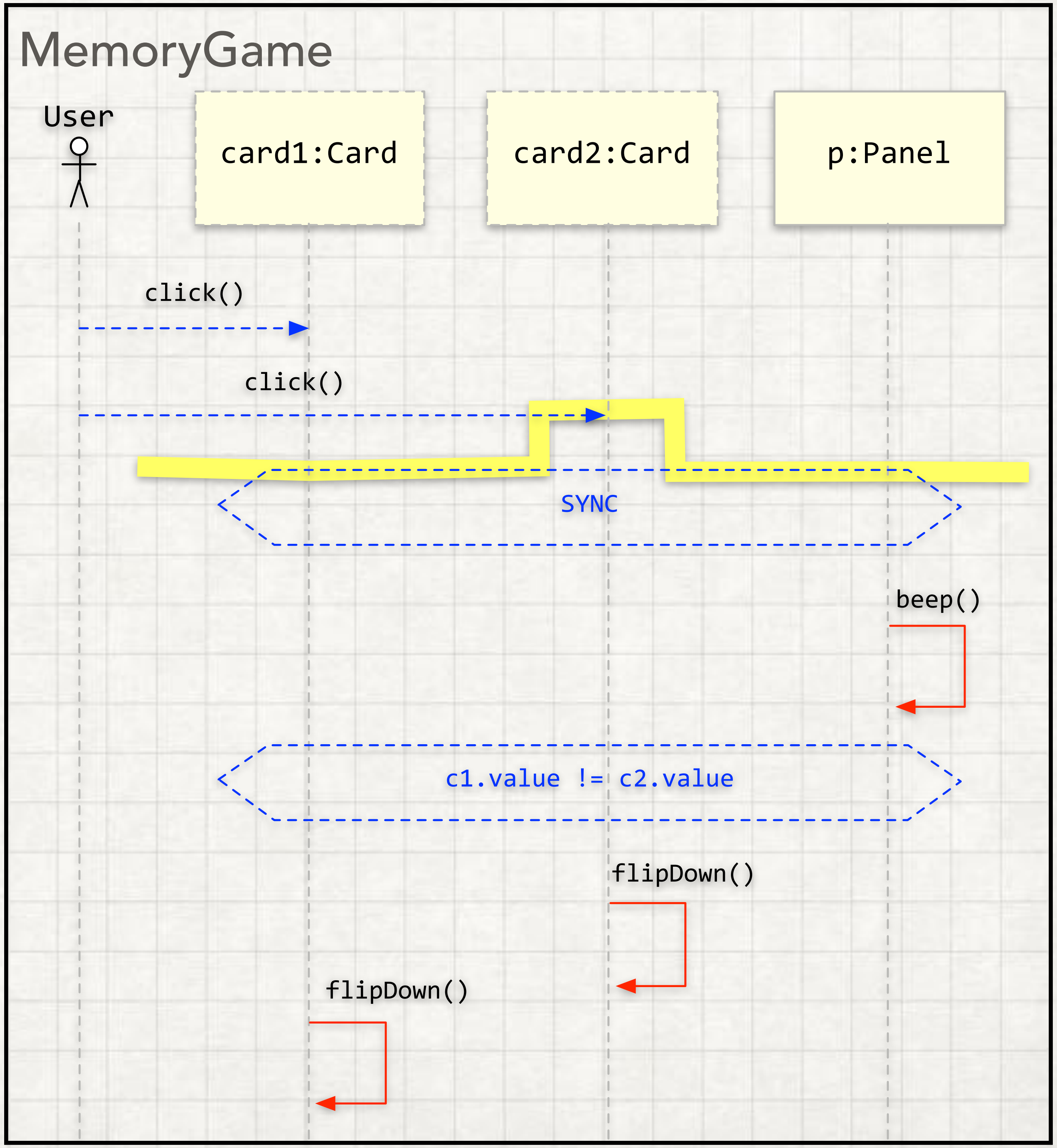
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



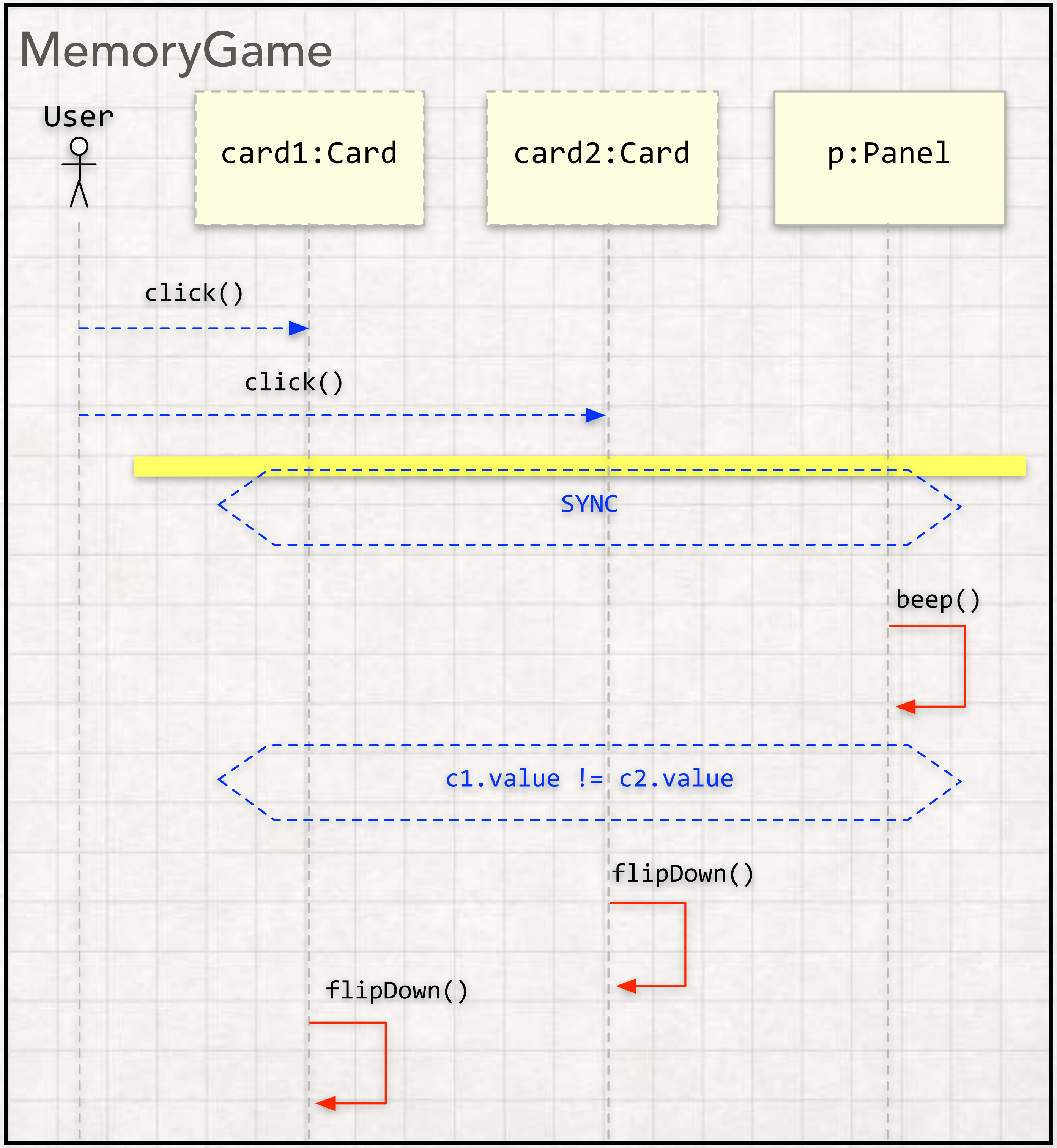
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



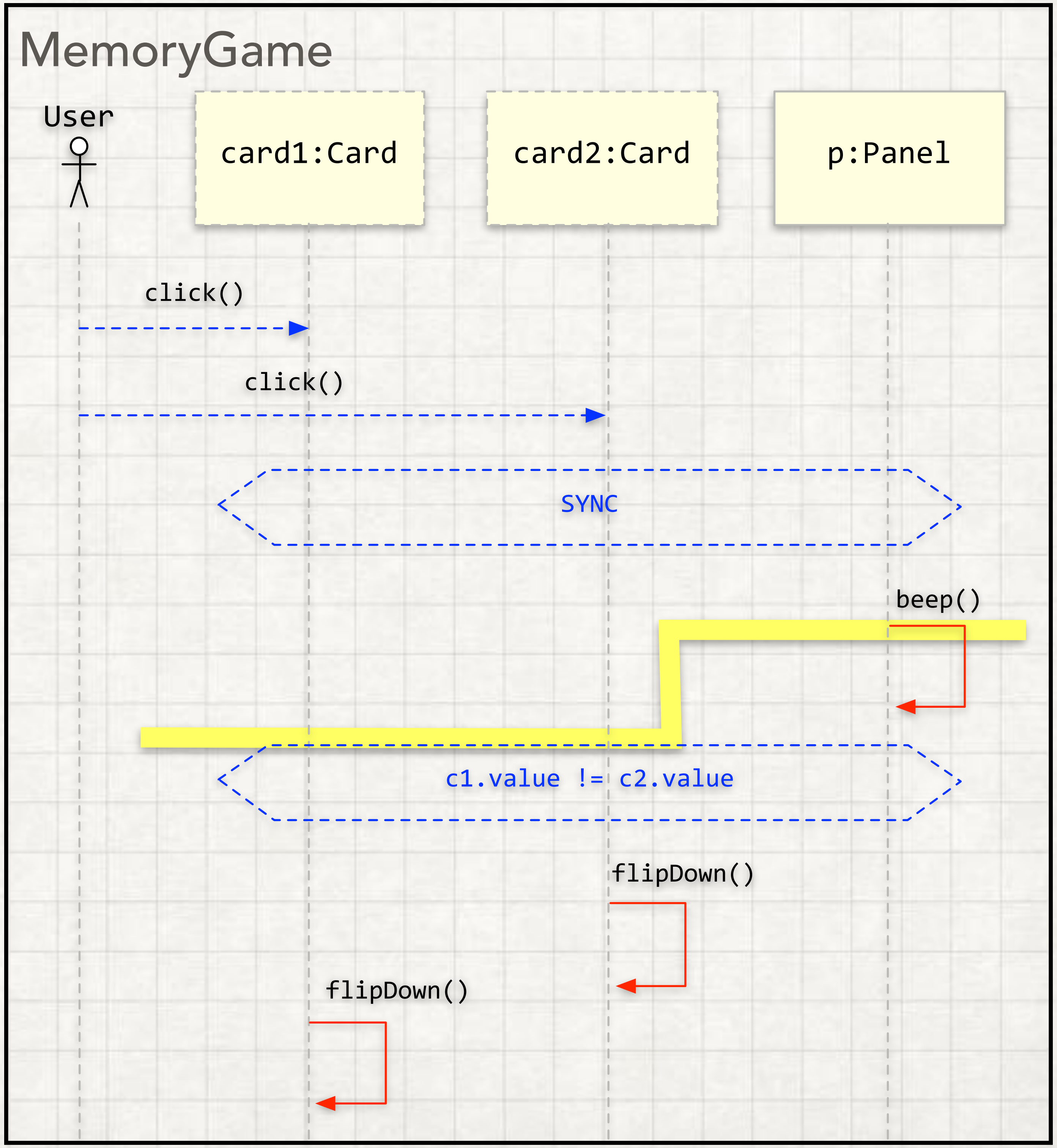
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



