Sample Code

Creating custom modelers for intelligent instruments

Create Custom Modelers with the CLIPS language and learn how the embedded rules engine works.

Download

iOS 12.2+ | iPadOS 12.2+ | macOS 14.2+ | Xcode 15.1+

Overview

Note

This sample code is associated with WWDC 2019 session 421: Modeling in Custom Instruments

Use this sample code project to step through the process of building a custom modeler for GoatList, an app that displays and modifies a list of goats. The GoatList app makes use of a `MobileAgent` pattern to break down a goal, such as sorting a list, into smaller subgoals. The subgoals execute at `stops`, locations where preconditions are met before execution continues.

Each Instruments Distribution Package target within the Xcode project uses incrementally more sophisticated modeling logic, written in the CLIPS language, to build an Instrument that visualizes the different moving pieces of the `MobileAgent` pattern as it is used within GoatList.

Configure the Sample Code Project

This sample requires Xcode 11 or later and iOS 12 or later.

Select GoatList from the target choices in Xcode to run the sample app in the iOS Simulator or on an iOS device. When the app launches, select a Modeling target to run on your Mac. The Modeling target launches Instruments. Next, select the Blank template and click the Add (+) button in the upper-right corner to search for the Instrument that was created when building the target. Searching for “Mobile Agent” should reveal the Custom Instrument. Once you select the proper device and target process within Instruments, you’re ready to record.

Design a Custom Modeler for MobileAgent Activity

The `os_signpost` API lets you mark important events or intervals in your app and attach optional messages, and offers an effective and powerful way to get data into Instruments. The `Mobile Agent` pattern implemented by this sample app uses just three `os_signpost` messages, yet they are vastly functional.

The rules in the `ExecutionModeling` (`mobile-agent-modeling.clp`) target rely on the `os_signpost` data emitted by the GoatList app to generate intervals, during which a `MobileAgent` actively executes at a stop. The target uses the `Mobile Agent Exec` signpost to generate the execution intervals in two steps: first, it determines which `MobileAgent` is executing; and second, it tracks when a `MobileAgent` begins executing. The modeling rules handle this bookkeeping by asserting facts into working memory.

Once the sample app emits a `Mobile Agent Moved` signpost, the modeling logic checks for execution interval facts associated with the `MobileAgent` that emitted the `Mobile Agent Moved` signpost. If the execution interval facts exist, the execution is considered complete, and the `RECORDER` module consumes the output.

Incorporate MobileAgent Transitions

The `MobileAgent` also transitions, an event that occurs when the `MobileAgent` moves from one stop to another to execute its next subgoal. You can use this movement to set up an execution environment for a `MobileAgent`, wait until certain conditions are satisfied before letting the `MobileAgent` execute at a stop, or signal failure status.

The `Mobile Agent Moved` signpost marks transitions; it’s the same signpost the `Execution Modeling` target uses to determine the end of an execution interval. In `ExecutionAnd TransitionModeling`, this signpost plays a new role. In addition to serving as the end of an execution interval, it determines when a transition has begun. Similarly, the `Mobile Agent Exec` signpost also takes on a new role: it signals the end of a transition.

As a result, there are four possible logical steps the modeler must consider when processing a signpost:

1. If it’s a `Mobile Agent Exec` signpost, assert a new execution bookkeeping fact.

2. If it's a `Mobile Agent Exec` signpost *and* a previous *transition* bookkeeping fact is present in working memory, then close the transition interval.
3. If it's a `Mobile Agent Moved` signpost, assert a new transition bookkeeping fact.
4. If it's a `Mobile Agent Moved` signpost *and* a previous *execution* bookkeeping fact is present in working memory, then close the execution interval.

Implementing the four steps requires the use of CLIPS Conditional Elements (CEs). Examples of CEs include `and`, `or`, and `not`. As the names imply, these CEs activate when all, any, or none of their input expressions are satisfied, respectively. `ExecutionAndTransitionModeling` uses these CEs to build rules that match the four possible cases.

