LSC Language Reference Manual

Assaf Marron and Smadar Szekely

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science

April 2014
(last update April 2015)

Acknowledgements

This work was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant and a Proof of Concept Grant to David Harel from the European Research Council (ERC) under the European Community's FP7 Programme, and by the Israeli Science Foundation.

1. Contents

How to use this book 5
Basic Concepts 6
The LSC Language 6
LSC 8
An LSC Example 9
LSC Specification 10
Play-Out 10
Strategies 11
Scenarios to Aspects (S2A) 12
Entities and Attributes 13
System 13
System Model 13
Class 13
Object 14
Property 16
Method 16
Lifeline 17
Dynamic Lifeline 18
Event 20
Message 20
Execute Message 22
Monitored Message 22
Visible Event 22
Hidden Event 22

Variable 23
Assignment 24
Condition 25
Wait Condition 26
Sync 26
Exit 27
Forbid 27
Subchart 28
Alternatives 29
Loop 30
LSC Reference 31
Violating Event 32
Anti-scenario 32
User 32
Environment object 33
Clock 33
Expression 34
Flow 36
Steps and Supersteps 36
Location 36
Cut 37
Global Cut 38
Current Location of a Lifeline 39
Synchronization Point 39
Event Ordering in LSC Execution 40
Enabled Event 41

Message Matching 41
Unification 42
Minimal Event 42
Active LSC 43
Live Copy of an LSC 43
Violation of an LSC 44
Completion of an LSC 46
Transition 47
References 48

How to use this book

This document lists and defines concepts and constructs in the LSC visual language in its UML-based variant and its related execution mechanism (S2A).

The order of entities in the reference is an attempt to enable linear reading. A concept that is needed for understanding the definition of other concepts usually precedes them. Forward references also exist, usually indicating further details, refinements and exceptions.

The terms in the reference are organized in groups:

- Basic Concepts
- Entities and Attributes
- Flow

For each term in the reference, there may be several sections:

- Synonymous Terms: Other names by which this entity may be known or referenced in LSC texts.
- Visual Notation: A textual description of the notation. In some cases the text is accompanied by a picture.
- Semantics: Explanation of the meaning of the notation.
- Notes and Variants: This section highlights specific issues and relevant points. In some cases it includes notes about the notation in the context of the PlayGo development tool.

Basic Concepts

The LSC Language

Behavioral programming (BP), also termed scenario-based programming, is a recently proposed software engineering approach that can be used in specifying, designing, programming and verifying complex reactive systems. BP originated from the joint work of Werner Damm and David Harel [1] on the language of *live sequence charts*, called hereafter LSC, and continued with the introduction of the Play-Engine execution environment with its play-in and play-out facilities [3]. Additional research and development produced tools, language enhancements, natural language support, analysis methodologies, execution algorithms, and theoretical examinations of the approach. More details about can be found in the behavioral programming website (www.b-prog.org) and in the publication list on David Harel's website.

A specification in the LSC language consists of charts, called live sequence charts (LSCs).

Each of the charts specifies a scenario of desired and/or undesired behavior of the system.

The specification is executable. The scenarios in the charts are executed simultaneously and are interwoven, yielding integrated system behavior. Depending on the tools used, the interweaving can be done at run time, or during compilation, or as part of an analysis algorithm.

Specifically, each LSC specifies events and event sequences, which, from its own point of view must, may, or must not occur under certain conditions and following certain (other) sequences of events.

The main advantages of programming in LSC include:

- Scenario Visualization: Depicting scenarios in easy-to-understand diagrams.
- Play-out: A key element of scenario-based programming is that the specification is executable. The execution infrastructure runs all scenarios simultaneously. It repeatedly synchronizes all scenarios, and selects and triggers an event that is proposed for execution by at least one chart and is not forbidden by any chart.
- Play-in: Another key concept of LSC is GUI-based programming, called play-in. In an LSC development environment (e.g., the Play-Engine or PlayGo), in addition to menu driven drawing and specification capabilities, the user can enter events by actually causing them to happen in a GUI application that represents the external interfaces of the final system. Thus the developer can specify events by pressing buttons, turning on lights, entering text in display fields, or manipulating other objects in a mockup of the system being developed. Additionally, the user can enter scenarios by specifying them in controlled natural language (i.e., Natural Language Play-in).
- Multi-Modality: In LSC, execution elements are associated with multiple modalities, expressing, as do modal verbs in natural language, what must happen, what may happen, and what may not happen. The modalities supported by LSC are:

- *must* vs. *may*: Using the terms *hot* and *cold* (collectively *temperature*), one specifies which events and conditions must happen or must be true at a given moment, and which are possible, but are not mandatory.
- *allowed* vs. *forbidden*: In LSC one can specify a forbidden sequence of events, thus affecting the execution of multiple scenarios with little or no explicit inter-scenario communication.
- *execution* vs. *monitoring*: The LSC specification distinguishes between specification of an event whose triggering is requested, and specification of an event whose triggering is monitored, i.e., waited-for.

The two variants of LSC Language: The Play-Engine variant and PlayGo UML-based variant

As stated above, the LSC language was first implemented in the Play-Engine tool described in detail in the book *Come Let's Play*, by Harel and Marelly [3]. Subsequently a UML compliant version was developed and implemented in the PlayGo tool [2] based on scenario-to-aspects compiler and execution mechanism described in detail in [4].

The main semantic difference between the two variants of the language is how events driven by the environment or other charts are monitored. In the Play-Engine, as in the original formal definition of the LSC language, every chart is divided into a prechart and a main chart. After the scenario specified in the prechart occurs – as driven by other charts or the environment, the main chart is enabled, and it can react to the scenario in the prechart by executing additional events. By contrast, in the UML variant, each event is tagged separately as *monitor* or *execute*. Enabled events marked as *monitor* are only waited for, while events marked as *execute* are considered by the system when selecting the next event to be triggered.

A second important difference is the support in the Play-Engine of explicit event forbidding, associated with specific scopes of when the forbidding is in force. In PlayGo this is implemented indirectly, by forcing a (hot) violation at the end of an undesired scenario causing the execution engine to avoid completing such scenarios.

A few other small differences exist, in terms of level of support of various language features. Of these we note here the fact that the Play-Engine supports both synchronous and asynchronous messages, while PlayGo supports only synchronous ones.

The infrastructure of the two tools is very different, with the Play-Engine being written in Visual Basic, and PlayGo in Java.

This document lists and defines concepts and constructs only in the UML-based variant of the LSC, as implemented in PlayGo.

LSC

Synonymous Terms

Live Sequence Chart; Chart, Scenario.
Note: The term LSC is used both for the name of the language and for a single live sequence chart.

Semantics

A formal specification of a certain aspect (in the general sense), or part, of a system behavior. An LSC often describes a part of system behavior that is (or can be) described in a single sentence or a single paragraph in a requirements document.

The term scenario may be used both as a synonym for LSC, and for a more abstract or conceptual sequence of events, which may be encoded by multiple LSCs.

Notes and Variants

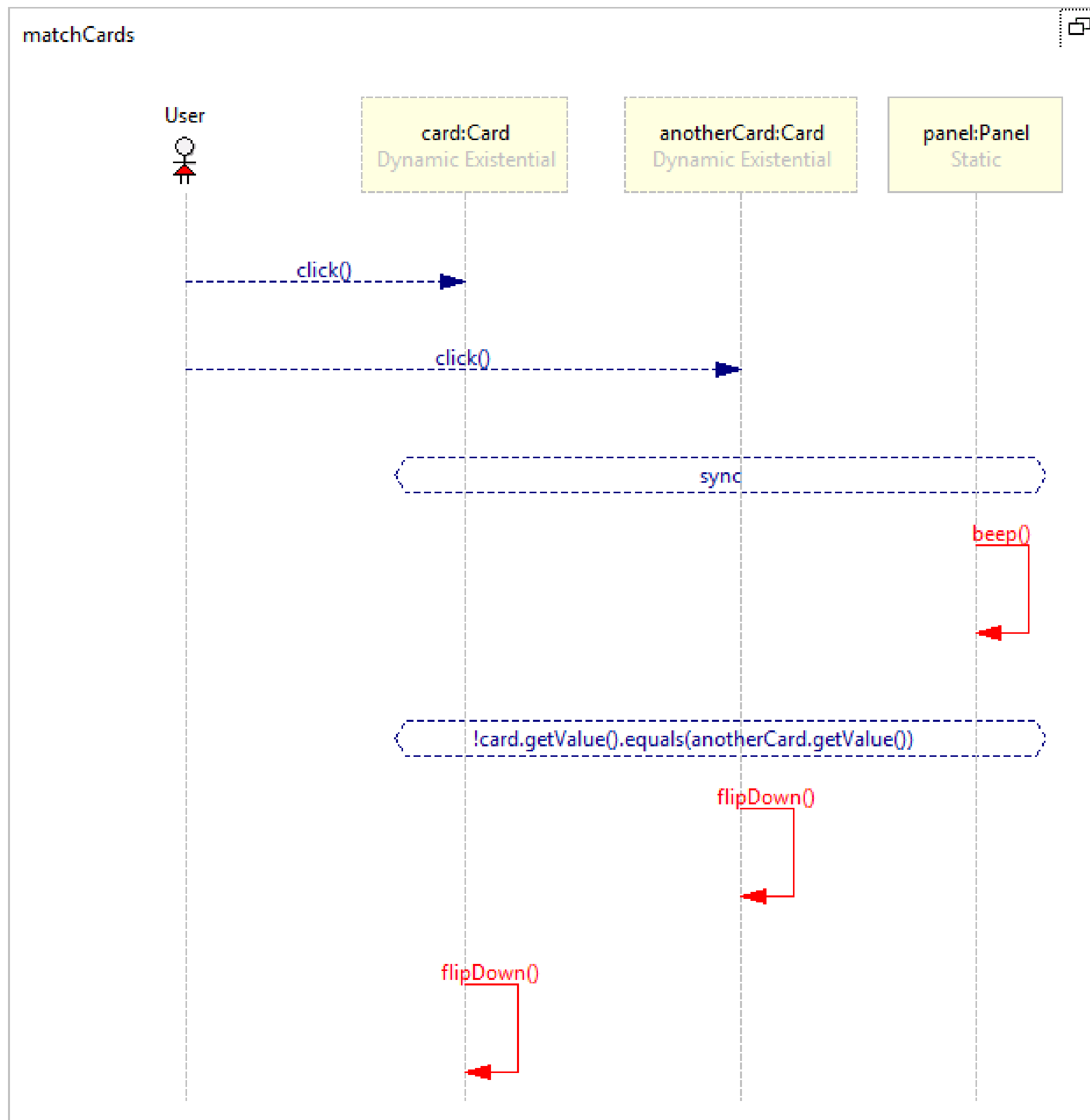
An LSC is associated with a unique name (an identifier) that distinguishes it from other LSCs.