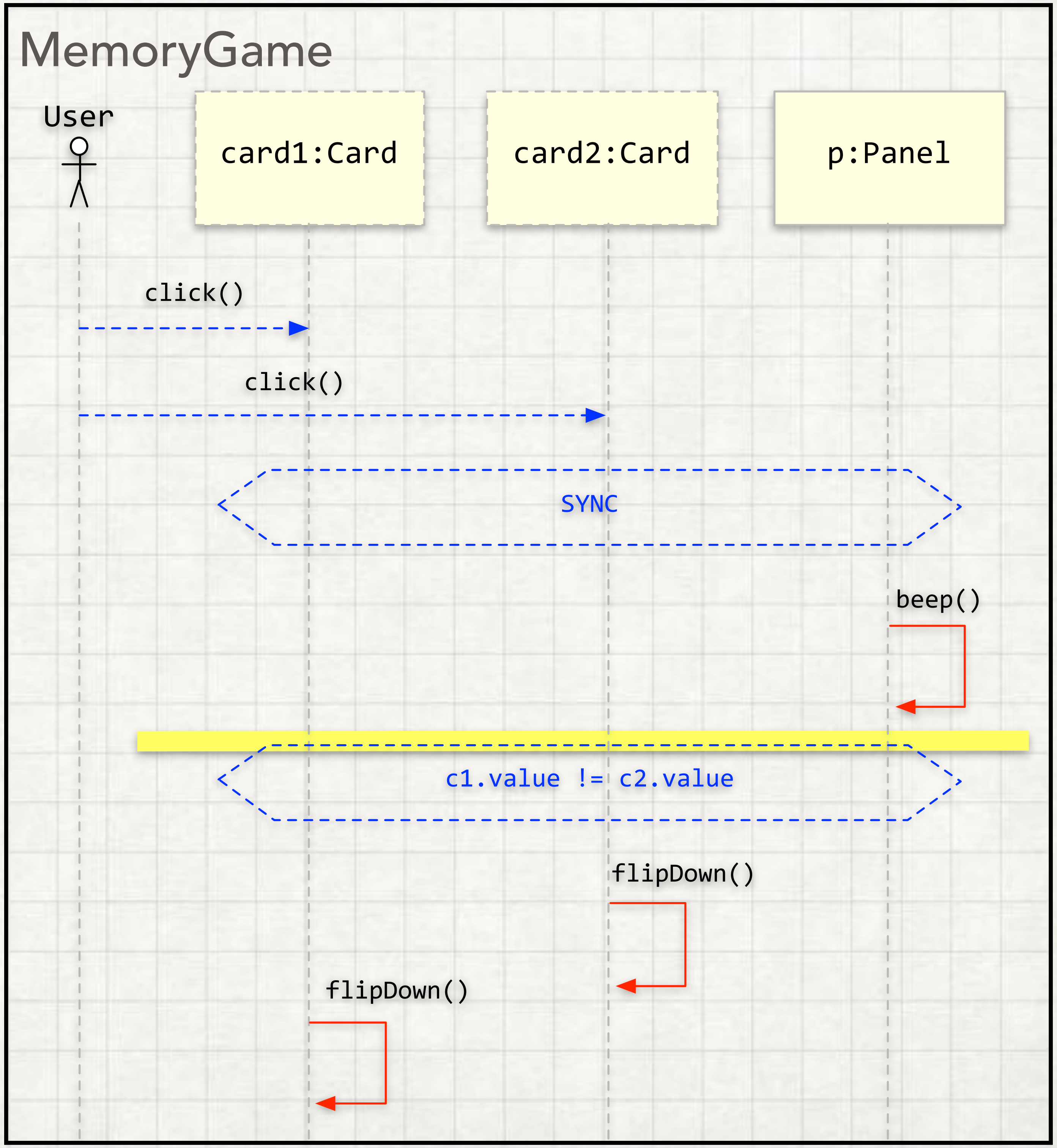
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
    - Original (PlayGo)
    - UML2 Compliant



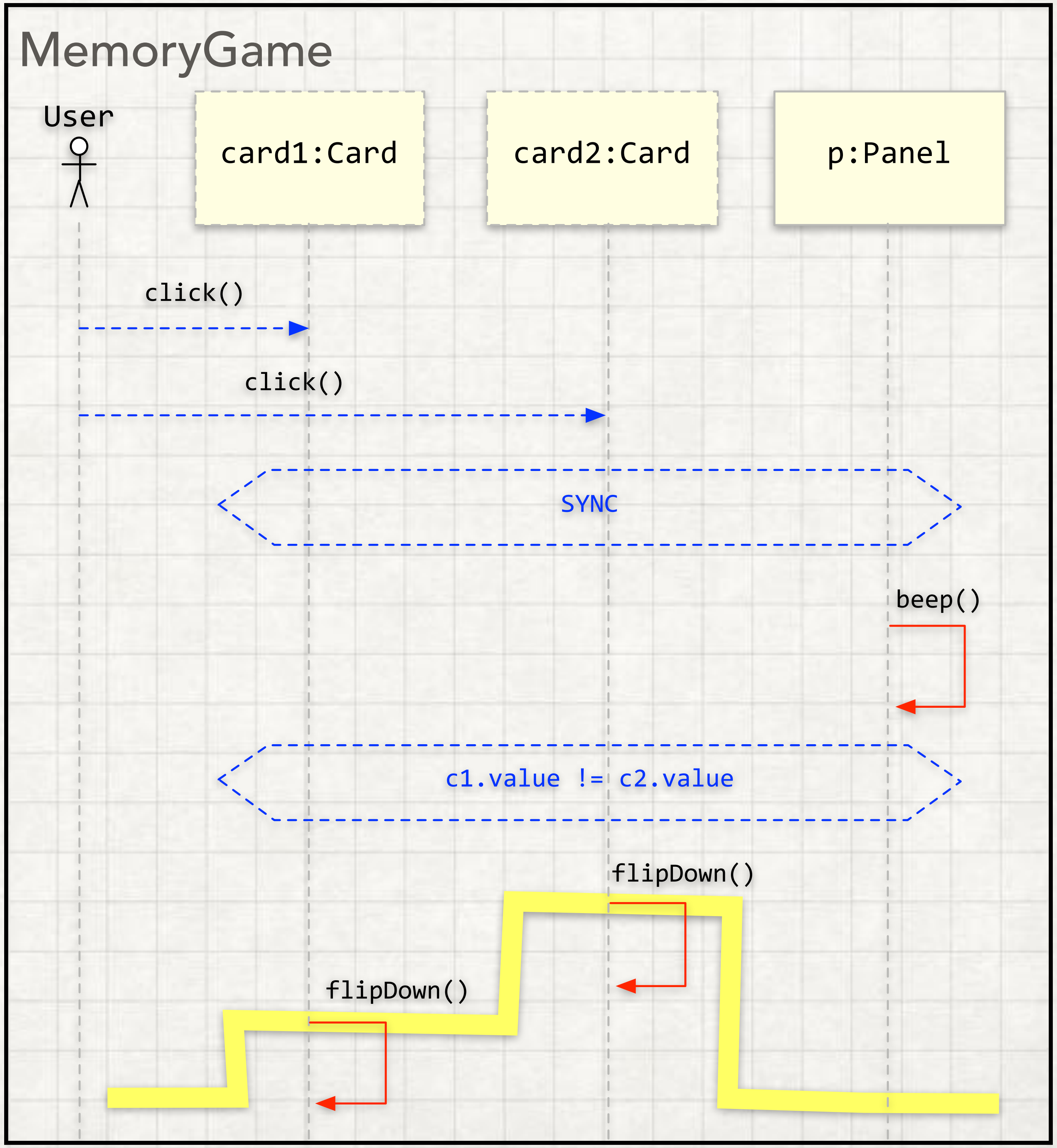
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



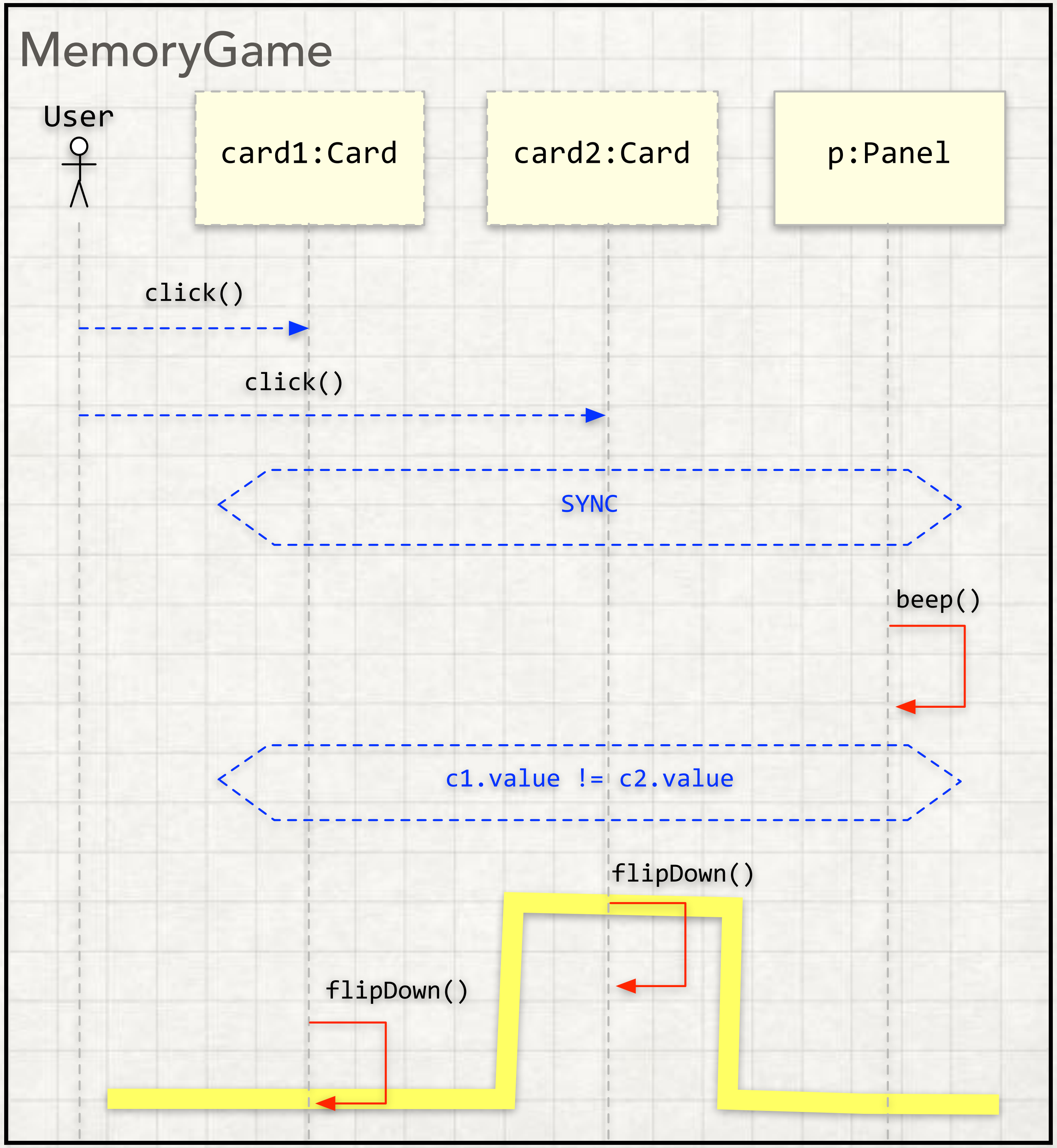
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



