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

- 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.

The BP Execution Cycle

- Request
- Block
- Behavior Threads
- Wait
The BP Execution Cycle

- Request
- Behavior Threads
- Wait
- Block
The BP Execution Cycle

- Request
- Block
- Wait
- Behavior Threads
The BP Execution Cycle

- **Behavior Threads**
  - **Block**
  - **Request**
  - **Wait**
The BP Execution Cycle

- **Wait**
- **Block**
- **Request**
- **Behavior Threads**
The BP Execution Cycle
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))
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
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

Query

source diagram

result₁

result₂

resultₖ

Semantic Mapping

CAB₁₁

CAB₁₂

CAB₁ₙ

...

CAB₂₁

CAB₂₂

CAB₂ₘ

...

CABₖ₁

CABₖ₂

CABₖₜ

BThread Registration

BProgram
Concept Overview: Semantic Mapping

Source Language

Query

result

1

result

2

result

k

Semantic Mapping

CAB

1,1

CAB

1,2

CAB

1,n

•••

CAB

2,1

CAB

2,2

CAB

2,m

•••

CAB

k,1

CAB

k,2

CAB

k,w

•••

BThread Registration

BProgram
Concept Overview: Semantic Mapping

Source Language

Query

Semantic Mapping

result_1

result_2

result_k

BThread Registration

BProgram

CAB_1,1

CAB_1,2

CAB_1,n

CAB_2,1

CAB_2,m

CAB_k,1

CAB_k,2

CAB_k,w

Source Language

Behavioral Programming

source diagram

Semantic Mapping
Concept Overview: Semantic Mapping

Semantic Mapping

Query

Source Language

Language Definition

Behavioral Programming

source diagram

result_1

result_2

result_k

CAB_{1,1}

CAB_{1,2}

CAB_{1,n}

CAB_{2,1}

CAB_{2,1}

CAB_{2,m}

CAB_{k,1}

CAB_{k,1}

CAB_{k,2}

CAB_{k,w}

BThread Registration

BProgram
Concept Overview: Semantic Mapping

Source Language

Semantic Mapping

Behavioral Programming

Source-Semantic Events

Language Definition

Query

Source diagram

result_1

result_2

result_k

CAB_{1,1}

CAB_{1,2}

CAB_{1,n}

CAB_{2,1}

CAB_{2,1}

CAB_{2,m}

CAB_{k,1}

CAB_{k,2}

CAB_{k,w}

BThread Registration

BProgram

Behavioral Programming

Source-Semantic Events

Language Definition

Semantic Mapping

Query

Source diagram

result_1

result_2

result_k
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
lifelineCAB(chart, l₁, ..., lₙ):
    bsync(waitFor: ChartStart(chart))
    for (i ∈ [1..n]):
        if (lᵢ is at bottom of subchart):
            bsync(wait: Done(subchart),
                  block: ChartEnd(chart))
            bsync(request: Enter(lᵢ),
                  block: ChartEnd(chart), VisibleEvents)
            bsync(request: Leave(lᵢ),
                  block: ChartEnd(chart))
    loop until *
    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))`

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

\[
\text{blockUntilCAB( } \text{blocked, waitedFor } ):
\text{bsync(waitFor: waitedFor, block: blocked)}
\]

\[
\text{syncCAB(snc):}
\text{bsync(request: Enabled(snc), block: Sync(snc))}
\text{bsync(request: Sync(snc), block: VisibleEvents)}
\]
“Block blocked until waitedFor is selected”

```
blockUntilCAB( Enabled(snc), Enter(l1))
```

```
syncCAB(snc):
    bsync(request: Enabled(snc), block: Sync(snc))
    bsync(request: Sync(snc), block: VisibleEvents)
```
“Block blocked until waitedFor is selected”

```
blockUntilCAB( Enabled(snc),
             Enter(li))
```

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

```
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($m$))
Hot, Executed Message

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

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

```python
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`:

```plaintext
cm

```enumeration

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

```plaintext
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

```xml
<lscl 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>
```
( Generate the JS for the passed message XML node. :)

```javascript
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

```javascript
( : 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:
  • AToF3 [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
Defining Semantic Variations of Diagrammatic Languages using Behavioral Programming and Queries

Thanks.

Questions?