Boku Programming Design

By Michael C. Miller

This documented describes the overall design of the programming model used in Boku prototype and speculates about possible future enhancements or changes. For details into the current programming model please review the document “Boku Programming Object Model”.

# Goals

Boku at its core is about exposing programming of virtual Actors (robots) to a novice audience. Specifically trying to target the below 10 age group. With this idea in hand, there were a few goals that we wanted to achieve.

Have actors react and feel real, like…

* Multiple natural inputs like sight, hearing, and touch.
* Playful outputs like expressions, kicking, and holding.
* Natural reactions and movement rather than exact robotic movement.

Minimize complex programming concepts that have been identified as problems for younger audiences, like…

* If / then / else
* Loops
* Variables
* Breaking code (exceptions)

Expose actions as higher level concepts, like…

* “Move Toward” rather than “turn right 34 degrees”
* “Eat” rather than “grab object, lift it to mouth, and then eat”

Rich set of programming tiles to maintain interest and allow more complex things to be built.

The ability to arbitrarily included new tiles and they just work

# Current Design and rough edges

## Parameter tiles broke arbitrary inclusion

The original “behavior” model required no direct understanding between any two tiles. This would allow any new tile to just be inserted and it would work. Enhancements restricted some tiles from being used from others to limit completely unusable combinations. But even this restricted the UI in some cases. An example would be a reflex like the following.

When - Scored – 10 points – Do – Move – Toward

The 10 points filter originally was a standalone tile that did its own filter checks; so the Scored sensor did not need to know anything about points filtering. But the model was switched to a parameter model, such that the points filter just provided the number and the sensor had to look for color filters and apply them. So if you add a new parameter to the system, you now have to update the sensor to filter with the new parameter; thus breaking the arbitrary inclusion design.

An example where this made a feature possible is either the time or score. With the following reflex

When – Timer – 10 seconds – 5 seconds – Do – Switch – Task B

Previously the timer value (a single tile) was the real timer and provided its own value. But the new parameter model allows the Timer to add all the parameter tiles (above shows 15 seconds) to allow more flexibility; but at the cost of arbitrary tile addition since any new filters would require changes to the timer sensor to look for them.

Overall the change has a positive effect, since new additions would be few and making changes to related tiles also would not be hard.

But it also will restrict us if we allow users to program an arbitrary tile.

## Understanding how Arbitration effects reflexes is complex

Arbitration gives us the ability to prioritize reflexes on their output even though they may not all run at the same time (general concept is that they run in parallel). Given the following example…

When – See – Fruit – Do - Shoot

When – touch – boku – Do - Shoot

When – touch – fruit – Do - glow – red

The actor will shoot at fruit first, and only shoot at boku when it sees no fruit. Since Boku can only be shooting at one thing at any given time, then the Shoot arbitrator picks the one to shoot at with the order from top to bottom providing the priority. Further, since touch is an event sensor (will be) and can fire out of sequence to See, arbitrator retains state for a reflex and continues to keep it within one consistent behavior. Without arbitration, defining which happens and which does not requires more complex visual information like If/ Then /Else and even more complex relationships between reflexes (today, the fact that the touch maybe event driven). But in this case, it’s easy to describe that when two shoots happen, the top one will happen and is easy to understand.

But the arbitrator is specific to the Actuator (output, in this case shoot). This simple arbitrator works fine and we have to date only needed one other type of arbitrator. Take the next example…

When – See – Red – Fruit – Do – Move – Toward

When – touch – Fruit – Do – Eat

When – See – Green – Fruit – Do – Move – Avoid

The more complex arbitrator is the movement arbitrator. Since movement actuator supports merging reflexes so more complex statements can be made like the above where the “toward” vector is merged with the “avoid” vectors. It’s understandable when reading but understanding how the arbitrator works and how it is merging the information is more complex.

Overall, it does a pretty good job of simplifying the users experience while retaining flexibility. If the tiles are picked well, the deep understanding of the arbitration is not required by the user as the tiles should be intuitive. Further, there is no easy replacement for the system that will not impose more complex ui (inter-related reflex with if/then/else and how to manage different sensor input timing).

# Future thoughts

## Complexity in tiles to handle filtering and modifying should be reviewed

With the above mentioned changes to the filters/modifiers for parameter support; the complexity originally present in the interfaces for handling the arbitrary modification should be reviewed. There are still cases today where it is being used (the same example above for timer, there is actually a hidden timer trigger filter that does the real timer work as sensor have no per reflex state as they are shared and one of the filters must manage this state).

Today, for modifiers, there are two methods in the interface, ModifyAction and ProvideParam, which should be reviewed. If the modifiers just become property bags (parameters) then these two methods are not needed and the actual selector, actuator, or actuator similar (work done for the actuator but not in the actuator, like most VerbActuators) will do the custom work with the parameters. This change is further along than filters; as you can see an example with the “Shoot”. In GameActor.DoShoot, it will walk the modifiers as if they were parameters and apply the values to the missile as needed.

Today, for filters, there are two methods to the interface, MatchTarget and MatchAction. If filters just become property bags (parameters) then these two methods are not needed and the actual sensor will do the custom work with the parameters. The Sensors already walk the filter lists to apply them with occasional custom requirements, so this on the surface seems like a good approach. The issue then becomes sensors have no per reflex state.

Since sensors do not contain per reflex state, then how would you do the score sensor (which fires only once the score value has been reached or passed, not every time it is reached or passed). Today, this is handled by the ScoreTriggerFilter (which is a filter the user never sees and is automatically added), which contains the per reflex sensor state that it has been triggered. Currently this relies on the above methods that we would like to get rid of. Again this could be changed to just be a custom call by the Score sensor into it for the state (variable filter, analogous to the parameter filter). You can see now, that as we deviate from the original behavior model design, we are now approaching a standard language design; which opens the question whether a bigger redesign down those lines might be interesting.

## Tile Compatibility standardized

The original intent was that compatibility could be defined in such a way as its just parameters in the CardSpace.Xml file. While for the most part this is true, there is still complex compatibility checking within the tiles. The intent was to have a way to do complex compatibility with code and refactor into standards as we understood more.

Currently the work to handle the count of type is done on a case by case basis. So how Slowly restricts itself to three but allows for up to three is custom within the modifier Slowly rather than a shared mechanism. Since there are others that have similar checks this would be good refactoring.

## Multiple When Clauses

Since the original creation, we have identified a desire to add functionality (and complexity) back into the system by supporting multiple when clauses in a single reflex. This would allow a statement like…

When See Boku AND

When See Balloon Do Shoot

For this to function, some decisions have to made or exposed for the user. Like, is the output from both when clauses merged and handed to the single Do Clause as input (set of items) or is the Do Clause associated with one of the when clauses and the other clauses acts like an Inhibitor (behavior programming jargon).

This will also require the reflex to include abstractions for clauses where today it’s only a visual grouping. Further, other code cannot assume a reflex has a single sensor and will have to change to support enumerator clauses and sensors.