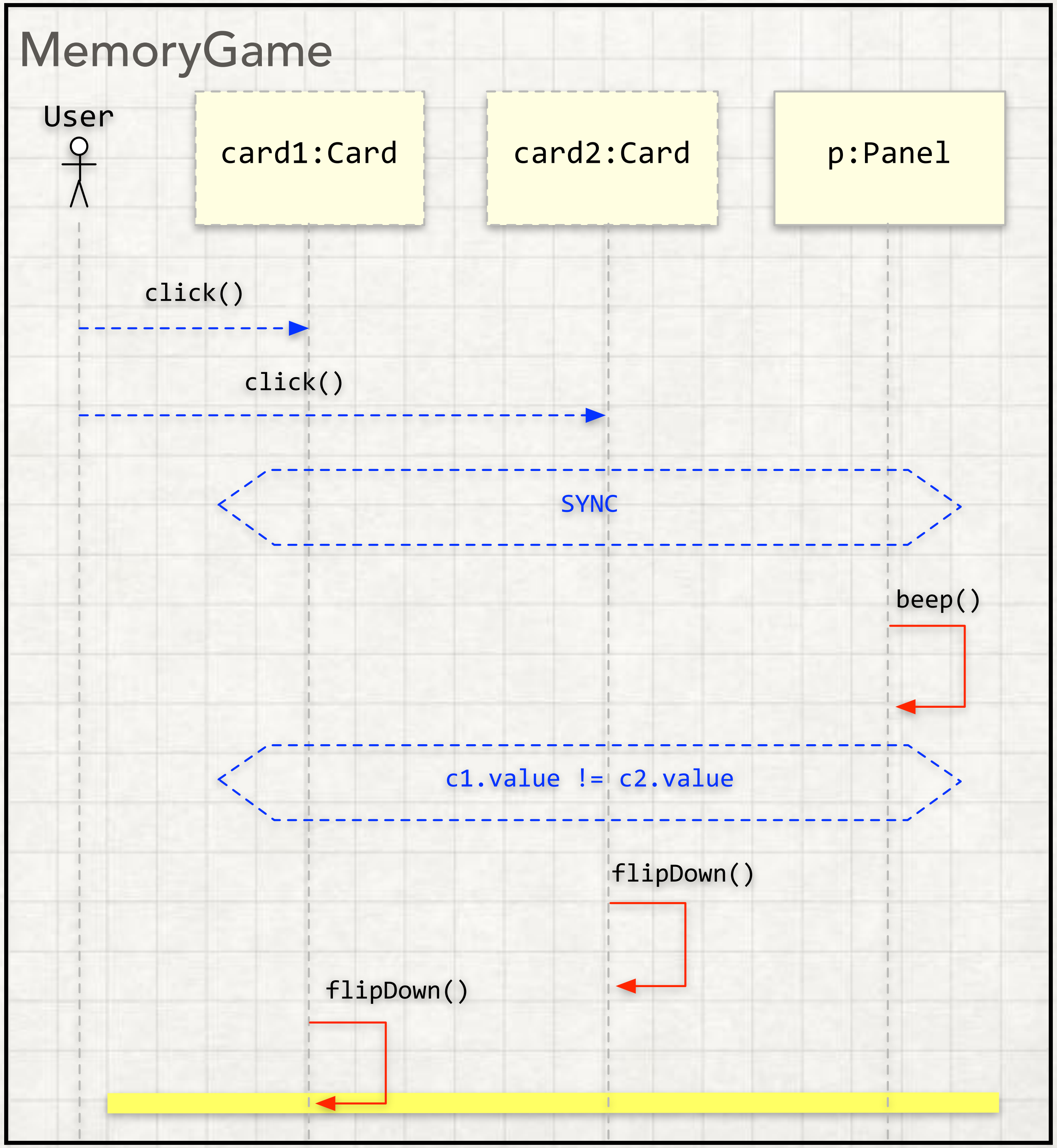
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



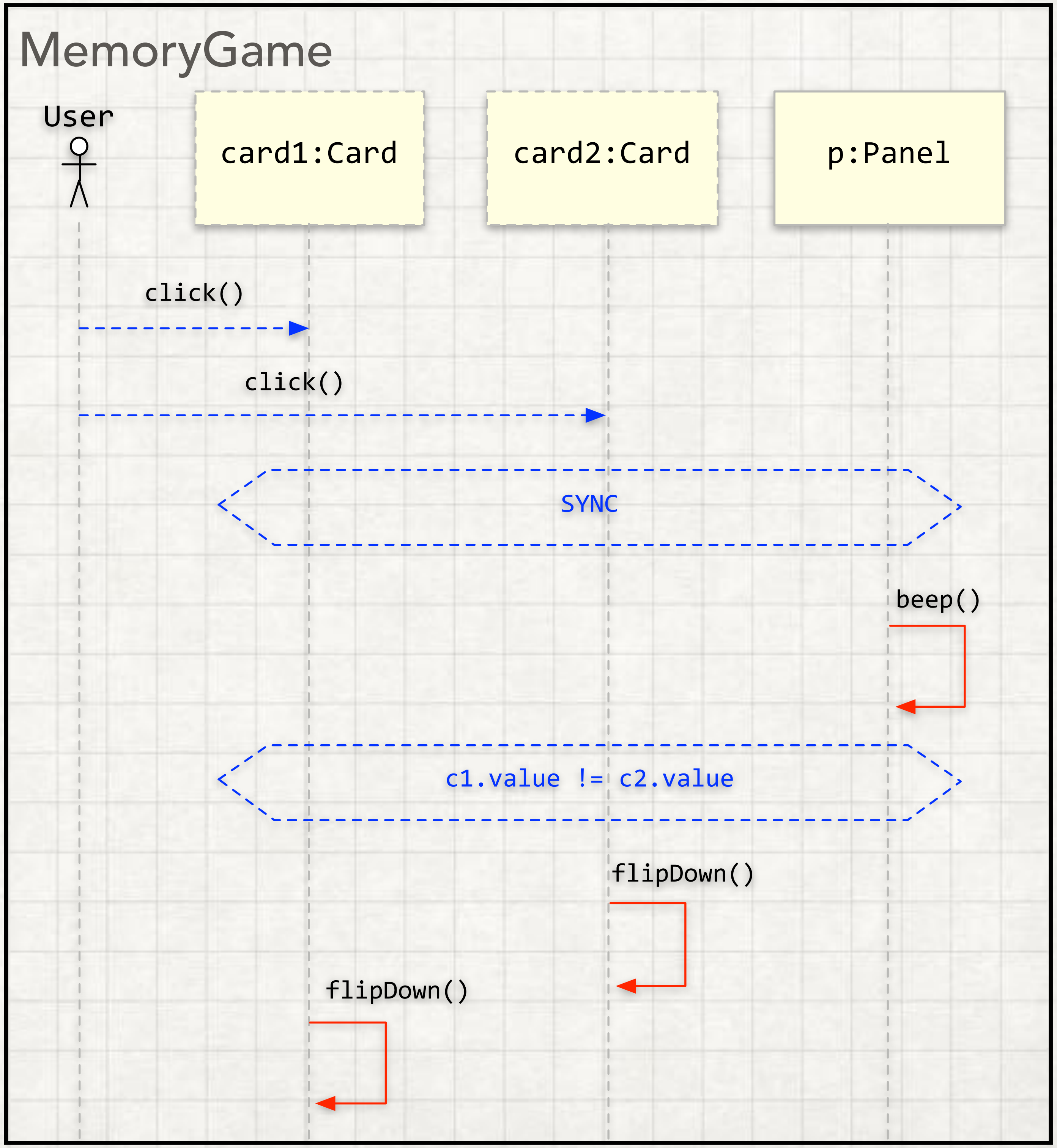
Live Sequence Charts (LSC) - a Quick Intro

- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant

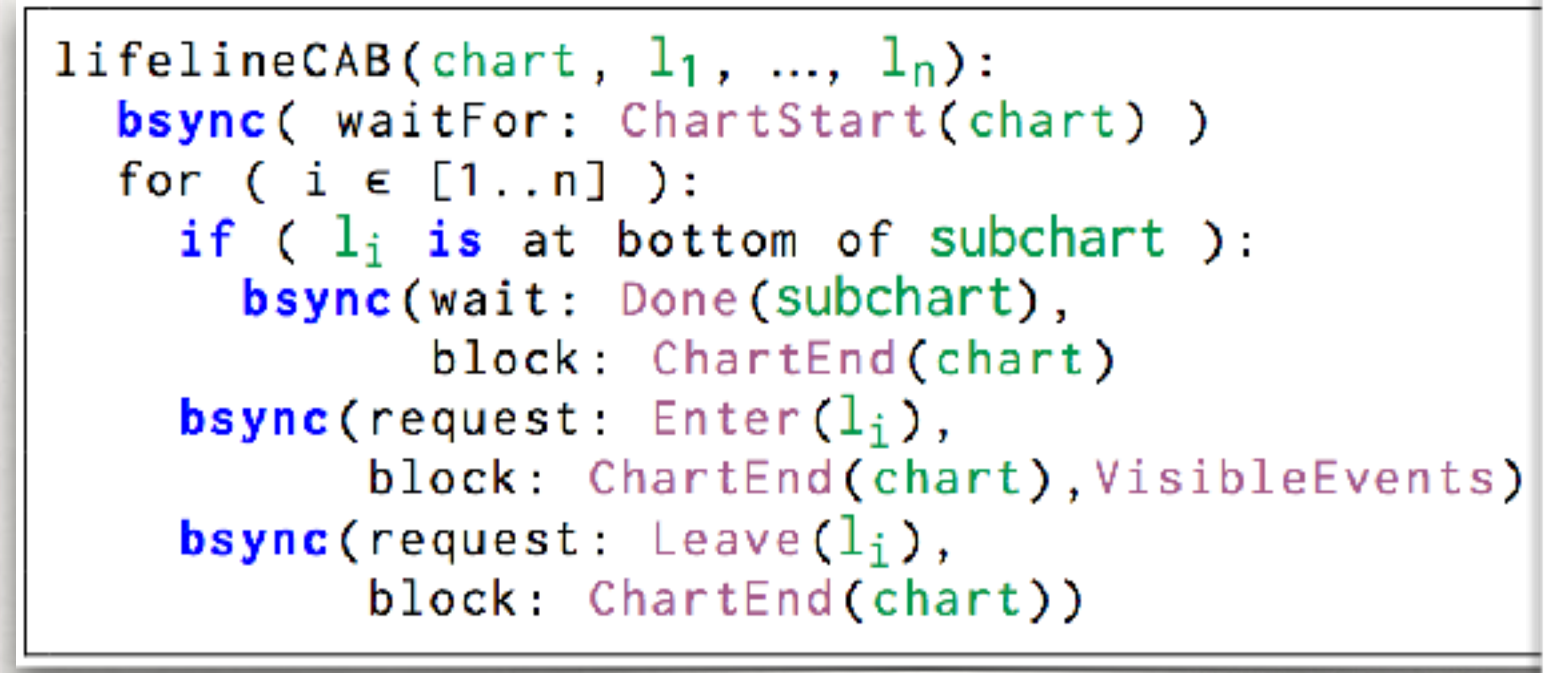


Live Sequence Charts (LSC) - a Quick Intro

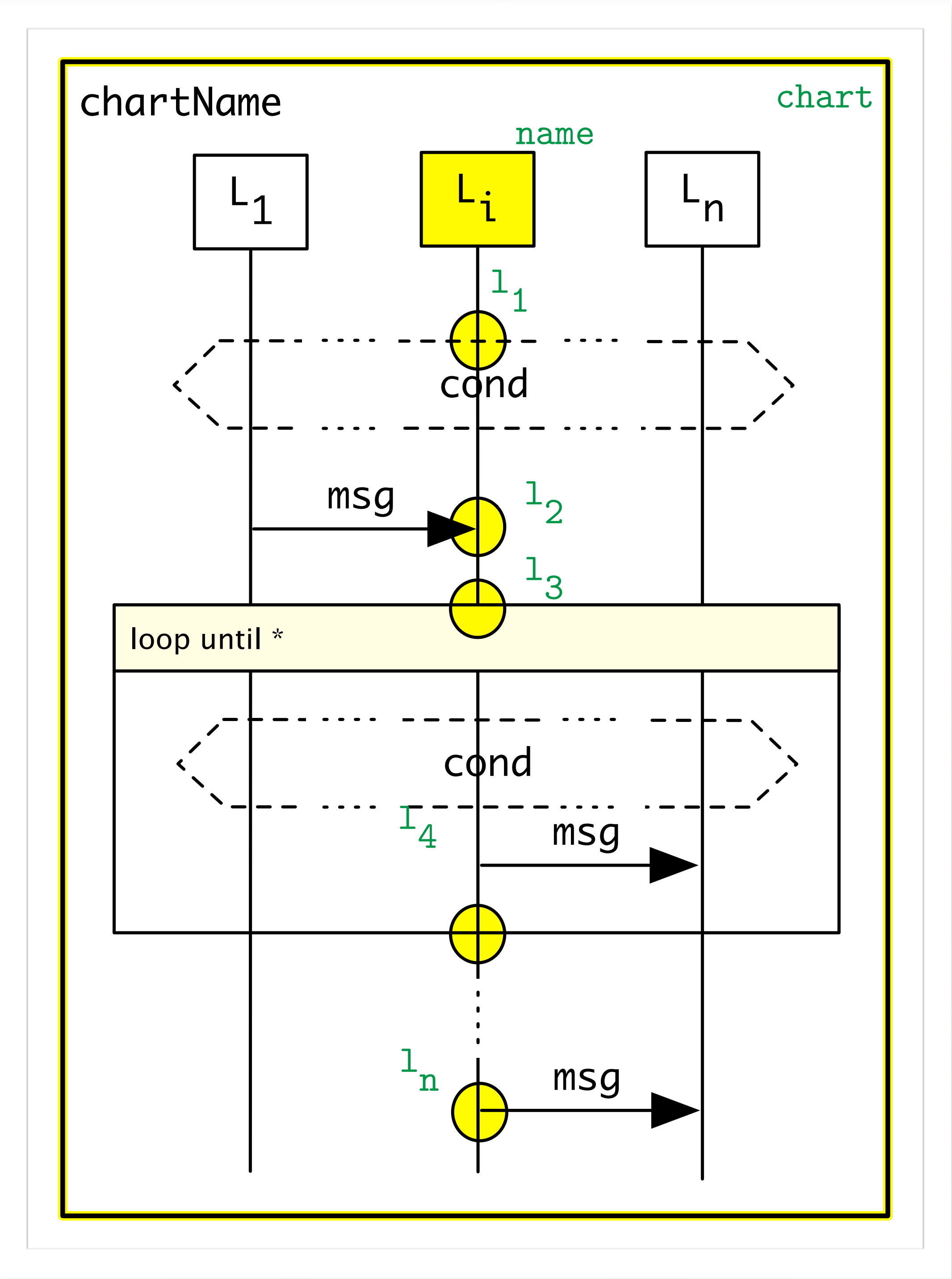
- Damm and Harel, 2001
- Diagrammatic programming language, extending classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities
- Rich set of features: concurrency, event unification, forbidden scenarios and more.
- Multiple Semantic Variations:
  - Original (PlayGo)
  - UML2 Compliant



Lifeline

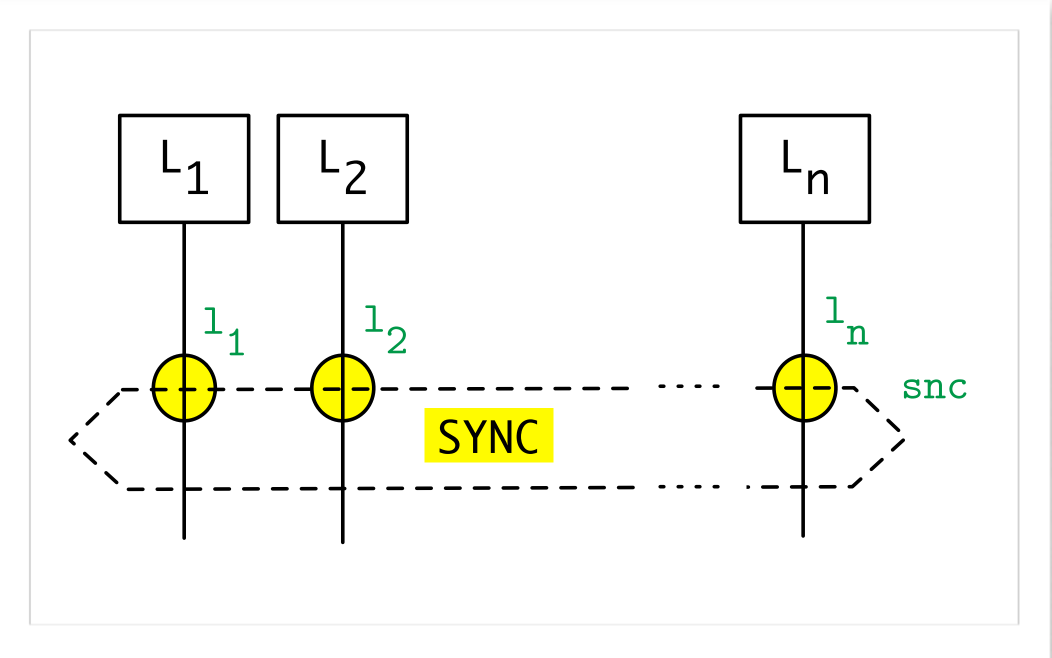


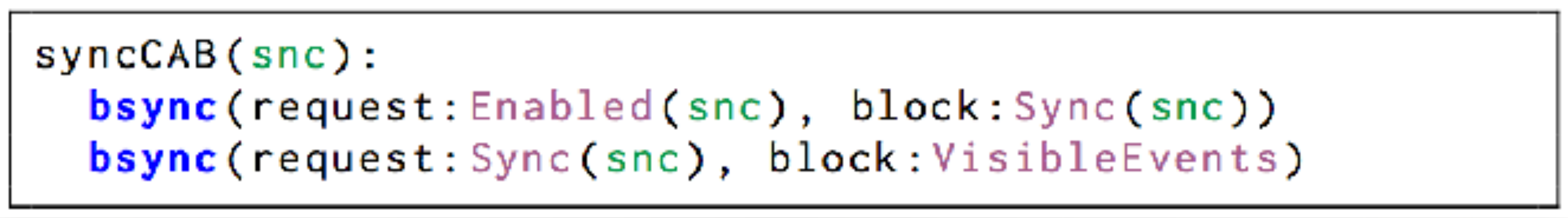
```
lifelineCAB(chart, l_1, …, l_n):
  bsync( waitFor: ChartStart(chart) )
  for ( i ∊ [1..n] ):
    if ( l_i is at bottom of subchart ):
      bsync(wait: Done(subchart),
            block: ChartEnd(chart)
    bsync(request: Enter(l_i),
          block: ChartEnd(chart),VisibleEvents)
    bsync(request: Leave(l_i),
          block: ChartEnd(chart))
```



Sync

- For each i in 1..n:
  - blockUntilCAB(Enabled(**snc**), Enter(**l_i**))
- For each i in 1..n:
  - blockUntilCAB(Leave(**l_i**), Sync(**snc**))

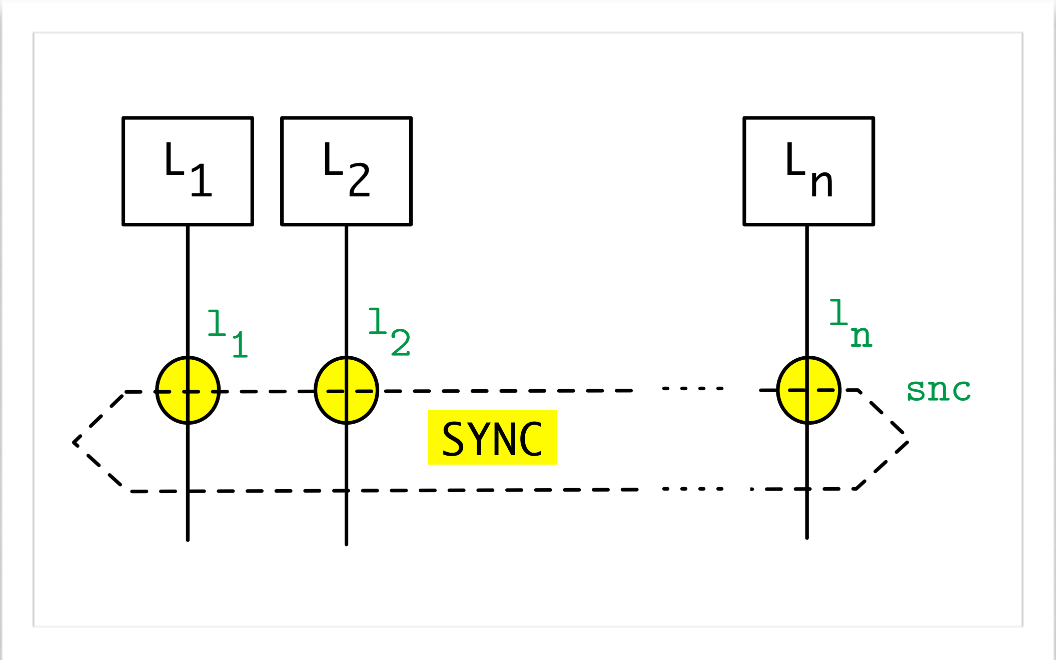


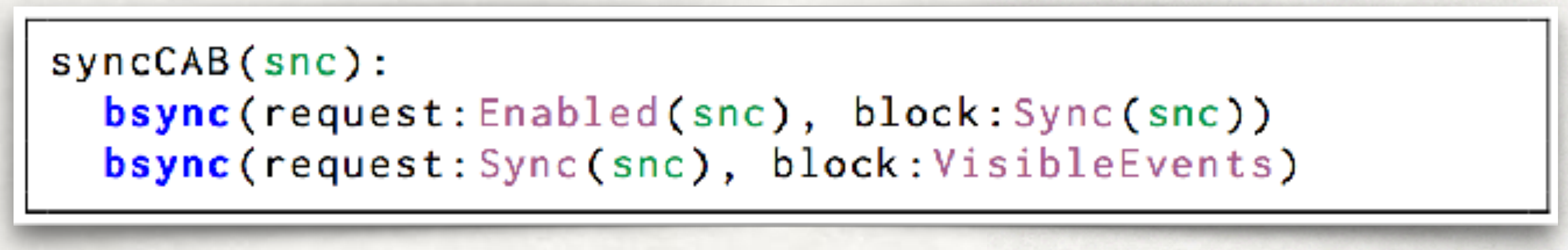


```
syncCAB(snc):
  bsync(request:Enabled(snc), block:Sync(snc))
  bsync(request:Sync(snc), block:VisibleEvents)
```

Sync

- For each i in 1..n:
  - blockUntilCAB(Enabled(**snc**),
    Enter($\mathbf{l_i}$))
- For each i in 1..n:
  - blockUntilCAB(Leave($\mathbf{l_i}$),
    Sync(**snc**))





