Defining Semantic Variations of Diagrammatic Languages Using Behavioral Programming and Queries

Michael Bar-Sinai
Computer Science Department
Ben-Gurion University
Be'er-Sheva, Israel

Gera Weiss
Computer Science Department
Ben-Gurion University
Be'er-Sheva, Israel

Assaf Marron
Faculty of Mathematics and Computer Science
Weizmann Institute of Science
Rehovot, Israel

***Abstract*—We present a methodology for describing executable semantics of diagrammatic modeling languages, and an execution engine based on such definition. Under proposed methodology, languages are defined using a set of pairs, composed of a query and a group of mappers. The queries, defined over the language's diagrammatic syntax, return language constructs. These constructs are mapped by the mappers to behavioral programming-based models. Resultant definition is executable, can inter-operate with similar definitions of other languages, and is accessible to practitioners who read code but shy away from transition formulae. We demonstrate our approach by defining a subset of the LSC language, and by implementing an LSC runtime engine based on that definition.**

***Index Terms*—Modeling; Computer Languages; Software Engineering;**

I. Introduction

Diagrammatic modeling languages hold great promise for software engineering, being able to depict — quite literally — structural and behavioral specifications. While some diagrammatic languages have been adopted for documentation and high-level design, their promise is still largely unrealized when it comes to describing executable models. Almost 30 years after Harel presented StateCharts [1], diagrammatic languages are still considered "doodles" by many practitioners [2].

A few factors that might be holding back the adoption of diagrammatic languages for execution are lack of accessible definition of executable semantics, and the absence of runtimes that can interoperate with text-based code.

This paper proposes an approach for defining executable semantics for diagrammatic languages. The definition is accessible to anyone who can read procedural formulation, and lends itself to runtime engine implementation. As the runtime is based on the Behavioral Programming paradigm, it can easily integrate with text-based code.

The rest of this paper is organized as follows: Section II briefly introduces Behavioral Programming. Sections III and IV introduce and discuss the concept of querying diagrams and mapping them to BP-based code, respectively. Sections V and VI apply the approach to a subset of the Live Sequence Chart language (LSC). Section VII describes a runtime tool for LSC, implemented based on those definitions. Section VIII demonstrates how the proposed approach can be used to accommodate semantic variation points. Section IX looks at related work, and Section X concludes.

II. A Quick Introduction to Behavioral Programming

Behavioral Programming (BP) [3], introduced by Harel, Marron and Weiss in 2012 [4], is a programming paradigm rooted in scenario-based programming. BP programs are composed of threads of behavior, called b-threads. B-threads run in parallel to coordinate a sequence of events via a synchronized protocol, as follows. During program execution, when a b-thread wants to synchronize with its peers, it submits a statement to the central event arbiter. This statement declares which events it requests to be selected, which events it waits for (but does not request), and which events it would like to block (prevent from being selected). When all b-threads have submitted their statements, the arbiter selects an event that was requested but not blocked. It then wakes up the b-threads that requested or waited for that event. The rest of the b-threads remain at their state, until an event they requested or waited for is selected. Blocked and waited-for events can be described using a predicate or a list. Requested events have to be specified explicitly. In this paper, submitting the synchronization statement is done by calling `bsync`.

III. Semantic Mapping to Behavioral Programming

When defining semantics for a diagrammatic language, we propose the language developer defines a set of pairs, composed of a query and multiple mappers, and a set of BP events called *Source-Semantic Events*. The queries, defined over the source language's terms and semantics, take a diagram and return a set of language constructs. The BP-mappers take each returned construct and generate a set of b-threads, called *construct agent b-threads*, or *CABs*.

CABs act on behalf of their construct during program execution. Source Semantic Events are used to signal events in the original program, e.g. "message passed" for LSCs or

"step completed" in an Activity Diagram [5]. The mapping process is described in Figure 1.

The set of query-mapper pairs define the executable semantics of the diagrammatic language. Run together, CABs generated by query-mapper pairs produce a valid execution of the mapped program. An execution of a program is considered "valid" if the order of the Source Semantic Events in its trace complies with the requirements of the diagrammatic language.