An LSC Example

The LSC below describes one part of a memory game. The LSC includes 4 lifelines labeled **`User`**, **`card:Card`**, **`anotherCard:Card`** and **`panel:Panel`**. It starts by monitoring two **`click`** events on two cards by the user. It then *beeps* and compares the value of the two cards. If they do not match, they are both flipped down by the system. Otherwise, if the condition is false (i.e., the cards match), the LSC terminates. Other parts of the game, such as recording success, are not handled by this LSC.

Note:

- The red and blue elements in the diagram indicate hot and cold LSC messages and conditions, respectively.
- The solid and dashed arrow lines indicate execute and monitor LSC messages, respectively.
- In the GUI application the cards are displayed face down. The method **`click`** displays the face of the card and the method **`flipDown`** displays the back of the card again.



LSC Specification

An LSC specification is a finite set of live sequence charts.

Play-Out

Semantics

Play-out is the basic process by which an LSC specification is executed. The execution consists of discrete steps. In each step one enabled event is selected and all LSCs are advanced accordingly. Event selection is based on a strategy; e.g., naïve, random, synthesis, etc.

Strategies

The core of the play-out process is a strategy mechanism that is responsible for choosing the next message to execute among all enabled messages, after all hidden events were executed. The choice is based on the specification and on the current state of the system.

PlayGo allows the definition, implementation, and use of new, user-defined play-out strategies. It also includes several built-in, pre-defined play-out strategies, which we now describe.

Naïve play-out

The *naïve play-out* strategy is the simplest one. It arbitrarily chooses a non-violating message from among the set of messages that are currently enabled for execution in at least one chart, and which will not hot-violate in any chart.

Random play-out

The random play-out strategy is similar to the naïve play-out strategy. However, it chooses the next message to execute randomly, using a 'seed' number. The user can either choose a constant seed, in which case the same message will be selected at the same point in execution in repeated runs, or ask PlayGo to use a random seed, thus causing different, random message selections in different runs.

Interactive play-out

The interactive play-out strategy allows the user to choose the next message to execute. The user is presented with a dialog that lists all the currently enabled non-violating messages and is expected to select one of them. The selected message is returned to the play-out mechanism for execution.

Smart play-out

Smart play-out is a smarter, safer way of choosing the next message to execute. It considers not only the current set of enabled non-violating messages, but also looks ahead and picks a finite sequence of messages that will lead to a successful (non-violating) superstep, if such a sequence exists.

To compute a safe superstep, PlayGo uses model-checking techniques. This is done by challenging the model checker to prove that there is no superstep that can take place without violating the LSC specification. If no such superstep exists, there is no way to proceed. But if there is such a superstep, the model checker provides it as a counter example. PlayGo uses the counter example to guide the execution.

Smart play-out is implemented in PlayGo using JTLV [5]

Synthesis

Looking one superstep ahead, as is done in the smart play-out strategy, may not be enough, because a safe superstep does not guarantee future execution free of deadlocks. An adverse environment may be able to cause system violation. Thus, PlayGo offers another strategy, based on the technology of controller synthesis, which attempts to compute the LSC specification from a controller whose behavior is guaranteed to satisfy the specification. (For the case where the LSC specification is unrealizable and a controller cannot be synthesized, PlayGo offers a debugging mode termed counter play-out). The computed controller is then used to guide the play-out mechanism.

The variant of synthesis used in PlayGo supports environment assumptions.

Scenarios to Aspects (S2A)

S2A (= "Scenarios to Aspects") is the LSC Compiler implemented in the PlayGo tool. It translates LSCs into AspectJ and Java code, and thus provides full code generation of reactive behavior from visual inter-object scenario-based specifications. The event selection is based on the selected strategy.

Entities and Attributes

System

Visual Notation

None

Semantics

The system is a collection of classes and objects, together with an LSC specification, with optional graphical and other external interfaces.

The system has behavior that is defined and driven by the LSC specification according to the strategy chosen by the user.

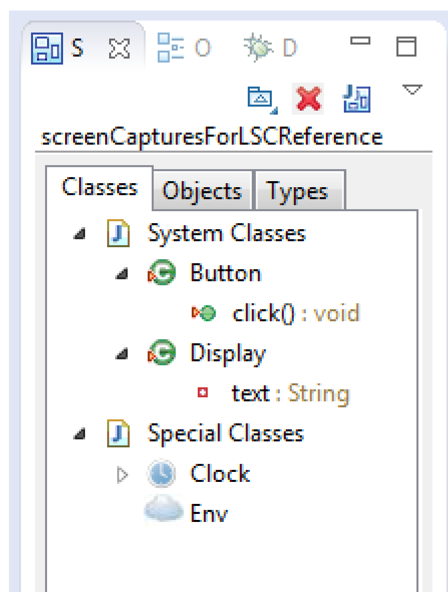
System Model

The system model is the collection of definitions of classes and objects that participate in its LSCs.

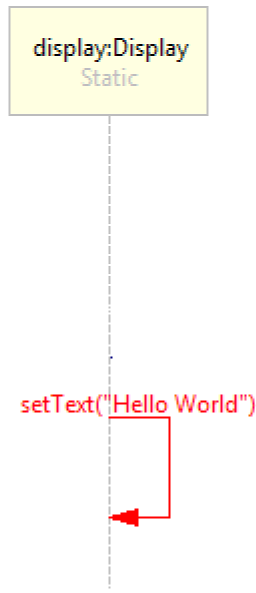
Class

Visual Notation

A class can be viewed from the system model view.



Classes appear in charts as part of the lifeline label, for example the lifeline below represents an object named display of class Display.



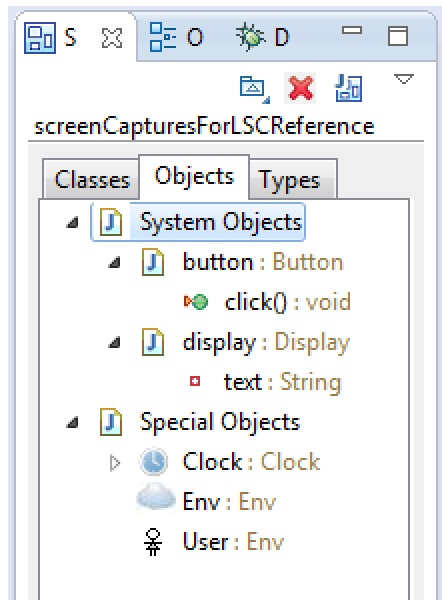
Semantics

A class is a named collection of methods and properties. Each object is associated with a class.

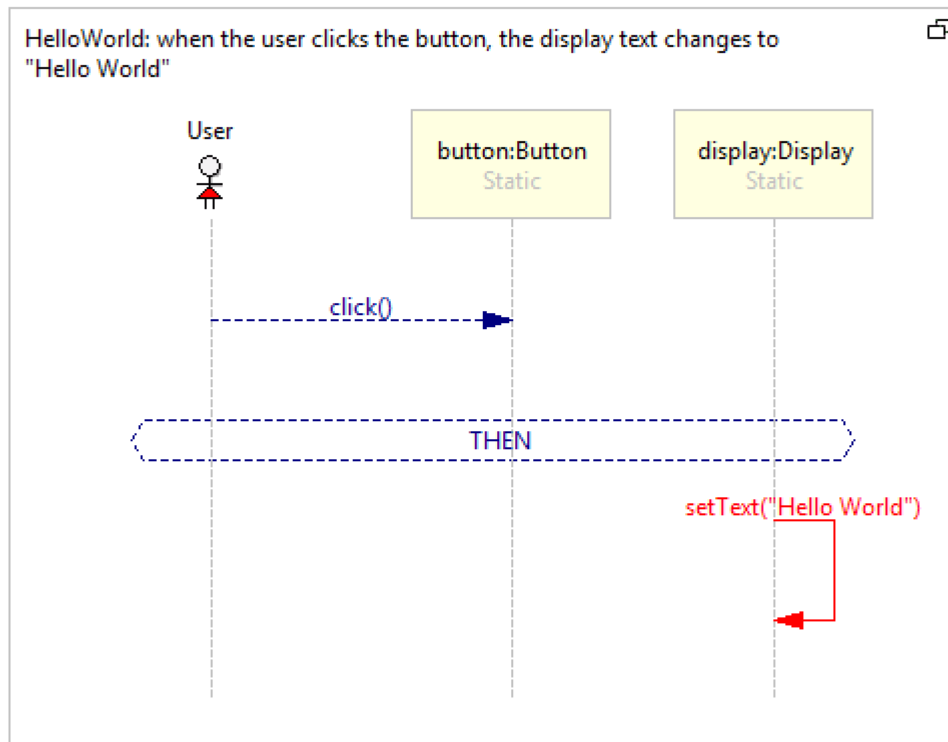
Object

Visual Notation

An object can be viewed from the system model view.