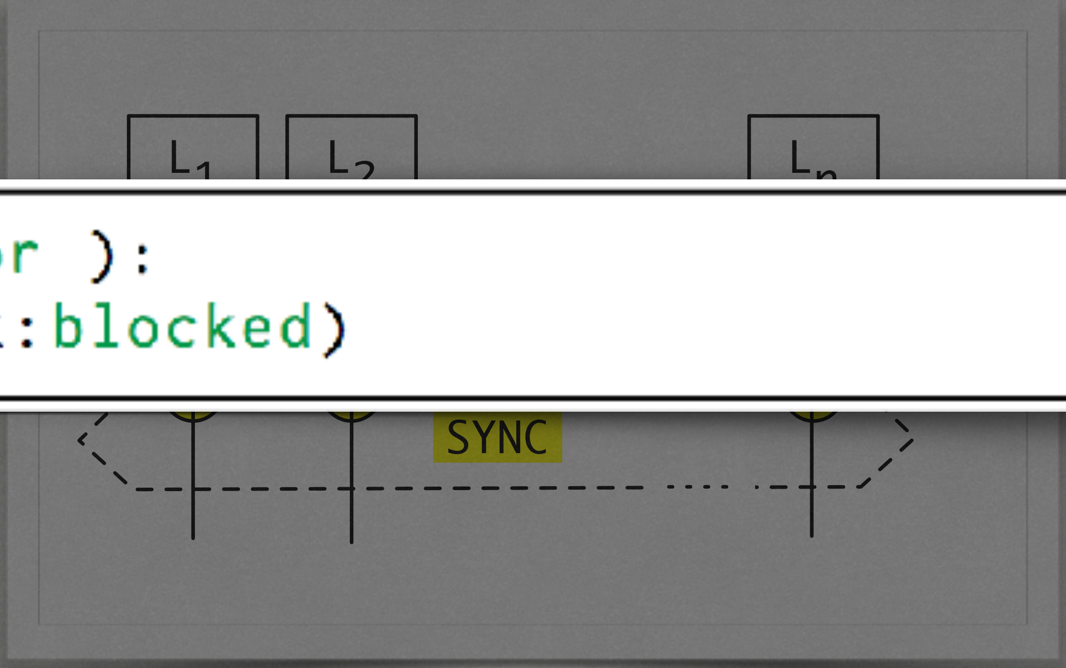
```
syncCAB(snc):
  bsync(request:Enabled(snc), block:Sync(snc))
  bsync(request:Sync(snc), block:VisibleEvents)
```

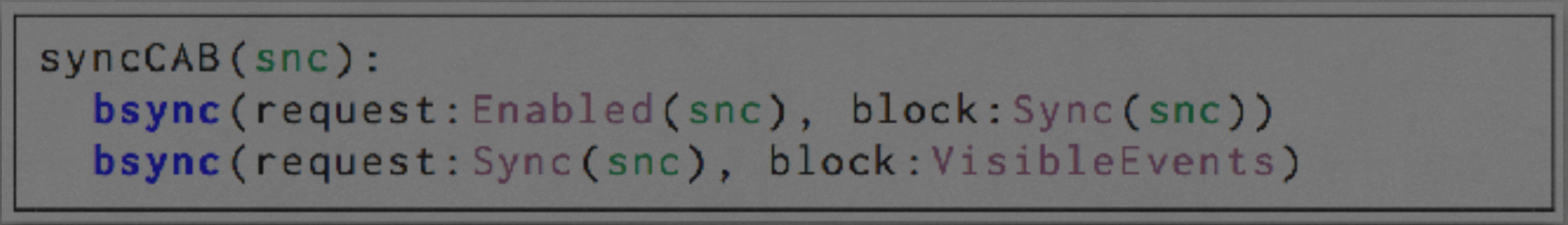
Sync

- For each i in 1..n:
  - blockUntilCAB(Enabled(snc),
    Enter(l_i))

```
blockUntilCAB( blocked, waitedFor ):
  bsync(waitFor:waitedFor, block:blocked)
```

- Sync(snc)





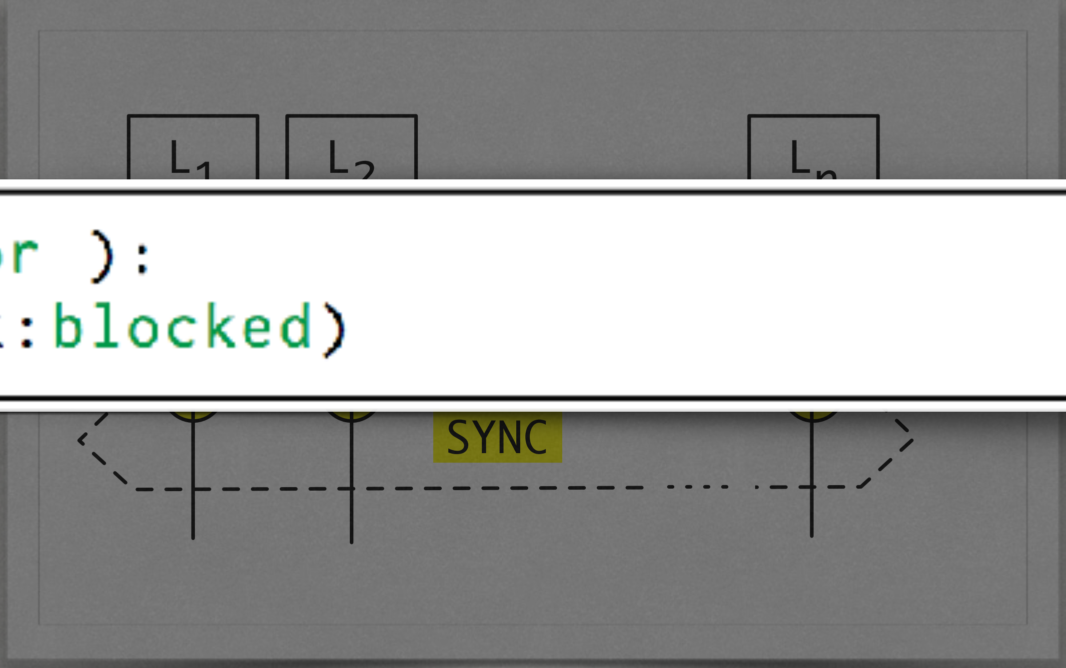
```
syncCAB(snc):
  bsync(request:Enabled(snc), block:Sync(snc))
  bsync(request:Sync(snc), block:VisibleEvents)
```

Sync

“Block `blocked` until `waitedFor` is selected”

- For each i in 1..n:
  - blockUntilCAB(Enabled(**snc**), Enter(l_i))
-
  Sync(**snc**)

```
blockUntilCAB( blocked, waitedFor ):
  bsync(waitFor:waitedFor, block:blocked)
```



```
syncCAB(snc):
  bsync(request:Enabled(snc), block:Sync(snc))
  bsync(request:Sync(snc), block:VisibleEvents)
```

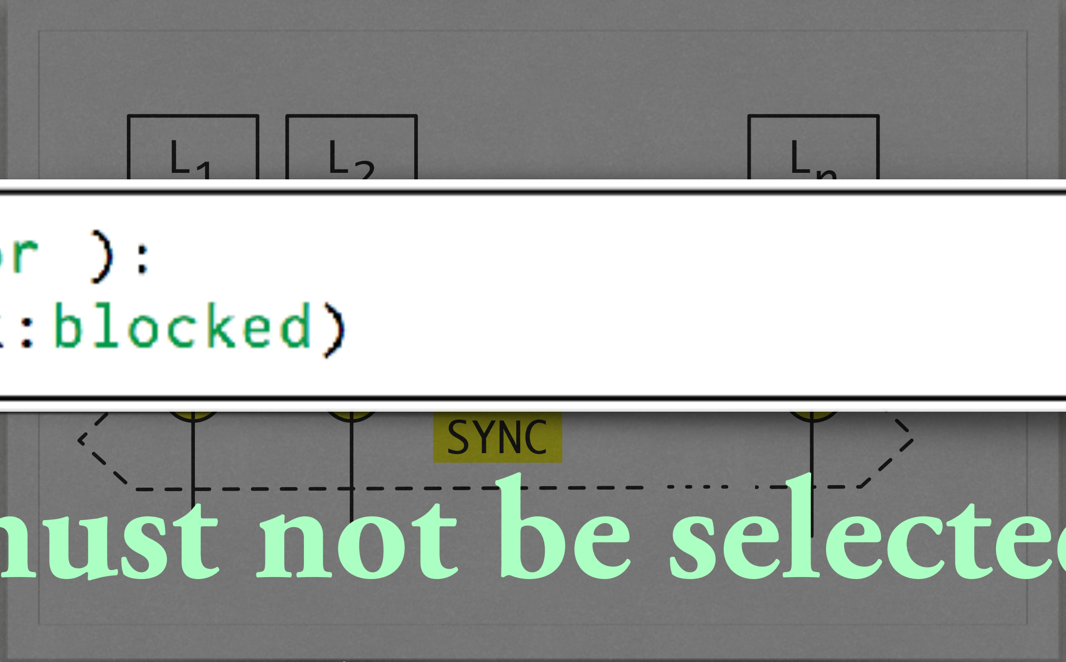
Sync

VISUAL DICTIONARY: LSC

“Block `blocked` until `waitedFor` is selected”

- For each i in 1..n:
  - `blockUntilCAB(Enabled(snc), Enter(l_i))`
- `blockUntilCAB(Leave(l_i)), Sync(snc))`

```
blockUntilCAB( blocked, waitedFor ):
  bsync(waitFor:waitedFor, block:blocked)
```

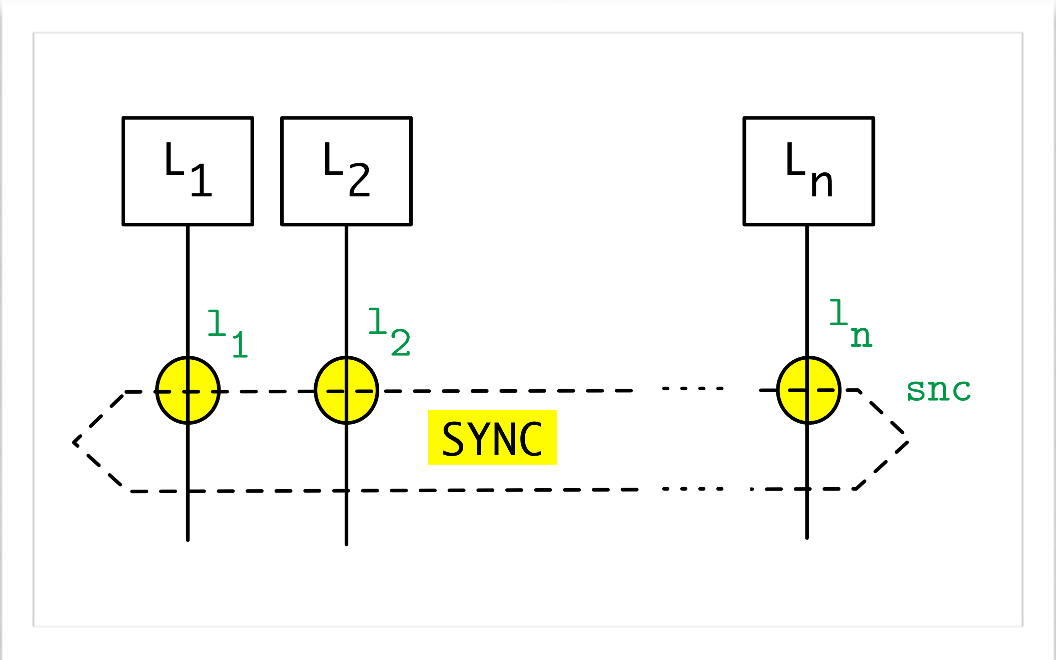


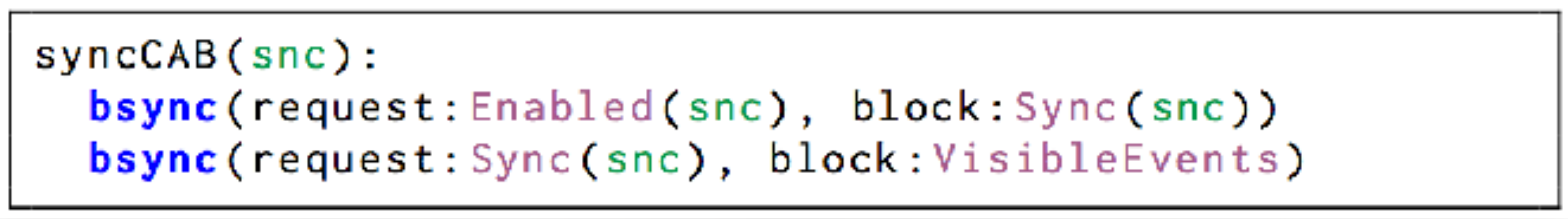
“`blocked` must not be selected until `waitedFor` is”

```
syncCAB(snc):
  bsync(request:Enabled(snc), block:Sync(snc))
  bsync(request:Sync(snc), block:VisibleEvents)
```