From a BP point of view, there is nothing special about a CAB or an source-semantic event. While both carry special semantics for the diagrammatic program, these semantics are only present in the interpretation of the behavioral program's event log — not while the program is executing.

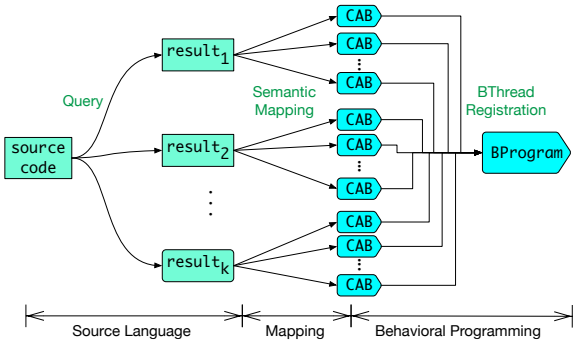


Fig. 1. Defining operational semantics for diagrammatic languages using querying and mapping. Queries select constructs from a diagram. Selected constructs are mapped to one or more *Construct Agent B-threads (CABs)*. Run together, those CABs generate a valid execution of the original program.

IV. Discussion

Describing the semantics of a formal diagrammatic language by mapping its constructs to BP is advantageous for a number of reasons. Resultant definitions are both formal and accessible, as they use simple code snippets rather than transition formulae. Furthermore, the definitions are executable, so language developers can test and verify them, and readers can write programs to test their understanding. This combination of readability, formality and executability improves upon existing practices.

Formality and executability are crucial for removing ambiguities. Obviously, non-formal definitions might be ambiguous (e.g. Chan et. al. about Java [6] and Fecher et. al. about UML2.0 [7]). But even fully formal semantics definitions that use transition formulae are prone to errors, as shown by Klein et. al. in [8].

The structure of the resultant definition is intuitive and easy to navigate, as the query-mappers pair structure is similar to that of a language reference. Our experience is that many CABs can be reused across multiple constructs. As CABs define semantics, this is not just regular code reuse, but also concept and comprehension reuse.

The proposed approach allows language developers to independently mix and match semantic variations of language constructs, as changing the semantics of a single construct is done by changing the relevant mapper only. Adding and removing language constructs can be experimented with in a similar way (See Section VIII).

Additionally, this type of definition allows for program analysis. For example, model checking tools such as BPMC [9], can verify that under a given specification, a certain event will always precede another.

Finally, since BP serves as common ground to diagrammatic languages defined using the proposed approach, these executable definitions allows for language interoperation. We have demonstrated such interoperation between LSC and UML Activity Diagram.

Our proposed approach is, of course, not perfect. When a language construct is mapped to many CABs, the reader is required to keep in mind the state of those CABs in order to fully comprehend that construct's behavior. This issue is somewhat alleviated by CAB comprehension reuse. Another issue is that the definition relies on BP, which is still in early adoption stages.

V. Case Study: Semantic Variations of LSC

We will now demonstrate our approach by creating the operational semantics of a subset of Live Sequence Charts (LSC). LSC is a diagrammatic programming language that extends classical message sequence charts, mainly with a universal interpretation and must/may, monitor/execute modalities. The language was developed by Damm and Harel [10] and was first implemented by a tool called Play-Engine [11]. A UML compliant variant is implemented by the PlayGo tool [12]. The semantics of the PlayGo version, which slightly differ from Play-Engine's, are described in [13].

An LSC system consists of scenarios and objects. Each scenario describes a facet of the system's behavior and is described in a live sequence chart (an LSC). The overall system behavior is the result of concurrent execution of all the LSCs it contains. An example LSC is shown in Figure 2.