The object behavior can be viewed in charts that have lifelines labeled by this object.



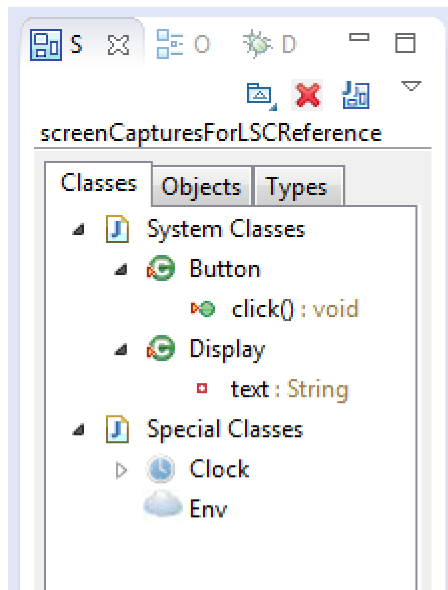
Semantics

An object is an entity (in the system model) that can participate in a scenario, can be represented by a lifeline, can send and receive events, and can participate in conditions and other LSC constructs. An object is associated with a class. Calls to an object's methods or the setting of its property values are represented as messages sent between objects. Object properties and methods can be used in conditions.

Property

Visual Notation

Properties can be viewed from the system model view.



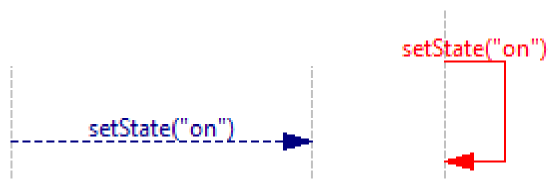
Semantics

Each class may define a set of properties. A property is associated with a range of values. Property values can be changed as a result of messages received by the object. The value of a property can be set or examined by the application, using its auto generated set and get methods or from the Java code of methods.
Properties are associated with property types, such as: Integer, String, Boolean or Enum.

Method

Visual Notation

In LSC, method calls appear as labels of messages. Such a label includes the method name followed by parameters in parentheses.



In PlayGo, Methods can be viewed from the system model view.

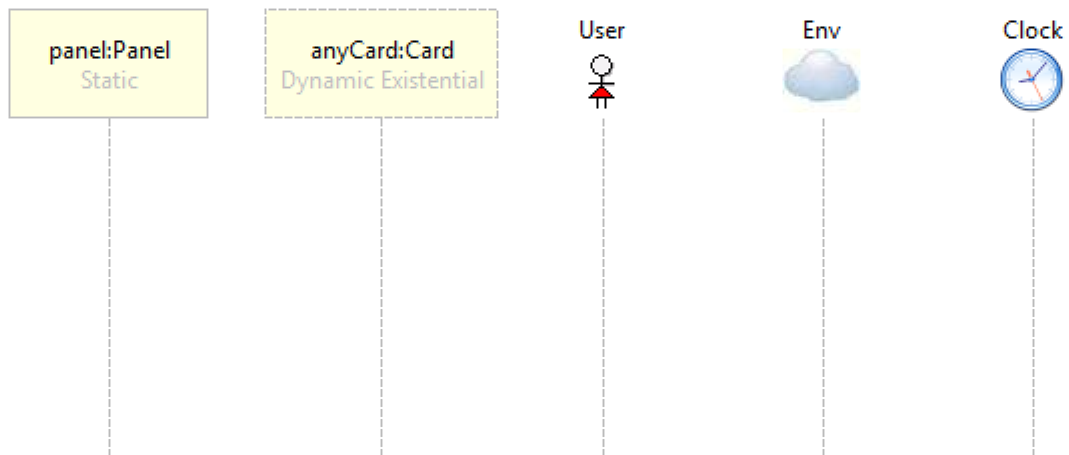
Semantics

Each class may define an arbitrary number of methods. A method is invoked when a message labeled by it is triggered. Every method has a name and optionally a set of parameters.

Lifeline

Visual Notation

A vertical line with a rectangular label on top (header) with the object name and optionally its class formatted as <object name>:<class name>.



Semantics

A lifeline is a representation (appearance) of one or more objects of the system.

A lifeline has locations associated with other LSC constructs. During a run, the current location advances along the lifeline, generally from top to bottom.

A lifeline can be static or dynamic:

- Static Lifeline (aka concrete lifeline):
  - A static lifeline represents a single object in one LSC. Other lifelines representing the same object may appear in other LSCs.
  - Only one static lifeline can appear in a single LSC for a given object.
  - In a static lifeline, during play-out, binding between the lifeline and the concrete object it represents is fixed, and may not change between LSC instances. A static lifeline is bound as soon as the LSC starts.
- Dynamic Lifeline (aka symbolic lifeline):
  - A dynamic lifeline is a place holder for an as-of-yet-unspecified object or objects from a given class.
  - A single LSC may contain more than one dynamic lifeline for a given class.

  - An existential dynamic lifeline represents one object. This is used, e.g., when specifying behavior such as “when any button is pressed, the light associated with this button is turned to red”.

  - A universal dynamic lifeline represents all the objects of the given class. This is used, e.g., when specifying behavior such as “when the controller issues message ‘abort’, all the lights are turned off”.
    Note: Universal dynamic lifelines are not supported in PlayGo at this time.

- A lifeline may be of one of the following types:

  - System – represents a system object.

  - Environment – represents an external/environmental module or subsystem.

  - User – represents the user.

  - Clock – represents the clock.

A dynamic lifeline has a polymorphic scope Deep or Shallow, to support scenario-based behavioral subtyping mechanism.

By default the polymorphic scope is defined to Deep. A dynamic lifeline tagged with deep scope represents objects instantiated from the lifeline type or any of its subtypes. While a lifeline tagged with shallow polymorphic scope represents objects instantiated from the lifeline's type, but not to objects instantiated from its subtypes.

See also Dynamic Lifeline .

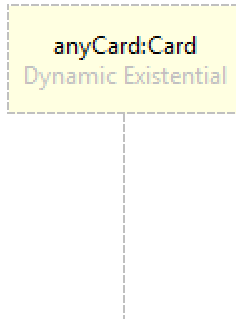
Dynamic Lifeline

Synonymous Terms

Symbolic lifeline

Visual Notation

A lifeline with a dashed border; at the lifeline label the first line contains a lifeline name, followed by a colon and a class name, the second line reads “Dynamic Existential”, and the last line optionally contains a Java expression (serves as the binding expression).



Semantics

A dynamic lifeline is a placeholder in an LSC for a concrete object, which will be bound at runtime.
The optional binding expression (in PlayGo this is a Java expression) controls how the concrete object will be selected from the entire set of objects in this class. It is expected to return/evaluate to an object of the class of that lifeline.

- When there is no binding expression and the first event on the lifeline is a monitored event, when the event occurs on any object of the class the lifeline is bound to that object.
- When there is no binding expression and the first event on the lifeline is an executed event, when the event is supposed to occur an exception (NullPointerException) will occur, since the lifeline is unbound.
- When there is a binding expression and the first event on the lifeline is a monitored event, when the event occurs on any object of the class the lifeline is bound to that object only if that object is equal to the one returned/evaluated by the binding expression.
- When there is a binding expression and the first event on the lifeline is an executed event, the lifeline is bound and the event occurs on the bound object. Binding is done as follows: At each cut change S2A tries to bind the lifeline by evaluating the binding expression. If binding succeeds S2A won't attempt to bind it again, but if binding fails S2A will try to bind it again in the next cut change.

If the lifeline is not yet bound, and an assignment or a condition refers to it, an exception (a NullPointerException) will occur.

Currently in PlayGo, LSC variables cannot participate in the binding expression.

An existential dynamic lifeline represents one object. This is used, e.g., when specifying behavior such as "when any button is pressed, the light associated with this button turns to red".

Note: All dynamic lifelines in PlayGo are existential. Universal dynamic lifelines are not supported at this time.

Variants and Notes

Tool shortcut: As mentioned, PlayGo does not support universal dynamic lifelines at this time. But it does provide ways to achieve the same semantics. To read about the way PlayGo provides for defining LSCs with semantics equivalent to universal dynamic lifelines, refer to: http://www.weizmann.ac.il/mediawiki/harelgroup/index.php/How_to_implement_Universal_Binding.

Event

Event is a collective name for the entities of message and hidden event.

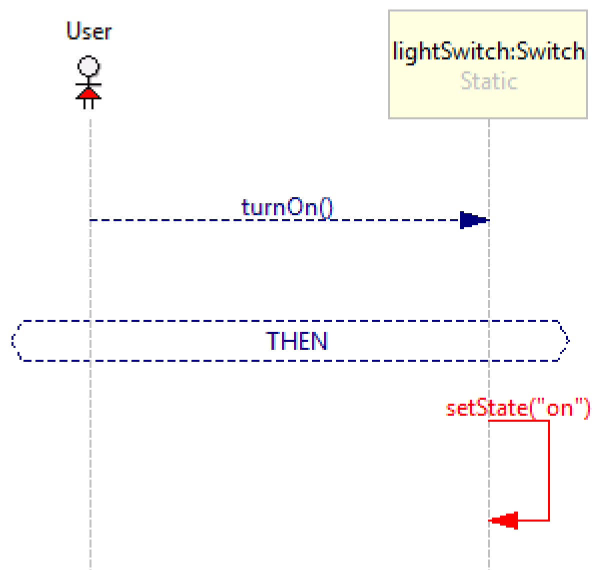
Message

Synonymous Terms

Visible Event

Visual Notation

An Arrow from a source lifeline to a target lifeline, labeled with a method call. The method call is comprised of a method name and optional parameters in parentheses. The method must be a valid method of the class of the target lifeline; i.e., an explicitly defined method or a getter or a setter of class property.