Sync

- For each i in 1..n:
  - blockUntilCAB(Enabled(**snc**), Enter(**l_i**))
- For each i in 1..n:
  - blockUntilCAB(Leave(**l_i**), Sync(**snc**))

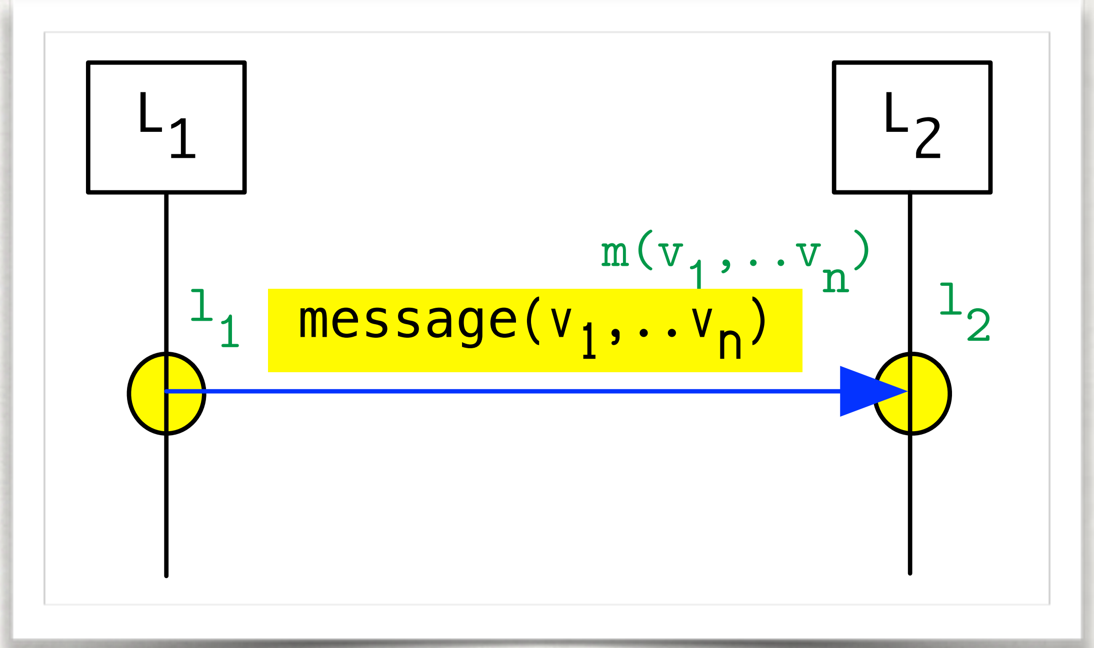


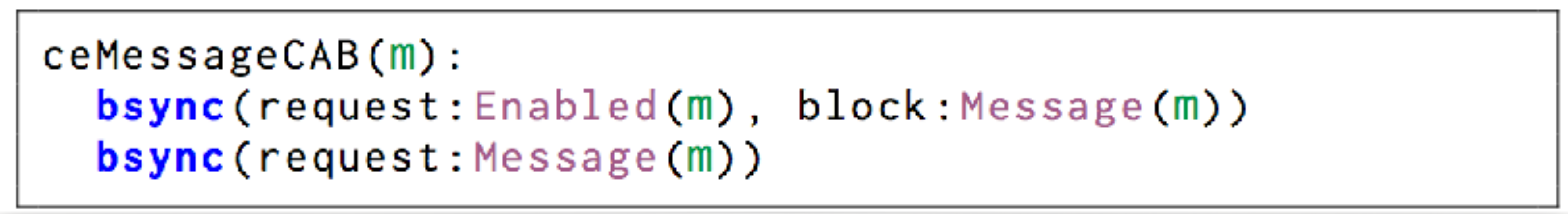


```
syncCAB(snc):
  bsync(request:Enabled(snc), block:Sync(snc))
  bsync(request:Sync(snc), block:VisibleEvents)
```

Cold, Executed Message

- BlockUntilCAB(Enabled(**m**), Enter(**l_1**))
- BlockUntilCAB(Enabled(**m**), Enter(**l_2**))
- BlockUntilCAB(Leave(**l_1**), Message(**m**))
- BlockUntilCAB(Leave(**l_2**), Message(**m**))
- For each non-affected variable **a**:
  - BlockUntil(Bound(**a**), Message(**m**))
- For each affected variable **a**:
  - BindFromCAB(**a**, Message(**m**))

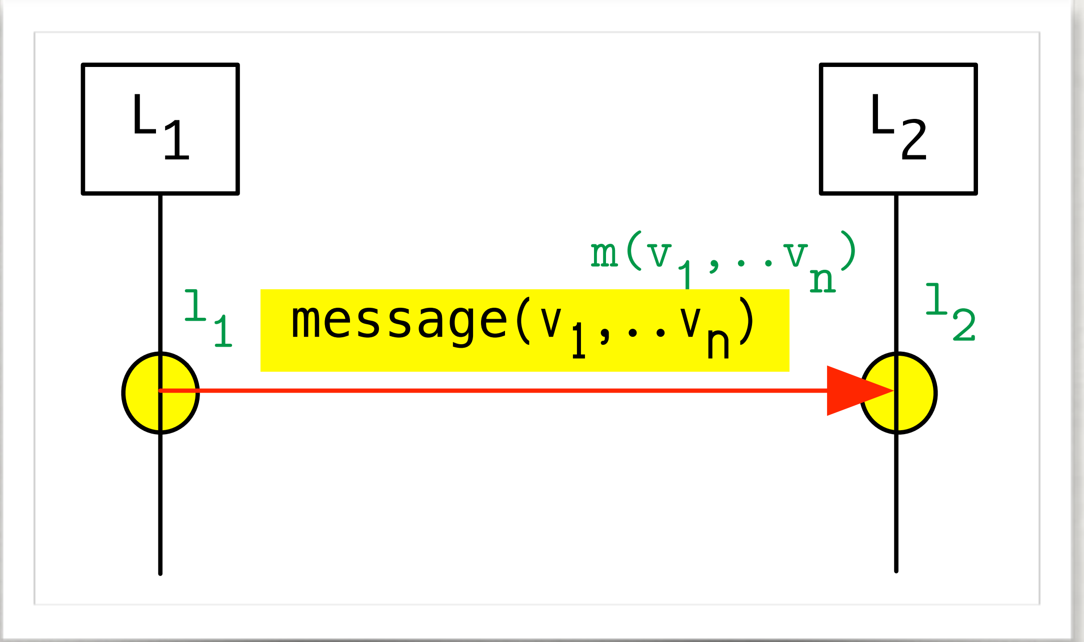


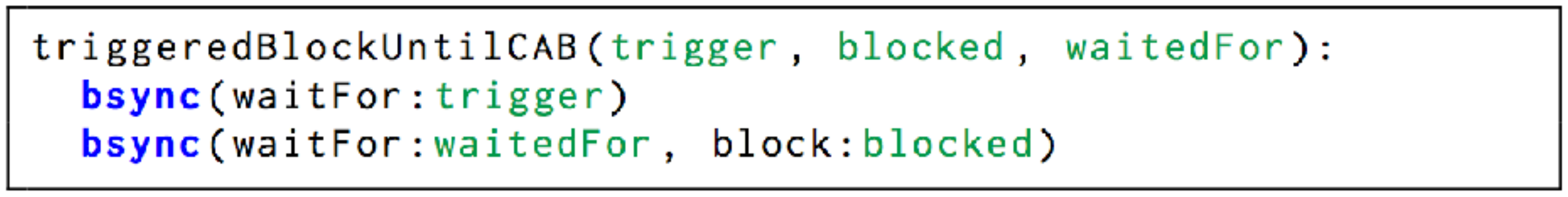


```
ceMessageCAB(m):
  bsync(request:Enabled(m), block:Message(m))
  bsync(request:Message(m))
```

Hot, Executed Message

- Same CABs as Cold, Executed Message
- triggeredBlockUntilCAB(Enabled(**m**), ExitEvents(**m**.parent), Message(**m**))

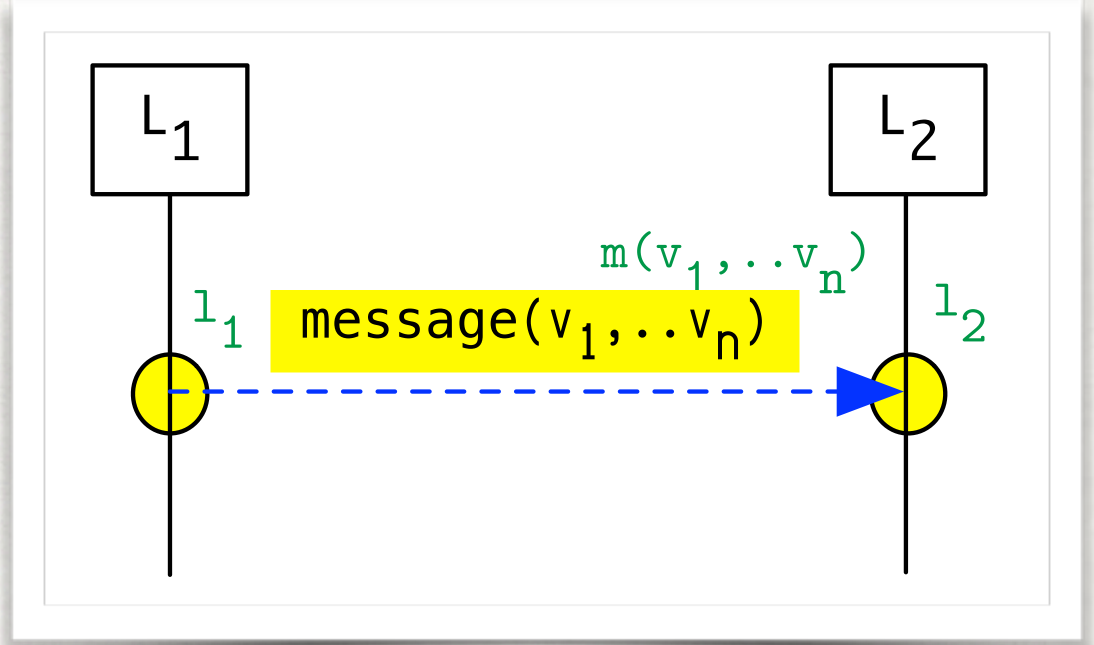


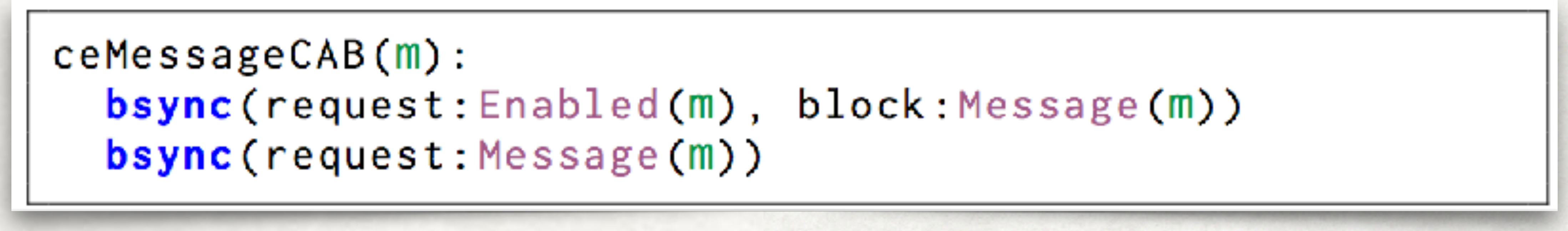


```
triggeredBlockUntilCAB(trigger, blocked, waitedFor):
  bsync(waitFor:trigger)
  bsync(waitFor:waitedFor, block:blocked)
```