Objects, which appear in LSCs as lifelines, can send messages to each other or to themselves. Sending these messages is depicted in the charts by horizontal arrows between lifelines. Messages can be tagged as must occur ("hot", red) or may occur ("cold", blue), and as executed ("execute", solid) or waited for ("monitor", dashed). The Play-Engine variant supports both synchronous and asynchronous messages; in PlayGo, all messages are synchronous.

Each LSC is comprised of lifelines and messages. LSCs may contain variable assignments and flow-control elements, such as loops and conditional execution. Special kinds of lifelines represents the user and the environment. Conditional guards, shown as elongated hexagons, specify statements that must be true for the execution to continue. A special condition called *SYNC*, always evaluates to true and is used to synchronize lifelines.

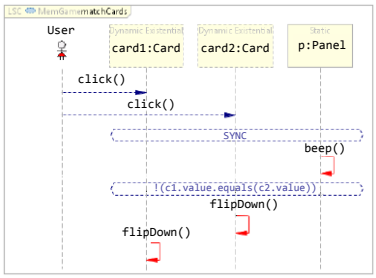


Fig. 2. LSC describing a basic move in a card memory game. After the user clicks two cards, a beep is issued, and the two cards are compared. If the cards are different, they are flipped back down. The `click` method shows the cards' faces, and the `flipDown` method turns them over again. The first two events may or may not happen, are thus *cold* (blue). The ensuing three events must happen once the first two events have, and are thus *hot* (red). The `Sync` construct forces the `Beep` to occur *after* the second `click`.

The point where a lifeline intersects with a message, a condition, or any other language construct, is called a *location*. During execution, lifelines proceed along their locations in downward vertical order. The collection of all current lifeline locations in an LSC, called a *cut*, is the equivalent of a program counter in traditional code.

The execution of an LSC consists of a series of events, such as message passing or condition evaluations. An event is called *enabled* when all its preconditions have been met — involved lifelines have arrived at their respective locations, and variables have been bound, etc. At runtime, the LSC engine repeatedly selects an enabled event for execution. Then lifelines move to their next locations, and the chart's cut is updated.

The system avoids executing forbidden events whenever possible. Forbidden events may be specified in various ways: A scenario of events ending in a hot false condition is considered forbidden. When an LSC is "strict", all events that appear in it and are not enabled by its cut are forbidden. Finally, the Play-Engine variant allows tagging individual events as forbidden in some designated scope. When a forbidden event is nevertheless executed, an exception called *violation* occurs.

LSC is an interesting language for demonstrating our proposed approach, since it is a real-world diagrammatic language with non-trivial semantics. Additionally, it has multiple semantic variants, which allows us to mix and match semantics of specific constructs.

This paper focuses on the operational semantics of a single LSC and on a subset of the constructs. The construct subset was selected such that it contains examples for all construct types, and so it can be intuitively extended.

VI. A Visual Dictionary for LSC

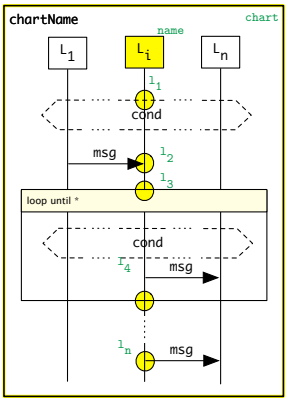
In this section we present a visual dictionary that lists the syntactic query and BP-mapping for each LSC construct. Query matches are highlighted with a yellow background. For example, in the Sync definition (Subsection VI-B), the SYNC hexagon and the intersection of its upper edge are matched, and are labeled `snc` and l_1 to l_n for the purpose of the BP-mapper pseudo-code that follows. CABs composing the BP-mapping for the construct matched by each query appear after the query's diagram. Re-used CABs are referred to by name, and listed at Subsection VI-J. This graphical representation of the queries is an idea for future implementations that may allow for an intuitive specification of language constructs. In our current implementation we use a textual query language and the graphical representation is compiled manually for the sake of readability.