Hot Message: The arrow line is red

Cold Message: The arrow line is blue.

Monitored Message: The arrow line is dashed.

Execute Message: The arrow line is solid.

Semantics

All messages are synchronous; i.e. sending and receiving occur at the same time. In other words, the sending and receiving lifelines advance past the locations associated with the sending of the message and receiving of the message at the same time.

If the sending lifeline is the same as the receiving lifeline, the message is called a "self-message". The sending location is also the receiving location.

A message may have parameters. A parameter is associated explicitly with a type:

- Exact – a constant; e.g., 2, "Hello World".
- Symbolic – a name of a variable; e.g., X, ANY, myCurrentTime
- Opaque expression – a Java expression; e.g., true, false, button.getState(), System.getCurrentTime(), MyDirectionEnum.NORTH

A message is a symbolic message if at least one of its parameters is associated with the type 'Symbolic'.

At execution time, a symbolic message is bound if all its variables are bound. Otherwise it is free.

A symbolic message is considered enabled only when it is bound (i.e., all variables that appear in message arguments are bound).

Once the message is triggered, if the message is of execution type 'Execute', the corresponding method is called. The method may carry out arbitrary actions, including modifying objects of the running application, e.g., it can change properties that can be referred to in conditions. If the message is of execution type 'Monitored', the method is not called (as it was already called either externally or by another message in another LSC, or even called by another method).

If the message appears as 'Execute' in multiple LSCs, and all are enabled, these appearances are unified and the method is called only once.

Cut changes: If the message involves two lifelines, the sending lifeline proceeds past the sending location, and the receiving lifeline proceeds past the receiving location. If the message is a self-message, the lifeline proceeds past the common sending-and-receiving location.

A message can be hot or cold. During play-out this affects the temperature of the cut when the message is enabled.

Notes and Variants

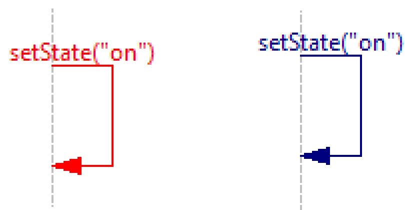
- A message sent by an environment object or by a user object is considered as an external message / external event. All other events are system events.

- A message sent by an environment object or by a user object cannot be of execution type 'Execute' but only 'Monitored'.

- In the Play-Engine there is a message of type 'property change'. In PlayGo, there is no such distinction and property changes are merely calls to setters and getters.
- Activities such as clicking a button on a GUI are represented as messages from a user lifeline to an object lifeline.
- When working with synthesis, a method should declare the property it changes, and the type of change that it performs on the property (i.e., sets its value, increases its value or decreases it).

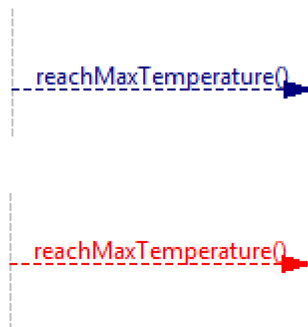
Execute Message

A message of execution type 'Execute'. The message can be either hot (in red) or cold (in blue).



Monitored Message

A message of execution type 'Monitored'. The message can be either hot (in red) or cold (in blue).



Visible Event

A synonym for message.

Hidden Event

Visual Notation

See visuals for specific constructs.

Semantics

Hidden events include:

- Exiting an LSC.
- Entering or exiting a subchart.
- An assignment.
- A condition.
- Entering, leaving or skipping a loop.
- Binding a dynamic lifeline with a concrete object.

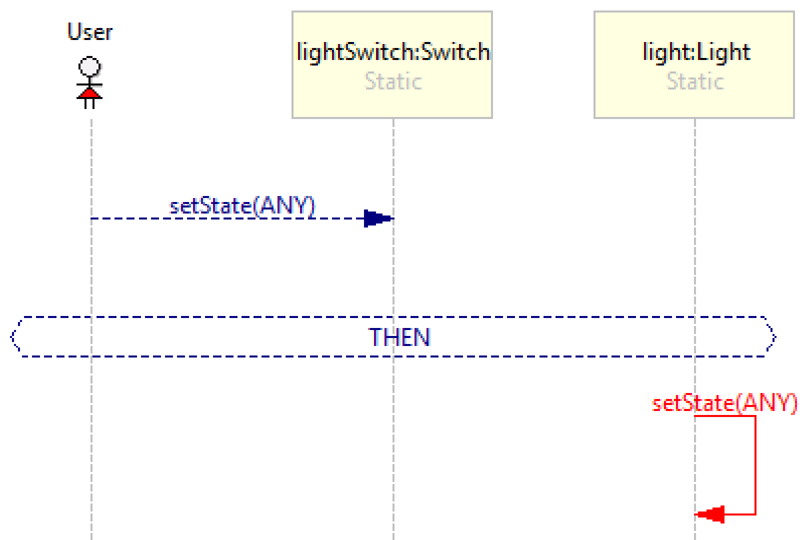
These events do not involve the exchange of messages.

See detailed semantics for each type of event.

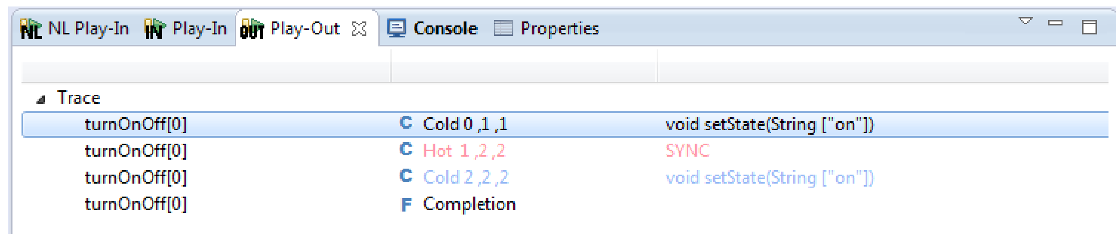
Variable

Visual Notation

The name of the variable, such as "X5", or "Altitude", or “ANY”, appears as a string in assignments, messages, conditions, and other constructs.



In PlayGo, the Variable and its value appear during play-out in the play-out view, displays the execution trace.



Semantics

A symbol that may appear in constructs, and which may assume different values at different times according to execution rules and unification processes.

Scope: A variable is local to an LSC, and therefore can be referred to only within the LSC in which it is defined. If multiple live copies of a given LSC are active, each has its own set of variables. All occurrences of a given variable name in a given activation of a given LSC refer to one variable (i.e., no separate scope for subcharts).

Each variable is associated with a type, which specifies its possible values.

A variable may be bound or free.

- We say that a variable is bound if and only if the symbolic name is associated with a value. Otherwise we say that the variable is free

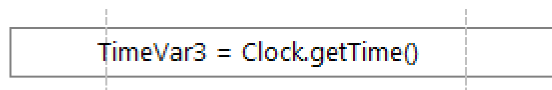
The initial state of a variable in an LSC is free, and it becomes bound as a result of one of the following:

- Execution of an assignment statement.
- Unification, in case the variable is used as a message argument.

Assignment

Visual Notation

A rectangle, containing a variable name on the left, an assignment symbol “=” in the middle, and an expression on the right hand side.



Semantics

An assignment stores a new value in a variable. The new value is associated explicitly with a type:

- Exact – a constant; e.g., 2, “Hello World”.

- Symbolic – a name of a variable; e.g., X, ANY, myCurrentTime
- Opaque expression – a Java expression; e.g., true, false, button.getState(), System.getCurrentTime(), MyDirectionEnum.NORTH

The locations where multiple lifelines intersect with an assignment constitute a synchronization point.
Participating instances are listed in the 'Covered' field of the assignment.

An assignment is executed as soon as possible after the previous event is executed (and then we say that the assignment is enabled).This occurs when all lifelines that are synchronized with it have reached it, and when all the variables in the right hand side are bound.

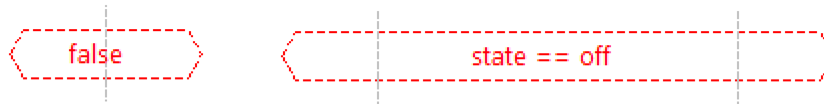
Condition

Synonymous Terms

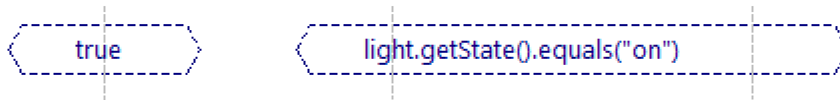
Assertion, condition without wait

Visual Notation

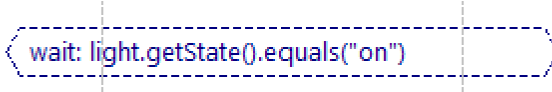
Hot condition: red dashed hexagon containing an expression as described below.



Cold condition: blue dashed hexagon containing an expression as described below.



Timing constraint: a hot or cold condition with the prefix 'wait' in the contained expression



Semantics

A condition contains an expression that is evaluated at runtime as soon as possible after the previous event is executed (and we then say that the condition is enabled).

A condition may be hot or cold.

The locations where multiple lifelines intersect with a condition construct constitute a synchronization point.

Execution Rules - Hot condition without wait:

- Covered objects are synchronized.
- Participating variables must be bound.

- The condition is evaluated once.
    - If it is true - all synchronized instances proceed in the current chart.
    - If it is false – the specification is violated (hot violation).

Execution Rules – Cold condition without wait:

- Covered objects are synchronized.
- Participating variables must be bound.
- The condition is evaluated once.
    - If it is true - all synchronized instances proceed in the current chart.
    - If it is false - the current LSC is violated (cold violation) and is exited.