Compress Signpost Data

Although `os_signpost` is convenient for getting data into Instruments, consider compressing that data to reduce the burden imposed on the recording technology. For example, in GoatList, the `MobileAgent` pattern sends integer codes, known as kind-codes, representing the type of `MobileAgent` and `MobileAgentStop` instead of a descriptive string in the signpost message.

On the modeling side, these codes expand into their longer string representations. Every target in the `Modelers` folder contains facts that map an integer kind-code back to a string representation. When the signposts send a kind-code, the modeling rules map the codes to their string representation.

Prevent Incomplete Facts from Reaching the Recorder

The `Modelers` targets contain several rules for annotating and augmenting the working memory facts into a complete fact that is viable for the RECORDER module to output into your data tables. As the MODELER module outputs the facts, it's important to prevent the RECORDER module's rules from seeing an incomplete fact and inserting it into the data tables. There are different mechanisms at work that make sure your modeling logic executes fully before being passed down to the RECORDER rules, two of which are described below.

The first mechanism involves how CLIPS determines which rules to fire next. CLIPS ensures that all activated rules in the MODELER module execute before any rules in the RECORDER module execute. In other words, a RECORDER rule isn't interleaved with the execution of rules in the MODELER module. All viable MODELER rules execute before any viable RECORDER rule executes.

Further, you can place restrictions on the LHS of a rule to determine when it's appropriate to fire. For example, a RECORDER rule can specify that it requires a `mobile-agent-execution-interval` fact to have a nonempty description of the stop in which the execution occurred. Thus, even if the kind-code to string mapping rules didn't have a chance to fire during an execution

cycle, and CLIPS considers which RECORDER rules are ready to fire, the nonempty restriction implies that the kind-code to string resolution occurs before the data tables record any data.

Avoid Overlapping Intervals

Storing all of the execution intervals on a single track may lead to UI stutters, which occur when your modeler commits overlapping intervals onto the same lane. The overlapping intervals make it ambiguous as to which interval should be displayed on the lane.

Sometimes, however, it's necessary to have overlapping segments, such as if `MobileAgents` execute concurrently on different threads. To remedy this problem, you need to somehow distinguish between the sets of data so they can be displayed on separate lanes or tracks. If you have multiple agents running in parallel, they can be distinguished by the thread they're running on. You can use a plot-template to distinguish between agents. A plot-template serves as a cookie cutter to stamp out plots for each unique instance of a distinguishing feature.

In the multithreaded MobileAgent example from the previous paragraph, a plot-template can be constructed that creates a plot per thread. Since the plots for each thread are distinct, the overlapping intervals won't contend for the same lane. The `PlotTemplatesModeling` target within this Xcode project uses a plot-template to resolve this problem. For more information and an example of constructing a plot-template, please refer to the `PlotTemplatesModeling` target.

Use Speculation Rules

Many interesting modeling problems deal with intervals at some point or another, and certain modeling constructs must be implemented to support immediate mode when dealing with intervals. Immediate mode in Instruments displays data on the Standard UI as soon as it is available, instead of processing the data as a large batch and rendering it once the trace completes. Instruments such as `Time Profiler` and `Core Data` support immediate mode.

To support immediate mode, you must take the concept of `Speculation` into consideration when building your modeler. For instance, in the case of a long-running `Mobile Agent`, the modeler rules assert the initial execution bookkeeping fact, but they have no way of knowing when the interval comes to a close. A `Mobile Agent` may be delayed in its execution or it may be executing an expensive task. In either case, there's an indeterminate amount of time between the initial `Mobile Agent Exec` signpost and the closing `Mobile Agent Moved` signpost. Without special rules, the Instruments UI will be forced to either remodel your events after the trace completes or stutter the UI in order to insert the new interval. If an interval is incomplete before the trace ends, the interval will also be omitted from the trace.

To remedy this, create a `speculation rule`. The `Analysis Core` asks your modeler to speculate about the end of an interval in order to render the interval's progress before it closes, as

well as give your modelers a chance to infer how an interval might close once the trace ends.