All CABs exit when an exit event of their parent chart is triggered. This is done using the BP idiom of *break-upon*: b-threads terminate when an event whose a member of their set of break-upon events is triggered. We use break-upon for readability — it is possible to simulate it using pure Request-WaitFor-Block, by adding the break-upon event set to the `wait` parameter of every **`bsync`**, and then exiting if the triggered event is a member of this set.

BP allows blocking and waiting for abstract sets of events. The model in this paper uses two such event sets: `VisibleEvents`, which contains all message passings, and `ExitEvents(chart)`, which contains all events signaling execution termination of a chart or a subchart.

Code listings in this paper serve both as an implementation and as a specification. Thus, we invite the reader to read them as both imperative and declarative. As an imperative code, **`bsync`**`(waitFor:E, block:VisibleEvents)` reads "wait for `E`, and until then block all visible events". When read declaratively, it says "No visible event can happen until `E` does".

A. Lifeline



Lifeline CABs are responsible for advancing the chart's cut. A `lifelineCAB`, started by its parent chart, begins by waiting for its parent chart's start event (first **`bsync`**). It then advances along its locations, repeatedly requesting to enter and leave them. During execution, a `lifelineCAB` blocks its parent chart from ending normally.

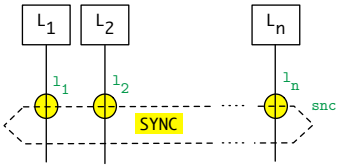
During the execution of subcharts, such as Loop (Subsection VI-I), `lifelineCAB`s wait for the subchart to be over;

inside the subchart, new lifelineCABs act on behalf of the lifelines. This is achieved by the **if** statement at the top of the iteration loop.

- Lifeline CAB:

```
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))
```

B. Sync

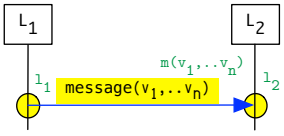


- For each $i \in \{1,\ldots,n\}$ instantiate a BlockUntilCAB, blocking Enabled(snc) until Enter(l_i) is selected.
- For each $i \in \{1,\ldots,n\}$ instantiate a BlockUntilCAB, blocking Leave(l_i) until snc is selected.
- Instantiate the following CAB:

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

This CAB enforces the SYNC to be enabled prior to being triggered. By blocking VisibleEvents in the second **bsync**, this CAB ensures that once enabled, the SYNC will be triggered prior to any visible event.

C. Cold, Executed Message



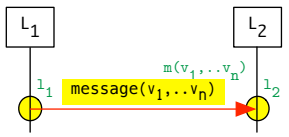
- Instantiate two BlockUntilCABs, blocking Enabled(m) until Enter(l_i) is selected (for $i \in \{1,2\}$). In case the sender lifeline is also the receiver, it is possible to omit one of these CABs.
- Instantiate two BlockUntilCABs, blocking Leave(l_i) for $i \in \{1,2\}$ until Message(m) is selected.
- For each variable v not affected by m, instantiate a BlockUntilCAB, blocking Enabled(m) until Bound(v) is selected. These CABs prevent m from being enabled until all variables it depends on are bound.
- For each variable v affected (bound) by m, instantiate a BindFromCAB, binding v when Message(m) is selected. These CABs announce the binding made by m for v.
- Instantiate a single ceMessageCAB (shown below):

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

This CAB forces the message passing event to be enabled prior to being triggered. The concept of an event being "enabled" is part of the LSC language, and so Enabled(m) serves as a source semantic event.

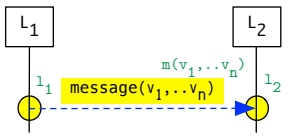
D. Hot, Executed Message

A "hot" message has to be executed once it was enabled (unlike a "cold" message which may or may not happen). The difference in behavior is achieved by adding a single CAB, as shown below. The rest of the CABs are reused.



- Same CABs as for the cold, executed message.
- A TriggeredBlockUntilCAB, where the trigger event is Enabled(m), after which the event set ExitEvents(chart) is blocked until Message(m) is selected.