Cold, Monitored Message

- `Same CABs as Cold, Executed message`
- `Small change to ceMessageCAB:`

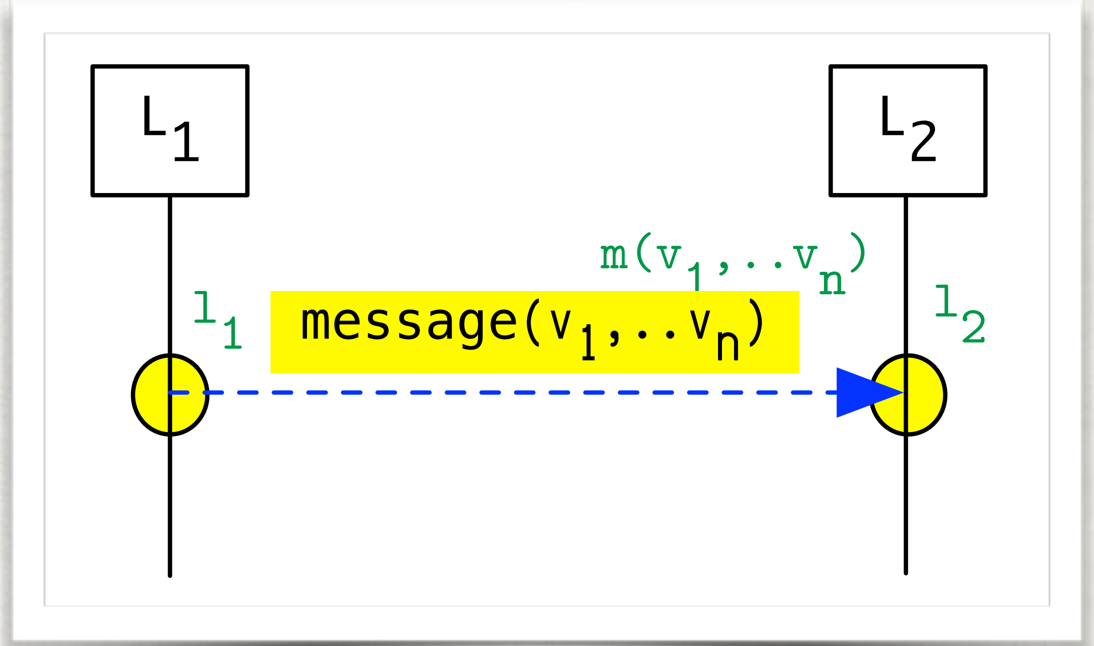


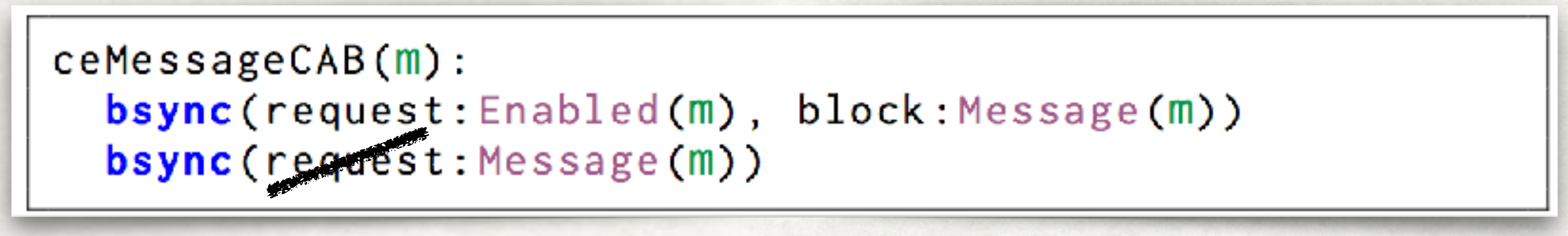


```
ceMessageCAB(m):
  bsync(request:Enabled(m), block:Message(m))
  bsync(request:Message(m))
```

Cold, Monitored Message

- Same CABs as Cold, Executed message
- Small change to ceMessageCAB:



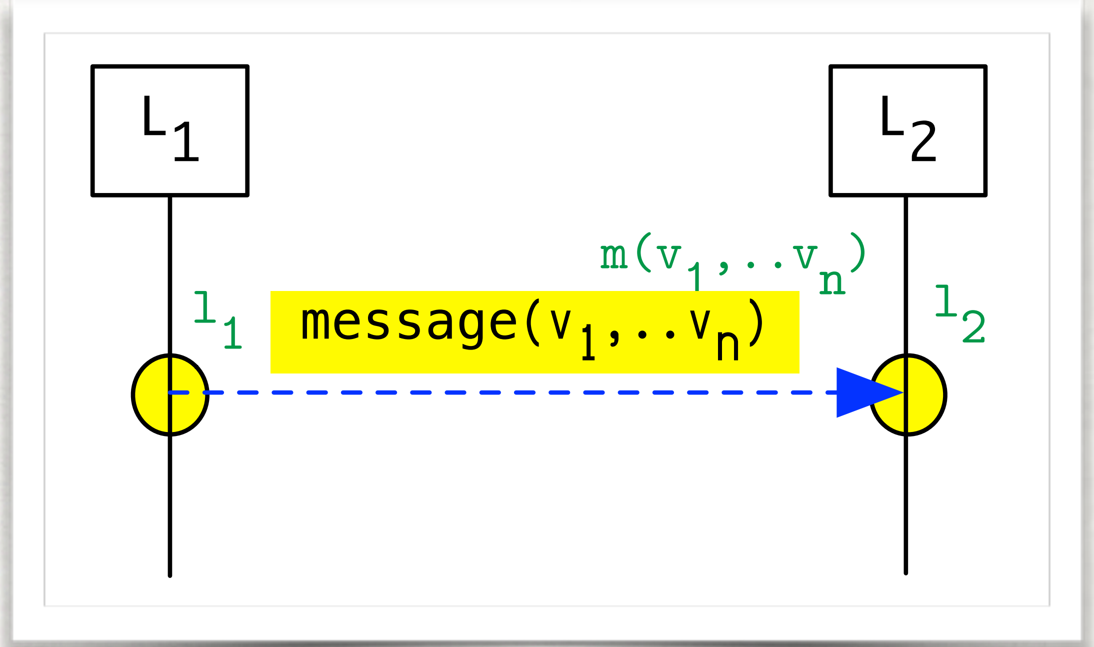


```
ceMessageCAB(m):
  bsync(request:Enabled(m), block:Message(m))
  bsync(request:Message(m))
```

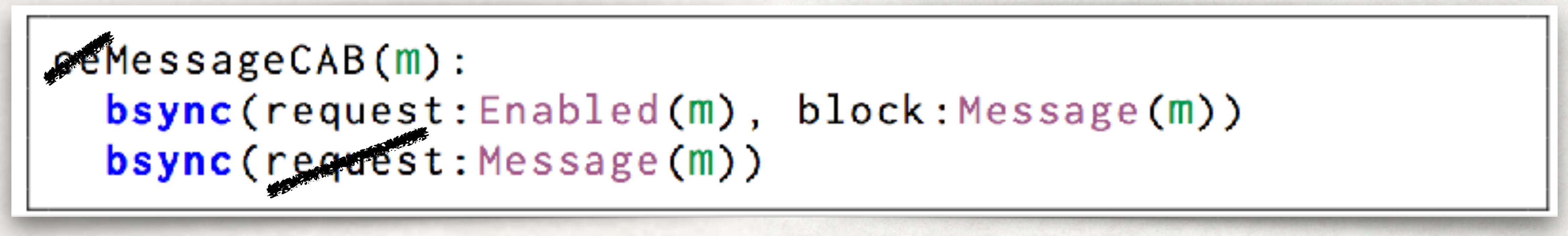
waitFor

Cold, Monitored Message

- Same CABs as Cold, Executed message
- Small change to ceMessageCAB:



cm

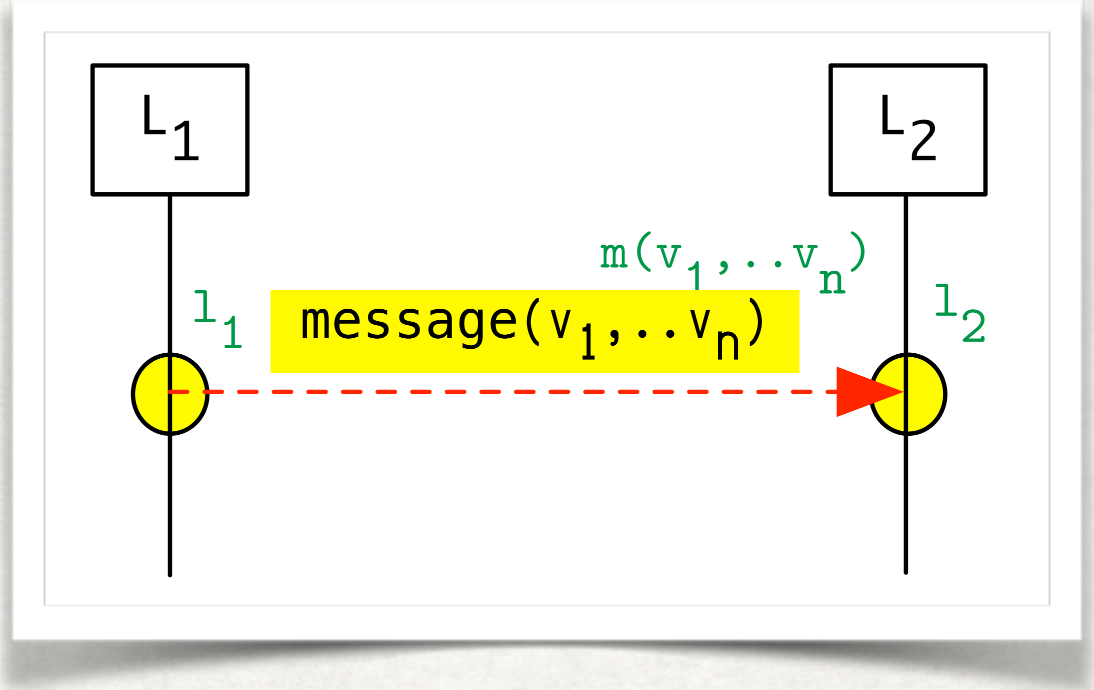


```
~~ce~~MessageCAB(m):
   bsync(request:Enabled(m), block:Message(m))
   bsync(~~request~~:Message(m))
```

waitFor

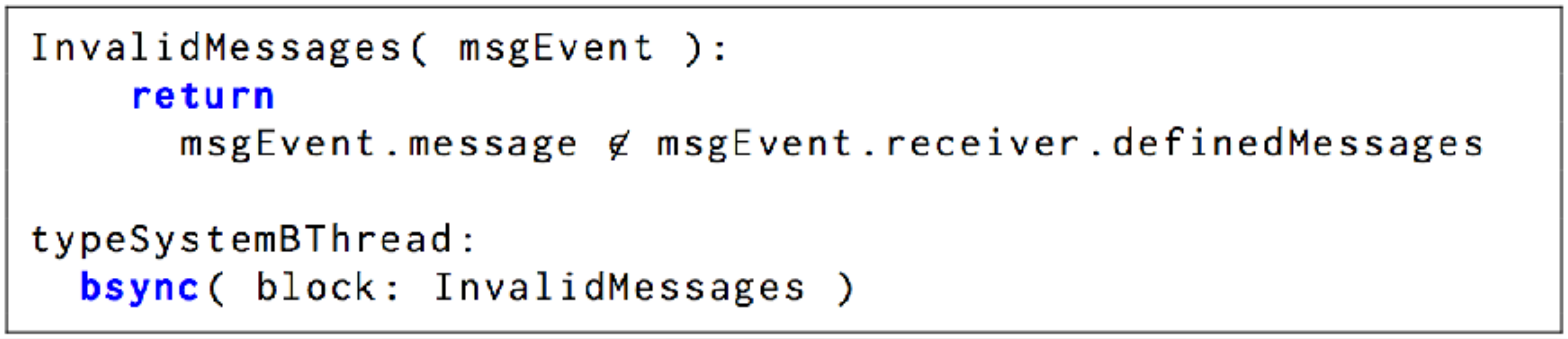
Hot, Monitored Message

- Same CABs as Cold, Monitored Message
- triggeredBlockUntilCab(Enabled(**m**), ExitEvents(**m**.parent), Message(**m**))



System-Wide Alteration: Type Safety

Prevent all messages that are not defined by the receiver from being sent.