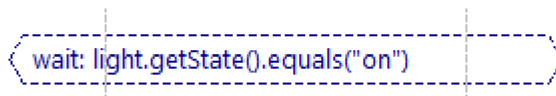
Notes and Variants

- An LSC cannot begin with a condition. In other words, a condition cannot be a minimal event.
- Conditions that are opaque Java expressions cannot participate in synthesis play-out. The reason is that the synthesis algorithm cannot readily analyze the possible values that can be returned by the opaque Java expression.

Wait Condition

Visual Notation

Same as condition, but with the word 'wait' preceding the condition expression.



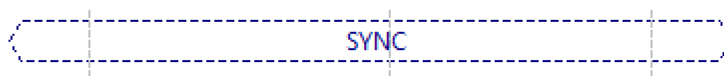
Semantics

Same as condition, but with the exception that instead of being evaluated once, the condition expression is evaluated at every clock tick and after every cut change, until it becomes true. When true - the next location in each synchronized lifeline is enabled.

Sync

Visual Notation

Blue dashed hexagon containing the text 'SYNC'.



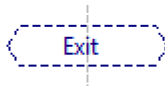
Semantics

The sync construct is equivalent to a cold condition that is always evaluated to 'true' and is used to enforce event order among lifelines.

Exit

Visual Notation

Blue dashed hexagon containing the text 'Exit'.



Semantics

The exit construct is equivalent to a cold condition that is always evaluated to 'false'. It is used to cause exiting the chart (LSC or subchart) that contains this exit (cold violation).

Forbid

Visual Notation

Red dashed hexagon containing the text 'Forbid'.

Semantics

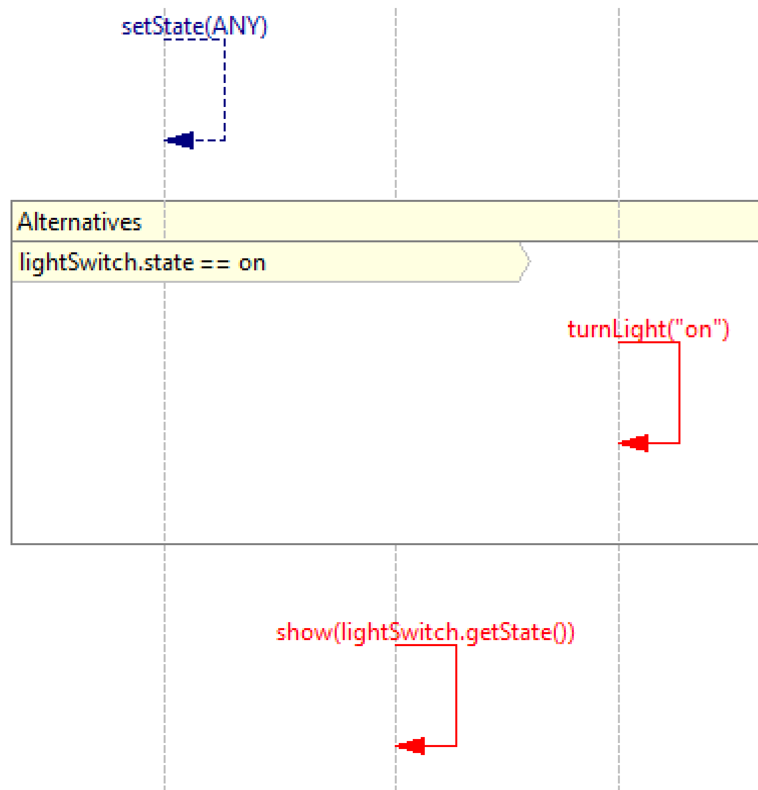
The forbid construct is equivalent to a hot condition that is always evaluated to 'false'. It is used to block/prevent/forbid the execution of the scenario that precedes the condition; i.e., when during execution a message that is followed by a forbid construct becomes enabled it will not be triggered (unless the sender is an environment object).

This is a technique/design pattern for defining an anti-scenario.

Subchart

Visual Notation

A black, solid rectangle. Lifelines may be visible in it, or may be hidden "behind" it.



Semantics

The subchart provides synchronization points for all its participating lifelines (lines visible through it). They all enter it and exit it together.

Lifelines that are hidden "behind" the subchart do not participate in it.

Notes and Variants

1. A subchart construct is generated only implicitly, as a result of defining either loop or alternatives constructs.

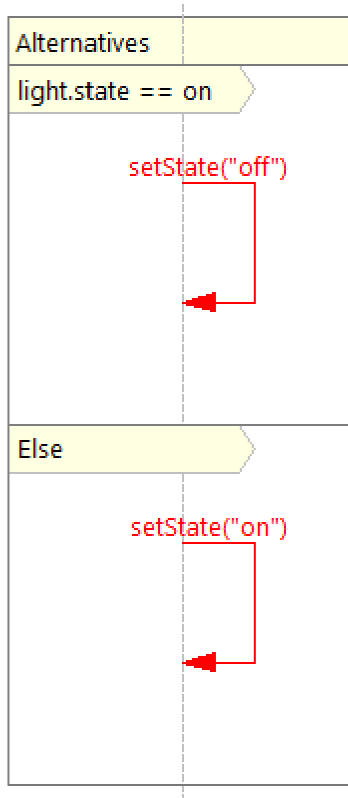
2. A subchart may contain all LSC constructs including, messages, conditions, assignments and other subcharts.

3. A subchart has to include all the lifelines that participate in its contained constructs; e.g., the source and target lifelines of a message in the subchart and all participating lifelines of a condition or assignment contained in the subchart.

Alternatives

Visual Notation

One or more subcharts, each of which has a condition hexagon inside (at the top left).



Semantics

A branching construct.

The locations where multiple lifelines enter or exit each of the subcharts constitute a synchronization point.
The conditions in the subcharts are evaluated one at a time, according to the order specified. Each one, in order, is evaluated as soon as possible (that is, when all participating lifelines are synchronized with it). At this point all variables and participating lifelines are assumed to be bound. If one of them is not bound, the execution stops.

- If the condition is true, the subchart is entered. When the subchart is exited all remaining subcharts of this alternatives construct are skipped.
- If the condition is false, the subchart is skipped, and the processing proceeds to evaluating the condition in the next subchart.

Notes and Variants

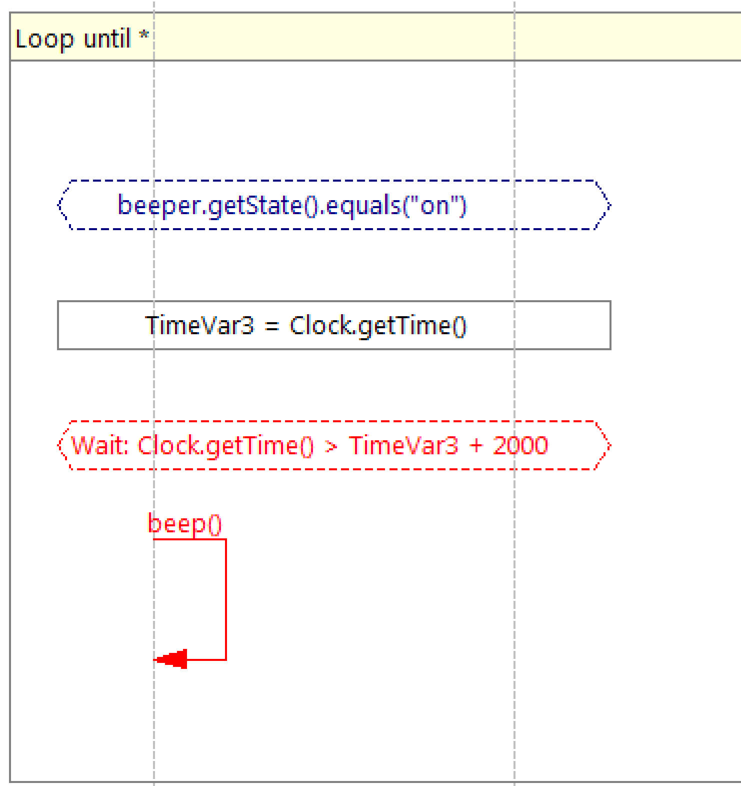
1. The alternatives construct can be used to create an if-then-else behavior, using subcharts, by placing a condition in the first, and 'true' in the second.

2. The condition of a subchart of an alternatives construct cannot be a wait condition.

Loop

Visual Notation

Subchart with loop control mark on left top corner ('*' or some natural number n).



Semantics

A construct that controls iterative execution. It is comprised of the loop control and the loop body. The loop control is either a natural number or a '*'.
Execution Rules:

- Participating objects are synchronized.
- If the loop control is a natural number, the loop repeats no more than that number.
- If the loop control is '*', the loop is unbounded: that is, the loop is carried out indefinitely.

- At the end of each iteration all objects are synchronized as in the ordinary end of a subchart.
- The loop exits when either a cold false condition forces an exit from the current subchart, or the specified finite number of iterations is exhausted.
- The loop body is executed repeatedly.