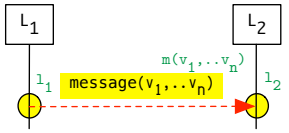
E. Cold, Monitored Message



- Except for the ceMessageCAB, all CABs used for the cold, executed message (Subsection VI-C) are reused "as is" for the cold, monitored one.
- Instantiate a single cmMessageCAB (below). This CAB is identical to the ceMessageCAB, except for the second **bsync** call which waits for the message passing event rather than requests it.

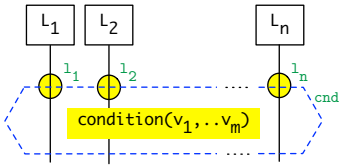
```
cmMessageCAB(m):
  bsync(request:Enabled(m), block:Message(m))
  bsync(waitFor:Message(m))
```

F. Hot, Monitored Message



- Same CABs as for the cold, monitored message.
- An additional TriggeredBlockUntilCAB, as for the hot, executed message.

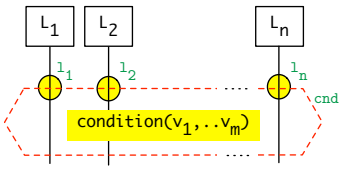
G. Cold Condition



- For $i \in \{1..n\}$, instantiate a `BlockUntilCAB`, blocking `Enabled(cnd)` until `Enter(l_i)` is selected.
- For $i \in \{1..n\}$, instantiate a `BlockUntilCAB`, blocking `Leave(l_i)` until `Condition(cnd)` is selected.
- Instantiate a single `coldConditionCAB` (below) with the matched parameters.

```
coldConditionCAB( cnd ):
  bsync(request:Enabled(cnd),
        block:Condition(cnd))
  if ( evaluate(cnd) ):
    resultEvent = Condition(cnd)
  else:
    resultEvent = ColdViolation(cnd)
  bsync(request:resultEvent,block:VisibleEvents)
```

H. Hot Condition

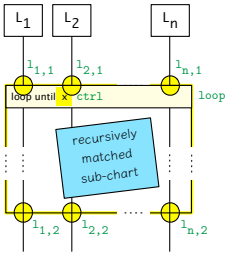


- Same as cold condition (Subsection VI-G), except that `hotConditionCAB` requests a `HotViolation` event.

I. Loop

An LSC loop is a type of a subchart: It can contain any LSC construct, including other loops, but uses the lifelines of its parent chart. A cold violation of a loop terminates it, but not its parent chart.

For every match of:



Instantiate the following CABs:

- `BlockUntilCAB`, waiting for `Leave(l_{i,1})`, while blocking `Enabled(loop)`.
- `loopCAB` (see below) with the matched parameters.

```
loopCAB( loop, ctrl ):
  bsync( request: Enabled(loop) )
  loopIteration(loop, ctrl)
  bsync(request:Done(loop), block:VisibleEvents)

loopIteration( loop, ctrl ):
  if ( ctrl > 0 ):
    startCABs(loop)
    bsync(request:ChartStart(loop),block:VisibleEvents)
    event = bsync(request:ChartEnd(loop),
                  waitFor:ExitEvents(loop))
    if ( event is ChartEnd(loop) ):
      loopIteration(ctrl-1, loop)
```

A `loopCAB` starts by requesting its loop is enabled. It then runs the loop repeatedly, using the sub-routine `loopIteration`. After the last iteration, it announces the loop is done by requesting a `Done(loop)` event. Blocking `VisibleEvents` ensures that once the loop is done, it is declared as such prior to any message being passed. The `loopIteration` sub-routine starts by checking whether a new iteration is needed. If so, it creates the CABs for the `loop` subchart, and requests its start event to be fired. This event signals the lifeline CABs of the subchart to start advancing along their location list. When the subchart execution is done, `loopIteration` checks whether the subchart ran to completion. If so, another iteration is attempted[1]. While the loop construct is executing, the lifelines of the parent chart `waitFor` it to end before entering their next location, below the chart.