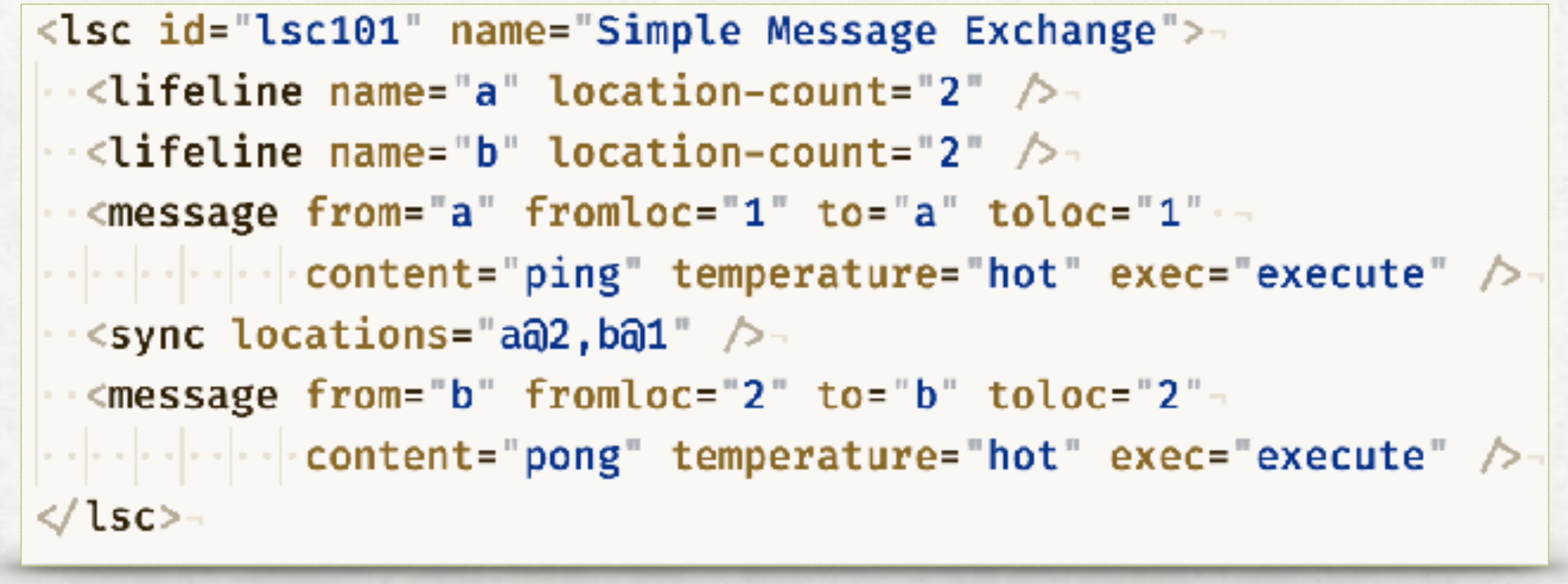


```
InvalidMessages( msgEvent ):
    return
      msgEvent.message ∉ msgEvent.receiver.definedMessages

typeSystemBThread:
  bsync( block: InvalidMessages )
```

Implementation

- LSCs represented in XML
- Queries and mapping done in XQuery
- Target language is Javascript, execution using BPjs
    - BP engine on top of Mozilla Rhino
    - Fork us on GitHub: github.com/michbarsinai/BP-javascript-search

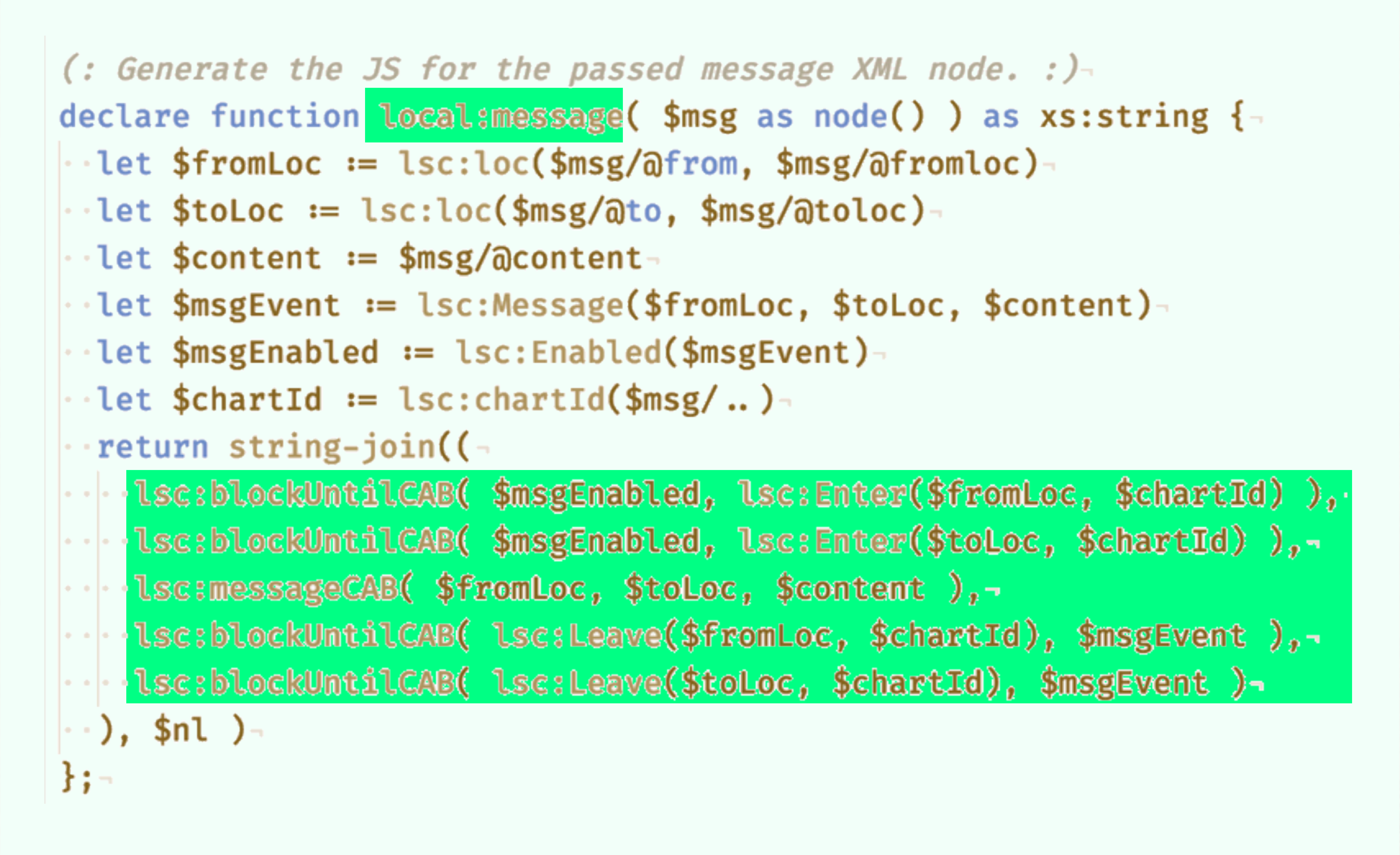


```
<lsc id="lsc101" name="Simple Message Exchange">
  <lifeline name="a" location-count="2" />
  <lifeline name="b" location-count="2" />
  <message from="a" fromloc="1" to="a" toloc="1"
           content="ping" temperature="hot" exec="execute" />
  <sync locations="a@2,b@1" />
  <message from="b" fromloc="2" to="b" toloc="2"
           content="pong" temperature="hot" exec="execute" />
</lsc>
```

Implementation

```
(: Generate the JS for the passed message XML node. :)
declare function local:message( $msg as node() ) as xs:string {
  let $fromLoc := lsc:loc($msg/@from, $msg/@fromloc)
  let $toLoc := lsc:loc($msg/@to, $msg/@toloc)
  let $content := $msg/@content
  let $msgEvent := lsc:Message($fromLoc, $toLoc, $content)
  let $msgEnabled := lsc:Enabled($msgEvent)
  let $chartId := lsc:chartId($msg/..)
  return string-join((
    lsc:blockUntilCAB( $msgEnabled, lsc:Enter($fromLoc, $chartId) ),
    lsc:blockUntilCAB( $msgEnabled, lsc:Enter($toLoc, $chartId) ),
    lsc:messageCAB( $fromLoc, $toLoc, $content ),
    lsc:blockUntilCAB( lsc:Leave($fromLoc, $chartId), $msgEvent ),
    lsc:blockUntilCAB( lsc:Leave($toLoc, $chartId), $msgEvent )
  ), $nl )
};
```

Implementation



```
(: Generate the JS for the passed message XML node. :)
declare function local:message( $msg as node() ) as xs:string {
  let $fromLoc := lsc:loc($msg/@from, $msg/@fromloc)
  let $toLoc := lsc:loc($msg/@to, $msg/@toloc)
  let $content := $msg/@content
  let $msgEvent := lsc:Message($fromLoc, $toLoc, $content)
  let $msgEnabled := lsc:Enabled($msgEvent)
  let $chartId := lsc:chartId($msg/..)
  return string-join((
      lsc:blockUntilCAB( $msgEnabled, lsc:Enter($fromLoc, $chartId) ),
      lsc:blockUntilCAB( $msgEnabled, lsc:Enter($toLoc, $chartId) ),
      lsc:messageCAB( $fromLoc, $toLoc, $content ),
      lsc:blockUntilCAB( lsc:Leave($fromLoc, $chartId), $msgEvent ),
      lsc:blockUntilCAB( lsc:Leave($toLoc, $chartId), $msgEvent )
  ), $nl )
};
```

Related Work

- Semantic Mapping: Done before. E.g:
  - **AToM3** [deLara, Vangheluwe, 2002]
  - **UML, fUML** [OMG]
  - Both use metamodels
- **“Coping with Semantic Variation Points in Domain-Specific Modeling Languages”** Latombe, Crégut, Deantoni, Pantel, B. Combemale *(EXE’15)*
  - Two-tier structure, top tier lists available options, lower tier selects options based on semantic variant

Thanks.

Questions?

Defining Semantic Variations of Diagrammatic Languages using Behavioral Programming and Queries