LSC Reference

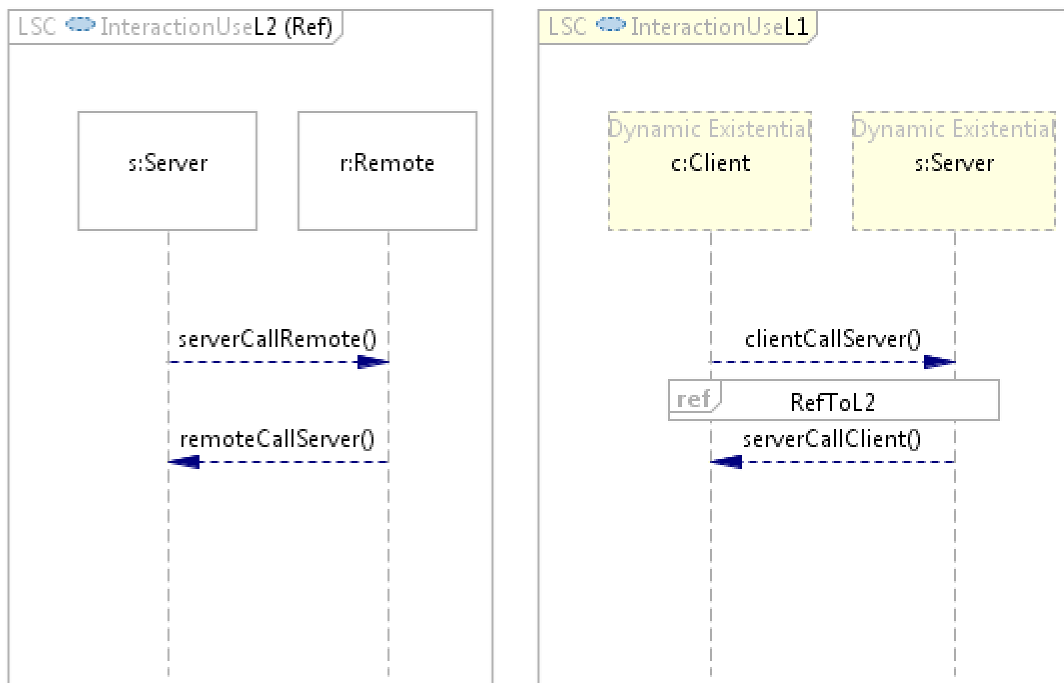
Synonymous Terms

Interaction Use

Visual Notation

The definition of an LSC Reference looks just like an LSC except that labels of lifelines, charts and subchart are not colored (instead they are displayed in black and white).

The use of an LSC reference is shown inside the LSC where the reference is made, as a black rectangle with the name of the LSC Reference.



Semantics

An LSC Reference is a macro for a subchart. At compile time, the contents of the LSC reference is used as part of the LSC, as if it were coded originally therein.

The LSC reference forms a subchart in the LSC where it is referenced.

Lifeline matching is done according to lifeline names. When the reference scenario contains lifelines that are not in the referencing scenario, these lifelines are added to the referencing scenario. All lifelines that appear in the LSC reference are covered by it.

An LSC reference may be referenced from more than one scenario in the specification, and may contain LSC references to other references. However, recursive referencing is not allowed.

Violating Event

Visual Notation

None.

Semantics

We say that an event is a violating event with respect to a particular cut if the occurrence (execution) of the event in this cut will cause a violation. See Violation of an LSC.

Anti-scenario

Semantics

A forbidden scenario, that is, a scenario that is not allowed to happen. An anti-scenario can be specified by placing a Forbid construct at the end of the forbidden sequence of events.

Play-out attempts to avoid the execution of anti-scenarios (see details in Strategies). If the execution of such scenario cannot be avoided (e.g., it is forced by external events) - a violation occurs.

User

Visual Notation

A lifeline with a small icon of a person.

Semantics

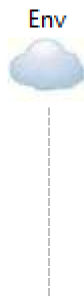
A predefined object. Events with the user object as the sender are triggered during play-in and play-out, when the controls of the GUI application are manipulated, or when events are played in or out from the system model without explicitly specifying a calling object.

The class of the user object is Env (see environment object).

Environment object

Visual Notation

A lifeline with a small icon of a cloud.



Semantics

A predefined object of class Env, representing the environment (real world) outside of the system. It can send and receive messages/events.

The predefined class Env has no properties or methods. These can be added by the user as required, based on the specification. Objects of class Env are treated differently in synthesis and smart play-out, namely the strategy cannot control the behavior of such objects, but some strategies (e.g., strong synthesis) may consider assumptions about their behavior.

Clock

Visual Notation

A lifeline with a small icon of a clock.

Semantics

A predefined object capturing the passage of time in the application.

Among other things, the clock has the methods **`start`**, **`tick`** and **`getTime`**, and a property **`time`**.

Once the **`start`** method is executed (either automatically by PlayGo or explicitly by the specification), the **`tick`** method is executed by play-out at fixed intervals.

The value of the **`time`** property of the clock (returned by **`getTime`**) is incremented with every tick.

Expression

Visual Notation

See context where the expression is used (condition or alternatives).

Semantics

An expression has a value at runtime. An expression may be either a Java expression or a synthesis compliant expression.

- Java expression – a snippet of Java code that is evaluated to a boolean value. An example is button.getState().equals(“on”). Also the constant values **`true`** and **`false`** are allowed.

- Synthesis compliant expression - a structured expression in a format with which synthesis is familiar and which it can analyze at runtime. This expression can be an and-or combination of expressions of the following types:

    1. Comparison of lifeline property and a constant value:

       PropertyValue, <lifeline name>, <property name>, <operator>, <constant value>, <logical operator>

       Where:

- The constant PropertyValue is the type of comparison (in this case the comparison of a lifeline property with a constant value).
- The <operator> can be one of the following: ==, !=, >, >=, <, <=
- The <logical operator> connects the present expression to the next one and can be && (and) or || (or). When there is no next expression a <logical operator> is still expected.

Example for comparing the property 'state' of the lifeline 'lightSwitch' to the value 'on':

PropertyValue,lightSwitch,state,==,on,&&;

2. Comparison of one lifeline's property with another's:

   PropertyProperty, <lifeline name>, <property name>, <operator>, <lifeline name>,<property name>,<logical operator>

   Where:
   - The constant PropertyProperty is the type of comparison (in this case a comparison of a lifeline's property with another's).
   - The <operator> can be one of the following: ==, !=, >, >=, <, <=
   - The <logical operator> connects the present expression to the next one and can be && (and) or || (or). When there is no next expression a <logical operator> is still expected.

Example for comparing the property 'state' of the lifeline 'lightSwitch' to the property 'state' of the lifeline 'light':

PropertyProperty,lightSwitch,state,==,light,state,&&;

3. A combination of two or more PropertyValue and PropertyProperty expressions. For Example:

   PropertyValue,lightSwitch,state,==,on,&&;
   PropertyProperty,lightSwitch,state,==,light,state,&&;

Flow

Steps and Supersteps

The execution of a message event and all the hidden events that follow it is considered a step.

In the context of execution, some events are driven by the user, while others are triggered by the system. Following a user action, PlayGo can execute a *superstep* – a sequence of steps that terminates when the next event is an environment event or a user action, or when the entire system's execution is terminated.

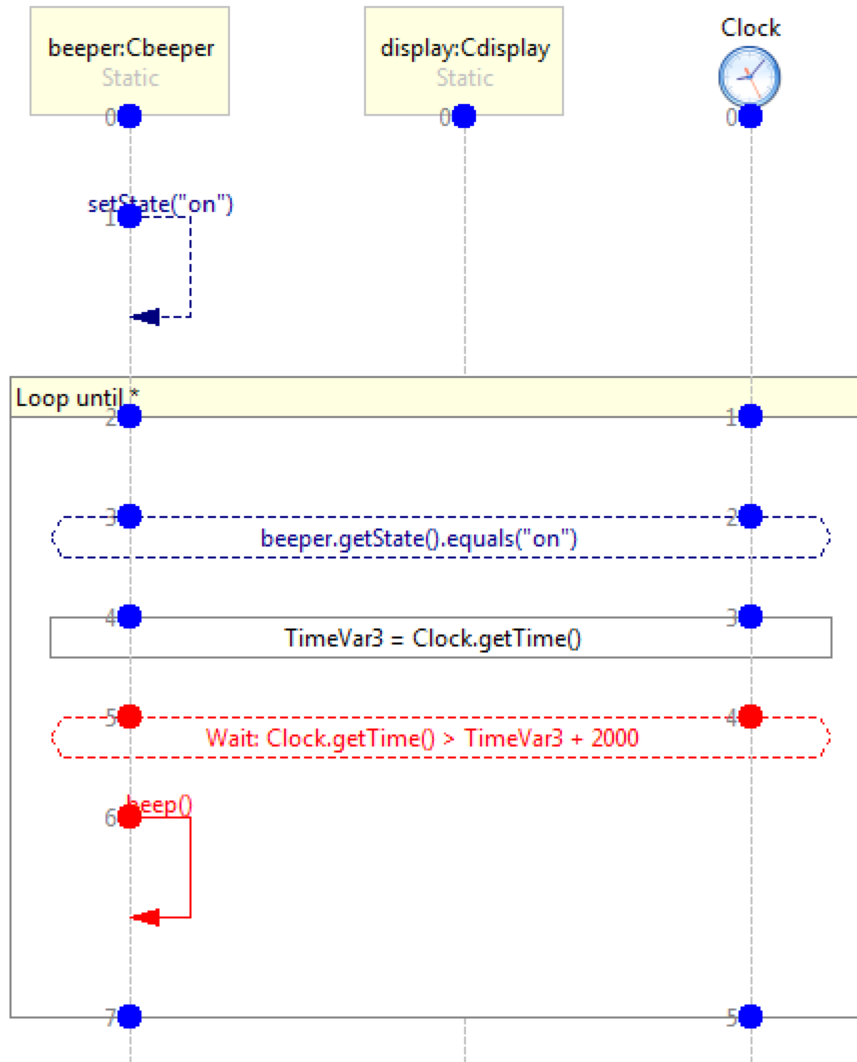
Location

Visual Notation

The intersection of a lifeline with any LSC entity (e.g., message, condition, assignment, subchart…). Optionally marked also with a solid filled circle. The color of the location reflects its temperature.

Hot locations: red filled circles.

Cold locations: blue filled circles.