J. Reused CABs

Common behavior aspects of constructs of the LSC language are captured for reuse by the following CABs:

1) BlockUntilCAB: This CAB block an event until another one is triggered. Used in all constructs to enforce correct execution order, the code for this CAB consists of a single `bsync` call.

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

2) TriggeredBlockUntilCAB: This CAB waits for an event, and then blocks a set of events until another event is selected.

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

3) BindFromCAB: This CAB is responsible for binding a single variable to a value extracted from a message. It waits for the message to be sent, and then requests a binding event announcing the new binding.

```
bindFromCAB( message, variable ):
  selected = bsync( waitFor:message )
  value = selected.message.get(variable)
  bsync( request:Bound(variable, value),
         block:VisibleEvents )
```

VII. Implementation

In order to test our approach, we implemented an LSC runtime engine. The engine takes an XML description of an LSC as input. It then uses XQuery [14] for both querying the source code and for generating BP code. The BP code is then executed normally, using a standard BP library.

Input: LSCs are described using straightforward XML-based format. Reminiscent to a verbal description of an LSC, the format includes nodes such as `<lifeline>` and `<message>`.

[1]For the special case of control value $*$, which means unbounded amount of iterations, we define $*-1=*$.

Queries and BP-Mappers: Queries over the XML files are done using XQuery. We use the BaseX [15] query engine which implements the XQuery 3.1 W3C standard [14], with no modifications. As LSCs contain recursive structures, the XQuery program consists of a top-level recursive query which is used to query to top-level chart. It then recurses down the chart containment hierarchy, querying over each construct it finds and mapping it to BP code. Construct queries look very much like the construct definitions listed above, phrased in XQuery (see Listing VII.1).

```
declare function local:message( $msg as node() )
 as xs:string {
  let $l1 := lsc:loc($msg/@from, $msg/@fromloc)
  (* more value definitions (omitted) *)
  return string-join((
    lsc:blockUntilCAB($msgEnabled,lsc:Enter($l1,$chartId)),
    lsc:blockUntilCAB($msgEnabled,lsc:Enter($l2,$chartId)),
    lsc:messageCAB($l1,$l2,$content),
    lsc:blockUntilCAB(lsc:Leave($l1,$chartId),$msgEvent),
    lsc:blockUntilCAB(lsc:Leave($l2,$chartId),$msgEvent)
  ), $newline )
};
```

Listing VII.1. XQuery code for detecting <message> nodes and generating the BPjs-based Javascript code that implements them. Compare this to the definition of cold, executed message (Subsection VI-C). The methods called by this query, such as lsc:messageCAB, return Javascript code.

BP engine: Generated BP code uses Javascript and the BPjs [16] library. BPjs was originally developed for [17]. As part of this work, we have heavily modified it to become a general purpose BP library. These modifications, however, did not include changes to specifically accommodate code generated by the XQuery part of the engine.

Using this engine, LSCs described using our XML format can be directly executed. The code is available at https://github.com/michbarsinai/BP-javascript-search.

VIII. Semantic Variations

Previous sections presented the concept of querying source code and mapping the results to BP as a mechanism for "breathing semantics into diagrammatic languages", to paraphrase [10]'s title. This mechanism allows for independently adding, removing, and altering the semantics of each construct, and for adding new constructs or removing them altogether. This section demonstrates such alterations, using the LSC example presented above.

A. Asynchronous Message

The messages described in Section VI are synchronous. We will now add an *asynchronous* message, which allows the sending lifeline to advance passed the send location without waiting for the receiving lifeline to receive the message. This type of messages exists only in the Play-Engine variant; PlayGo does not support it.