A `speculation rule` operates very similarly to other rules within the RECORDER module. It too outputs entries into a data table, but only temporarily. The `speculation rule` also doesn't have the complete information about an interval. For example, consider the `mobile-agent-recording.clp` file in the SpeculationModeling target. The `speculation rule` requires the presence of a `speculate` fact in the working memory. The fact includes a slot for the time at which speculation was requested.

```
(defrule RECORDER::speculatively-record-execution
    (speculate (event-horizon ?end))
    (table (table-id ?output) (side append))
    (table-attribute (table-id ?output) (has schema mobile-agent-activity))
    (mobile-agent-execution-started (start ?start)
    (instance ?instance) (stop-kind ?stop-kind&~sentinel))

    =>
    (bind ?duration (- ?end ?start))
    (create-new-row ?output)
    (set-column start ?start)
    (set-column duration ?duration)
    (set-column instance ?instance)
    (set-column state "Executing")
    (set-column activity-type "Green")
    (set-column stop-kind ?stop-kind)
    (set-column-narrative activity "Executing at stop %string%" ?stop-kind)
)
```

Although this rule doesn't have the complete information about the `mobile-agent-execution-interval`, it does have the initial `mobile-agent-execution-started` bookkeeping fact, as well as the time at which speculation was requested. These two times can be used to determine a speculative duration. The rest of the speculative data table is filled out with any information that could be gathered at the time of speculation.

With the speculation rule in place, the `Analysis Core` can now query the modeler when it needs to perform any speculation.

Specify Engineering Type Tracks

Instruments 11 and later include support for engineering type tracks. When combined with augmentations, engineering type tracks allow you to specify an engineering type to add in the Instruments UI dynamically alongside your other instruments' tracks as your data tables populate.

For example, if there are multiple threads executing the `MobileAgents`, the responsible threads can be displayed as tracks in the UI, with customizable lanes and titles. The engineering type tracks effectively promote ordinary engineering types into first-class citizens within the Instruments UI. Augmentations then work with these promoted types to visually render them onto the Instruments UI.

The `EngineeringTypeTracksModeling` target uses these features to display visited stops as separate tracks in the UI. The activity of the MobileAgents displays within these tracks.

CLIPS Modeler Functions

Building a custom modeler requires the use of the CLIPS language. Included with Xcode, the CLIPS distribution provides a rich set of functions for extracting and deducing information from your facts. This sample project made extensive use of CLIPS functions, but not every single one had a use for modeling GoatList. To that end, a broad variety of CLIPS functions are documented below.

Data Output Actions

`create-new-row`

Creates a new row in an output table that is bound to the modeler. The only parameter is the INTEGER `table-id` of the bound table. This is typically matched as part of the rule, using the table schema and attributes to locate the table-id. Once the row has been created, `set-column` can be used to fill in each column of the new row. Columns that are not filled out assume a default value, which is typically the sentinel for that engineering type if one is defined.

```
(table (side append) (table-id ?output))
(table-attribute (table-id ?output) (has schema example))
 =>
(create-new-row ?output)   ;; create the row before trying to set columns
```

`set-column`

Sets the specified column of the most recently created row in the RHS of the rule. The first parameter is the column mnemonic, as defined in the schema, and the second parameter is the value to set. Conversion from the implementation type (e.g. INTEGER, STRING, FLOAT, EXTERNAL-ADDRESS) to the engineering type of the column will be done automatically when possible.

```
(set-column size-column ?example-size) ;; Sets size-column to the value ?example-siz
```

`set-column-narrative`

`set-column-narrative` is similar to `set-column`, except that it applies a format string to create an engineering value of type `narrative`. Narratives are like long text strings; however, each piece of the narrative is preserved and tagged with the engineering type specified in the format string. This information allows the UI to break the string down later for a richer presentation. The first argument is the column mnemonic, followed by the format string, followed by the series of arguments, as in a printf-style format string.