Semantics

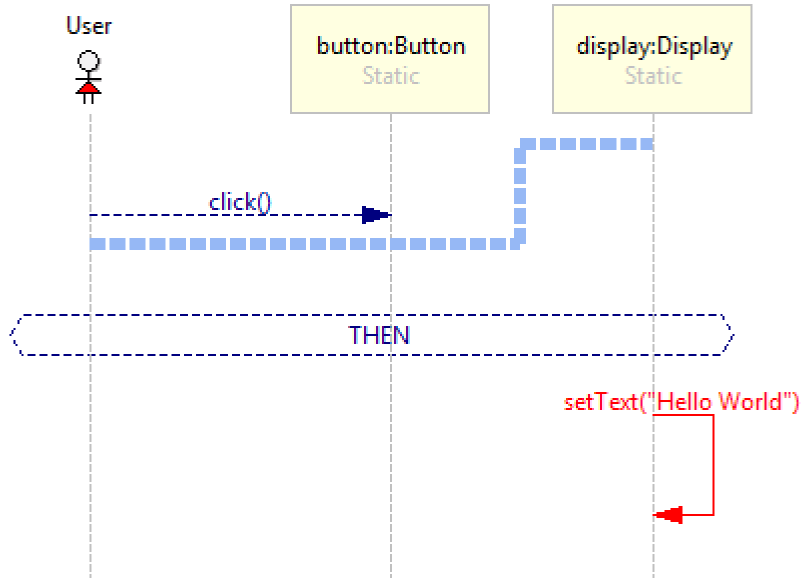
A location identifies the intersection of a lifeline with an LSC entity. It represents a possible state of the lifeline and marks its smallest unit of progression. Each lifeline is associated with one initial location, which appears under the label/header of the lifeline. A transition moves the current location of a lifeline from one location to another.

A location may be hot or cold. The location's temperature is determined by the temperature of the LSC entity that defines it.

Cut

Visual notation

A thick step line, in red or blue, which intersects each lifeline of the LSC once.



Semantics

The term cut refers to a cut in a single LSC.

A cut exists during play-out, and is a mapping of each lifeline in a live copy of an LSC to its current location.

A cut may be hot or cold

- A hot cut is one of which at least one lifeline is in a hot location.
- A cold cut is one that is not hot (all its lifelines are in cold locations).

The system may remain in a cold cut indefinitely. In a given LSC, exiting the LSC from a cold cut does not create a violation.

For a hot cut, each lifeline in which the enabled event is hot must eventually proceed past the current location. It is required not to terminate or exit the chart from that location.

Notes and Variants

During play-out PlayGo displays the active LSCs with their cuts.

Global Cut

Visual notation

Implicit from the visualization of the cuts of all individual LSCs.

Semantics

The global cut of an LSC specification is the set of all cuts of the LSCs therein.

A global cut may be hot or cold.

- A hot global cut is one for which at least one of the cuts is hot.
- A cold global cut is one that is not hot (all its cuts are cold).

Current Location of a Lifeline

Visual Notation

The location below the intersection between a cut and a lifeline.

Semantics

Given a cut, the current location of a lifeline is the one following (i.e., immediately under) the location of the cut.

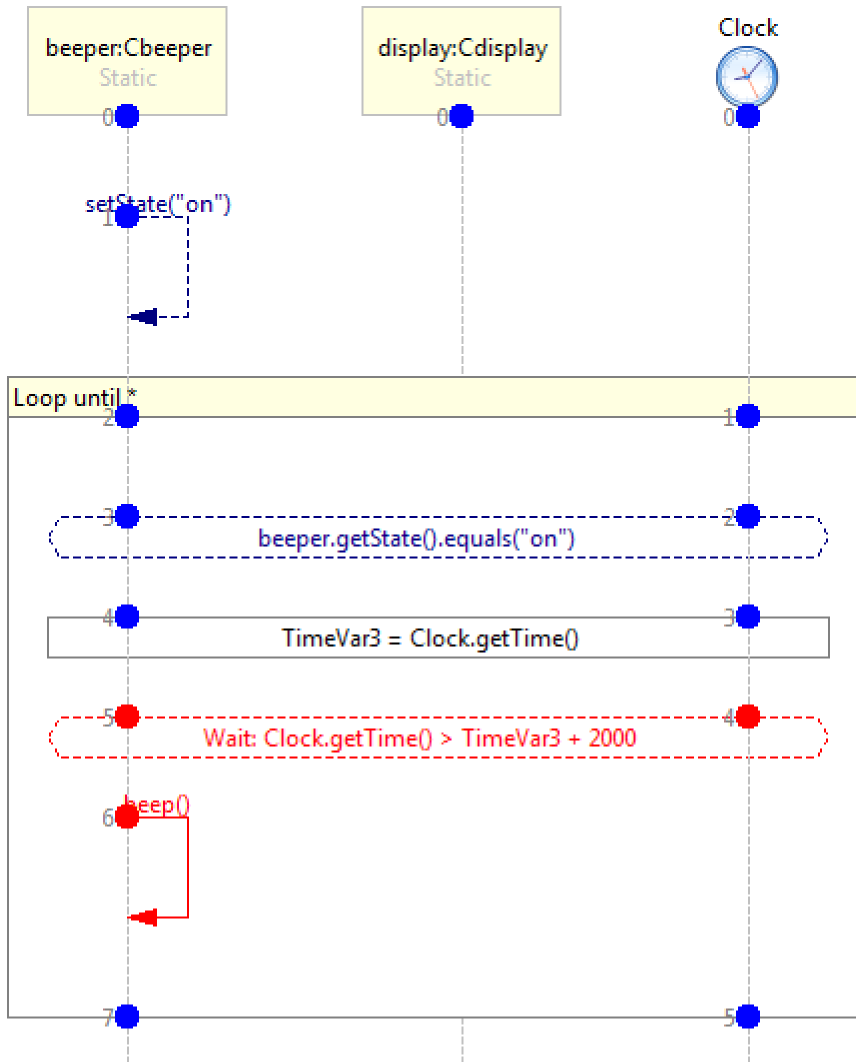
Initially, the current location follows the minimal event that participated in activating the LSC (or is the topmost event if the lifeline in question did not participate in activating the LSC).

The current location changes with transitions.

Synchronization Point

Visual Notation

In the figure below, the loop subchart is a synchronization point of its participating lifelines (beeper and Clock). Correspondingly, each participating lifeline has a location in the first (as well as the last) intersection point with the subchart.



Semantics

A collection of locations from different lifelines that are subjected to the requirement that none of these lifelines can proceed past its location in the synchronization point until all other instances reach their locations in the synchronization point.

Event Ordering in LSC Execution

Semantics

When two locations are on the same lifeline, the higher one precedes the lower one.

When two locations are in the same synchronization point, they are simultaneous. The sending and receiving of a message in a message event are considered to be synchronous.

Enabled Event

Visual Notation

None

Semantics

An event is considered enabled if in all participating lifelines all the event's locations are the current locations and all the prerequisites for triggering it are satisfied.

Notes and Variants

Multiple events can be enabled at a given time.

Conditions and assignments are enabled and executed as soon as their preceding event is executed.

Message Matching

Semantics

When an enabled executed message is considered for execution (this will be referred to as the first message), it is searched for in all LSCs in the specification. As part of the search, details of the messages are compared: sender, receiver, message name and actual message arguments. When messages are successfully matched, we say that they are unifiable. Unifiable messages may or may not be actually unified during a particular run.

Note: the first message must have already its source and target lifelines bound to concrete objects and variables in the arguments (if exists) bound to concrete values.

For a message (this will be referred to as the second message) to be unifiable with an enabled executed message, the following must hold:

- The message's name must be the same.
- If the second message's source (respectively, target) is concrete, it must be the same as that of the first message.
- If the second message's source (respectively, target) is a symbolic lifeline, then both lifelines must be in the same class and the concrete object in the first message cannot be forbidden in the binding expression of the symbolic lifeline of the second message.
    - When the message is triggered, the symbolic lifeline in the second message is bound to the same object as the concrete lifeline in the first message.
- If an argument in the second message is of type 'exact' or 'opaque expression', the argument value must be equal to the corresponding argument value in the first message.
- If an argument in the second message is of type 'symbolic':

- If that argument is bound, then its value must be equal to the corresponding argument value in the first message.
- If that argument is free (i.e., not bound), then no additional comparison is required.
    - When the message is triggered the argument is bound to the same value as the corresponding argument in the first message.

Unification

Semantics

Positive message unification is the successful matching of an enabled executed message (the first message), with a message that is either a minimal event, or is enabled (the second message).

- If the second message is a minimal event, then positive unification activates the chart.
- The cuts of all LSCs in which the unified event is enabled advance simultaneously (i.e., in the same step).

Negative message unification is the successful matching of an enabled executed message (the first message), with a forbidden message (the second message).

In general, for two messages to be negatively unified the same conditions as for positive unification must hold.

- If the second message is non-enabled, then if the first message is an enabled executed message, it is deferred.
- If the first message has already occurred and the second message is a non-enabled message and its containing LSC is in a cold cut, then a cold violation occurs.
- If the first message has already occurred and the second message is a non-enabled message and its containing LSC is in a hot cut, then a hot violation occurs.

In general – successful negative unification may cause the suspension of processing of an LSC. In this case, if no other LSC can proceed, the entire system is suspended.

Minimal Event

Semantics

A cold monitored message, which is minimal in the partial order induced by the LSC (it is not preceded by other events).

An LSC starts when its minimal event occurs. An LSC without a minimal event, i.e., its first event is not a cold monitored message, will not start and the messages in it will not be executed.

When an LSC starts, a live copy of it is created.

Active LSC

Visual Notation

None.

Semantics

An LSC with at least one live copy.

Live Copy of an LSC

Visual Notation

See Cut.