Instantiate the following CABs:

- BlockUntilCAB, blocking Enabled(m), waiting for Enter(l_1).
- BlockUntilCAB, blocking Received(m), waiting for Enter(l_2).
- BlockUntilCABs, blocking Received(m), waiting for Sent(m).
- BlockUntilCAB, blocking Leave(l_1), waiting for Sent(m).
- BlockUntilCAB, blocking Leave(l_2), waiting for Received(m).
- For each variable v not affected by m, a BlockUntilCAB, blocking Enabled(m), waiting for Bound(v).
- For each variable v affected (bound) by m, a BindFromCAB, blocking Sent(m), waiting for v.
- A single asyncMessageSendCAB (shown below), forcing the message sending event to be enabled prior to being triggered.

```
asyncMessageSendCAB(m):
  bsync(request:Enabled(m), block:Sent(m))
  bsync(request:Sent(m))
```

- A single asyncMessageReceiveCAB, requesting that the message is received.

```
asyncMessageReceiveCAB(m):
  bsync(request:Received(m))
```

  As there are blockUntilCABs blocking the message from being received prior to being enabled, fully bound, sent, and having a lifeline in location to receive it, this CAB does not need to handle all these preconditions and their possible orderings.

The mapping for asynchronous messages can be used either as a replacement for the synchronous message mapping, making all messages asynchronous (e.g. in a "what if we made all messages asynchronous" scenario), or as a mapping for a new type of specification idiom.

B. Strict vs. Tolerant Modes

An LSC can be *strict* or *tolerant*. A Strict LSC is violated if an event that appears in it happens when it is not enabled. Such event will not cause any violation for a tolerant LSC. In PlayGO, all LSCs are strict. In Play-Engine, both modes are allowed.

Strictness can be imposed by adding b-threads to the mapping of a chart, one for each message appearing in the LSC. Each b-thread is initialized with the events that are present in its respective chart. The function cut(chart) returns the cut of chart, which is the set of all the locations its lifelines are currently in. Obtaining the cut of a given chart does not require direct communication with that chart or its lifelines — the cut can be obtained by waiting for the Enter events of that chart's locations. A cut is considered HOT if at least one of its locations is HOT.

```
chartEvents = events_of( chart )
nonBlocked = ∅
repeat:
  event = bsync( waitFor: AllEvents,
                 block: chartEvents \ nonBlocked )
  if ( event is Enabled(x) ):
    nonBlocked = nonBlocked ∪ {x}
  else if ( event ∈ nonBlocked ):
    nonBlocked = nonBlocked \ {last_event}
  else:
    if ( isHot(cut(chart)) ):
      bsync(request:HotViolation, block:VisibleEvents)
    else:
      bsync(request:ColdViolation, block:VisibleEvents)
```

Listing VIII-B.2. Enstrictor, a b-thread that makes an LSC strict

C. Adding a Type System by Blocking

The LSC implementation presented so far uses dynamic typing, as it does not verify that receiving lifelines implement the messages they receive. Using event blocking, we can block messages unimplemented by their receiver in a purely incremental manner, by adding a b-thread.

BP allows b-threads to block sets of events by passing a predicate to `bsync`. `InvalidMessages`, defined in Listing VIII-C.3, is a predicate valid for all messages unimplemented by their receiver. In order to prevent these messages, the `typeSystemBThread` can be added to the system.