The format string argument contains tokens surrounded by '%', such as: `Some text with a %thread% in the middle`. The value of the token must be a valid engineering type identifier. This example is a "thread" engineering type.

```
(set-column-narrative narrative-column "Some text with a %thread% in the middle" ?th
```

Data Transformation

`process-from-thread`

The "thread" engineering type is associated with a particular process. This function takes a thread value as an argument and produces the process object with which it is associated.

```
(bind ?proc (process-from-thread ?thread)) ;;  Extracts the process object from ?thr
(set-column process ?proc) ;; set the process column to the value ?proc
```

`pid-from-process`

Extracts the INTEGER pid from its argument, which must be a `process` engineering type.

```
(bind ?process (process-from-thread ?thread))
(bind ?pid (pid-from-process ?process))
```

`bit-test`

A Boolean return function that tests the nth bit of the first argument, and returns true if it is set, false otherwise. The second argument supplies the bit index to test. The bit index ranges from [0-63], where bit 0 is the least significant bit of the INTEGER.

```
(bit-test ?value-to-test 63) ;  returns true if the most significant bit in ?value-t
```

`extract-bits`

Returns an INTEGER by extracting a range of bits from the first argument. The return value has been shifted over such that the lowest bit in the extracted range becomes bit 0. Bit 0 is the least significant bit. The second and third arguments hold the range of bits to extract, inclusively. Conventionally, the highest bit in the range is the second argument, but if they are reversed, the function still works as expected.

```
(extract-bits ?integer-value 63 32) ; extracts the most significant 32-bits and shi
```

`make-int64`

Takes two 32-bit INTEGER arguments and concatenates them to create a single 64-bit INTEGER return value. This function takes the upper 32 bits of the first argument and place them into the upper 32 bits of the return value, and takes the lower 32 bits of the second argument and places them into the lower 32 bits of the return value. This function is typically useful for reconstructing a 64-bit value that has been broken apart into multiple fields. Even though the return and argument values are signed INTEGERs, they are treated as unsigned bit patterns internally.

```
(make-int64 ?msb ?lsb)
```

`memory-to-string`

Treats its INTEGER arguments as a run of continuous memory values and extracts a UTF8 encoded string from it. The function automatically detects if the run of INTEGERs are 32-bit or 64-bit, assuming that the run is consistently one or the other, but *not* a mixture of both. The UTF8 encoding convention allows us to reliably make that determination with little overhead.

Note: you cannot pass a multifield value in as a single argument. On failure, this function will return the symbol `sentinel`.

```
(memory-to-string ?part1 ?part2 ?part3) ; treats the three INTEGER arguments as a ru
```

Miscellaneous

`log-narrative`

When the Analysis Core has been configured with the extra logging enabled through the Instruments Inspector, a modeler-log table is created that is written to with this function. The

function does nothing when logging is disabled, although there is still a small amount of overhead in the call. The first argument is a format string, and the following arguments are interpreted like a printf-style argument.

```
(log-narrative "Resolved stop kind code %uint64% to %string%" ?stop-kind-code ?kind)
```

`(new integer-array)`

Creates an empty `integer-array` and returns it.

An EXTERNAL-ADDRESS type of `integer-array` has been built in that allows the modeler to bypass having to save runs of integer data in multifields. Multifields are generally efficient, but with a high rate data or when you don't want changes to the array to cause rule activation, you may want to create an object of this type and use it in place of a multifield value. `integer-arrays` are typically useful when your modeler needs to accumulate long lists of identifiers that it will emit as a single array at the end of modeling, or some other periodic event like a "tick".

`add-integer`

Appends the second argument to the `integer-array` in the first argument. Does not cause rule activation.

`nth-integer`

Returns the nth integer of the `integer-array` in the first argument. The second argument is n. Follows CLIPS index convention where 1 is the first element, and not 0 like in many other languages.

`length-of-array`

Returns the nth integer of the `integer-array` in the first argument. The second argument is n. Follows CLIPS index convention where 1 is the first element, and not 0 like in many other languages.

`sort-array`

Sorts the `integer-array` in the first argument. Does not cause rule activation.

`duplicate-array`

Copies the `integer-array` in the first argument and returns the EXTERNAL-ADDRESS.