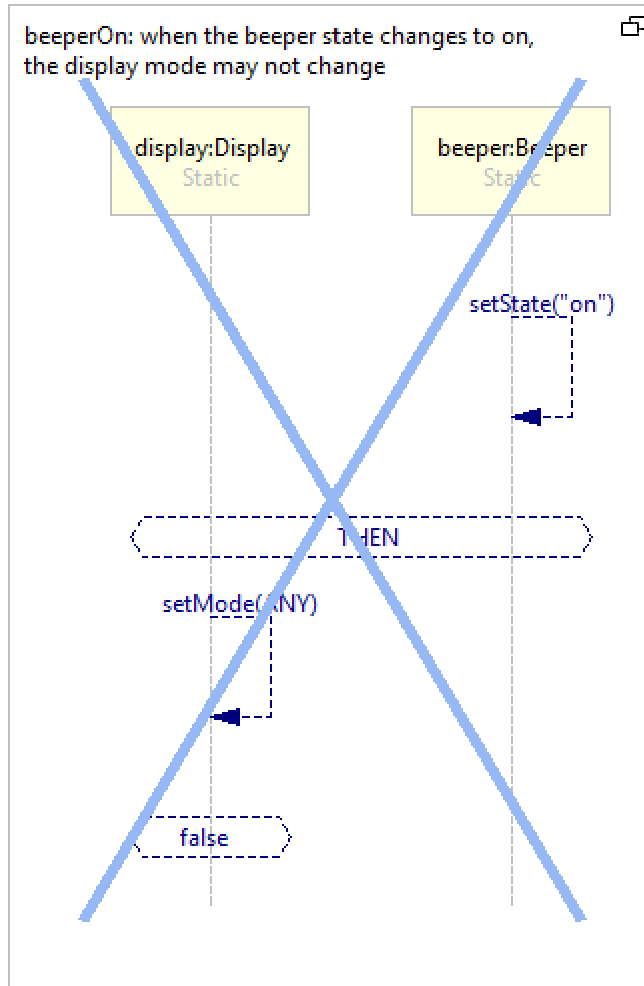
Semantics

A live copy of an LSC is a copy of an LSC in the specification associated with a cut, bound and unbound lifelines, local variables, etc… The live copy is created when the LSC starts (i.e., following an occurrence of a minimal event). As soon as a live copy is created its participating constructs are those that are considered in the play-out process.

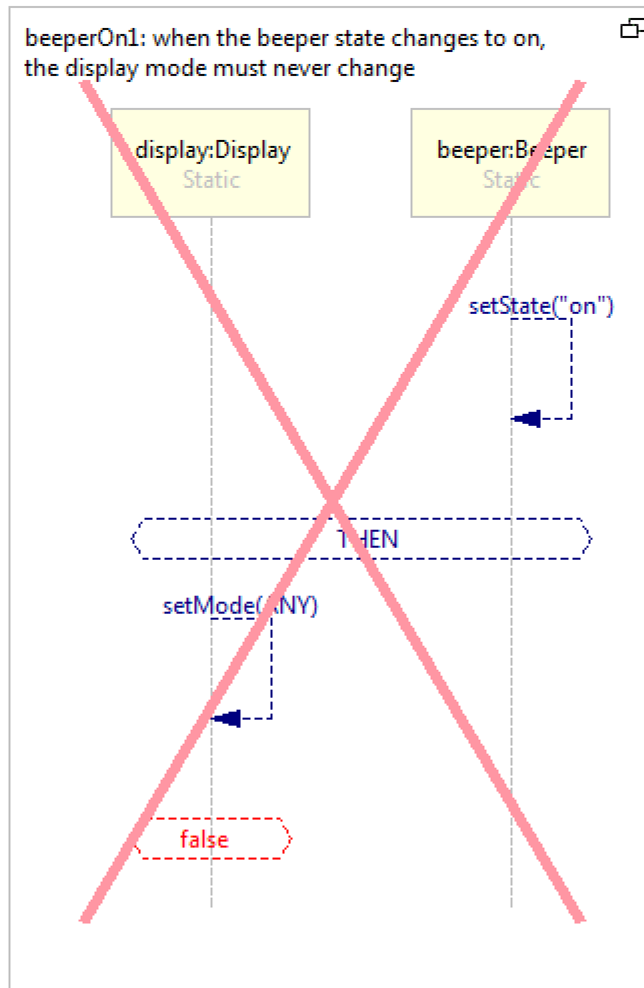
Violation of an LSC

Visual Notation

Cold violation: A blue X over the entire LSC.



Hot violation: A red X over the entire LSC.



Semantics

Cold violation:

Once an event occurs, it cold-violates a live copy of an LSC, if either:

- it is unified with an event that appears in the LSC but is not enabled there, and the current cut is cold, or
- the event is an evaluation of a false cold condition.

When a cold violation occurs, the live copy is closed.

A cold violation is not considered inconsistent with the specification and does not have other effects on play-out.

Cold violation is a way to “gracefully” exit in the middle of a scenario. It is sometimes explicitly defined in the specification using the exit construct.

Violation (= Hot violation):

Once an event occurs, it violates a live copy of an LSC if one of the following is the case:

- it is unified with an event that appears in the LSC but is not enabled there, and the current cut is hot, or
- the event is the evaluation of a false hot condition, or
- the event is an evaluation of a false cold condition and the current cut is hot.

When a hot violation occurs, the live copy is closed.

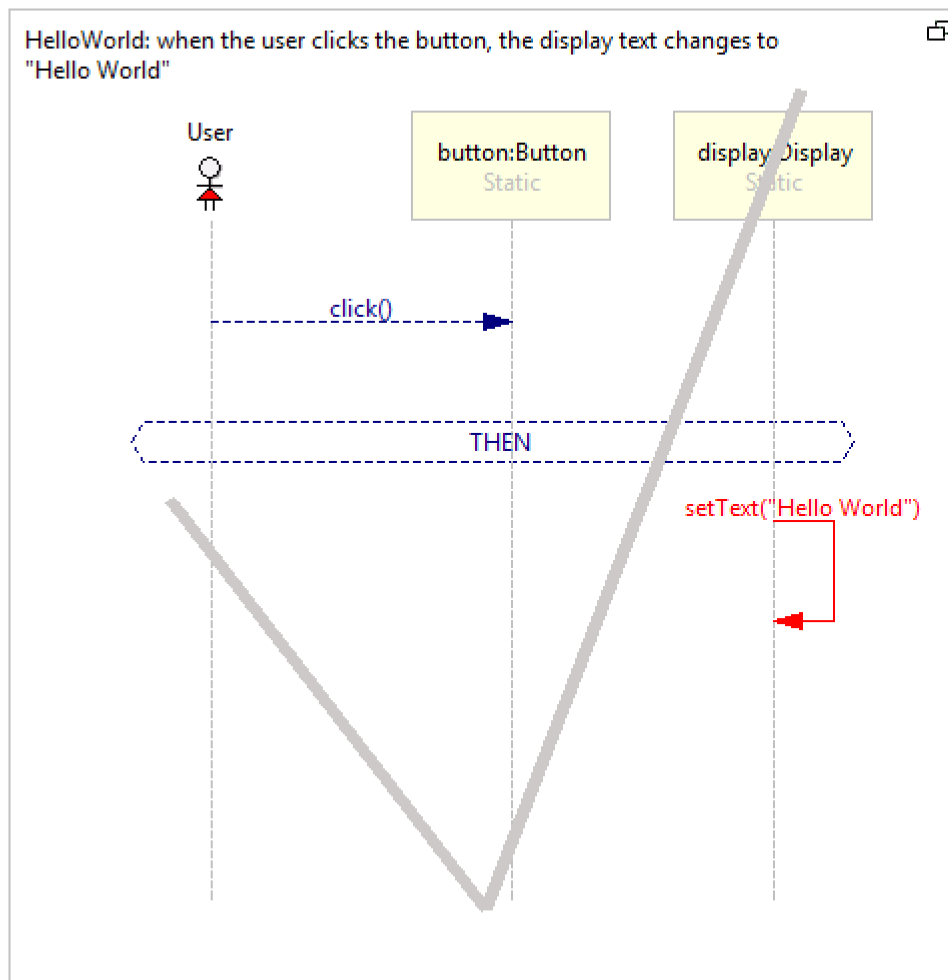
A run in which a hot violation occurred is considered as not satisfying the requirements.

Play-out tries to avoid hot violations when possible.

Completion of an LSC

Visual Notation

Gray checkmark (v sign) over the entire LSC.



Semantics

The current location of all lifelines is at the end (bottom line) of the main chart. The play-out has completed successfully for this live copy of the LSC. The live copy is then discarded.

Transition

Visual Notation

Transitions are shown as the progression of cuts in the play-out view during play-out.

Semantics

1. When an event occurs it is checked against all LSCs.

    a. If the event was minimal for the LSC, a new live copy is created, its static lifelines are bound, and its cut advances past the minimal event.

    b. The cut of all pre-existing live copies in which the event was enabled advance past the event, possibly completing the live copy.

    c. If the event that occurs violates one or more of the live copies, these copies are closed.

2. The coordinator collects current information from all live copies of all LSCs (sets of hot/cold enabled/violating events), and uses a strategy to select the next event to execute (if any).

3. When no event is selected by the strategy, the next event must be external; that is, it must arise from the invocation of a method by the environment. If any part of the application invokes a method associated with an LSC message, the message event is considered to have occurred. If an external event does not occur, the play-out process is considered deadlocked.

4. The occurrence of an external event signifies the beginning of a new superstep. System-events will be processed until there are no enabled system events. Play-out then waits for the next external event.

References

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

[2] D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: towards a comprehensive tool for scenario based programming. In *IEEE/ACM 25th Int. Conf. on Automated Software Engineering (ASE)*, pages 359–360, 2010.

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

[4] S. Maoz, D. Harel, and A. Kleinbort. A compiler for multi-modal scenarios: Transforming LSCs into AspectJ. *ACM Trans. on Software Engineering and Methodology (TOSEM) 20(4)*, 2011.

[5] A. Pnueli, Y. Sa'ar, and L. D. Zuck. JTLV: A framework for developing verification algorithms. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 171–174. Springer, 2010. URL: jtlv.ysaar.net.