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

typeSystemBThread:
  bsync( block: InvalidMessages )
```

Listing VIII-C.3. Type System Event Set and BThread. This code assumes each lifeline has a list of defined operations

Traditionally, when typing constraints are imposed on a program, they are imposed on all of it. Our proposed approach of imposing typing constraints offers more flexibility: It can be lifted or imposed with no change to the rest of the code. It can be limited to parts of the code by altering the predicate. Or, it can pass the invalid method call to a special handler that can perform any arbitrary operation. This is somewhat similar to SmallTalk's `doesNotUnderstand:` method, invoked when an object receives a message it has no method for.

IX. Related Work

The notion of describing semantics by mapping one domain onto another is not new. AToM3 [18], for example, is a tool for creating and transforming meta-models, uses graph-based meta-models and transformations to achieve this. UML [5] has its own metamodel, used to describe its diagrams. Executable UML [19] (also known as fUML) adds executable semantics to a subset of UML's diagrams. Our work differs from both in that it does not use a metamodel per se — the queries extract data from the source language, but the result is still in the source language, not in a metamodel.

In [20], Latombe et. al. presented a way of coping with semantic variations in domain specific modeling languages. They use a 2-tier structure, where the top tier lists all options according to a set of available semantics, and the lower level selects the correct option according to the desired semantic variant. Our approach differs in that it uses changes in queries and mappers to vary the semantics. Thus, options not available by the selected variant are not generated.

This work was also partly motivated by the call expressed in [21], for endowing the conventions behind complex diagrams (biological ones, in the case of [21]) with explicit formal semantics.

X. Conclusion

By querying the syntax of a diagrammatic language and mapping the result to BP-based code, we can formally define the operational semantics of said language. Resultant definition has a is executable, accessible to practitioners who shy from state change formulae but read code readily, and allows the language developer to independently experiment with different semantics of language constructs. We have demonstrated the proposed approach using a subset of the LSC language, and presented a working software tool based on that definition.

References

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[2] D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of "Semantics"?" *IEEE Computer Magazine*, 2004.

[3] A. Marron. Behaviroal programming website. [Online]. Available: http://www.b-prog.org

[4] D. Harel, A. Marron, and G. Weiss, "Behavioral programming," *Communications of the ACM*, vol. 55, no. 7, 2012.

[5] OMG, *Unified Modeling Language Superstructure Specification, v2.0*, Aug. 2005. [Online]. Available: http://www.omg.org

[6] J. Chan and W. Yang, "Ambiguities in java," *CTHPC*, vol. 2, pp. 51–62, 2002.

[7] H. Fecher, J. Schönborn, M. Kyas, and W. de Roever, "29 new unclarities in the semantics of uml 2.0 state machines," *Formal Methods and Software Engineering*, pp. 52–65, 2005.

[8] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flat, J. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. Findler, "Run your research, on the effectiveness of lightweight mechanization." POPL, January 2012, pp. 285–296.

[9] D. Harel, R. Lampert, A. Marron, and G. Weiss, "Model-Checking Behavioral Programs," in *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, 2011, pp. 279–288.

[10] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *J. on Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[11] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[12] D. Harel, S. Maoz, S. Szekely, and D. Barkan, "PlayGo: towards a comprehensive tool for scenario based programming," in *ASE*, 2010.

[13] A. Marron and S. Szekely, *LSC Language Reference Manual*. Department of Computer Science and Applied Mathematics Weizmann Institute of Science, 04 2014.

[14] J. Robie, M. Dyck, and J. Spiegel, *XQuery 3.1: An XML Query Language*, Std., 2015. [Online]. Available: http://www.w3.org/TR/xquery-31/

[15] C. Grun, "Pushing XML Main Memory Databases to their Limits," 2006.

[16] M. Bar-Sinai and M. Weinstock. BP-Javascript. [Online]. Available: https://github.com/michbarsinai/BP-javascript-search

[17] M. Weinstock, "A behavioral programming approach to search based software engineering," in *Proceedings of the ACM Student Research Competition at MODELS 2015*.

[18] J. de Lara and H. Vangheluwe, "Atom3: A tool for multi-formalism and meta-modelling." in *FASE*, ser. Lecture Notes in Computer Science, R. Kutsche and H. Weber, Eds., vol. 2306. Springer, 2002, pp. 174–188.

[19] OMG, *Semantics Of A Foundational Subset For Executable UML Models (FUML™) v1.2.1*, January 2016. [Online]. Available: http://www.omg.org/spec/FUML/1.2.1/

[20] F. Latombe, X. Crégut, J. Deantoni, M. Pantel, and B. Combemale, "Coping with Semantic Variation Points in Domain-Specific Modeling Languages," in *1st International Workshop on Executable Modeling (EXE'15)*, 2015.

[21] E. Fox Keller and D. Harel, "Beyond the gene," *PLoS ONE*, vol. 2, no. 11, pp. 1–11, 11 2007. [Online]. Available: http://dx.plos.org/10.1371%2Fjournal.pone.0001231