User Guide for Atlas Looping Construct

Project Team:

Rachel Hahn	rhahn@wpi.edu
Ian Johnson	icjohnson@wpi.edu
Benjamin Mattiuzzi	bmmattiuzzi@wpi.edu

gr-roboticlang@wpi.edu

Project Advisor:
Professor Michael A. Gennert

This is supplemental material for the paper "The Design and Implementation of Looping Constructs for Robots".

Introduction to atlasLoop

The construct that we created, atlasLoop, is designed to help a programmer be able to easily encase a robotic action in a looping function. Specifically with the robot Atlas, there are issues where an action may have to be run multiple times before it is completed correctly. This code can easily get complicated and confusing, so we designed this function to automatically do some of that work.

To best understand how atlasLoop works, look at the action of picking up a hammer. This action contains many sub-actions such as opening hand, extending arm, moving arm above hammer, closing hand on hammer, and moving arm up. These sub-actions should be run sequentially, but some of them depend on the success of the previous sub-action. For example, moving arm above hammer needs the arm to be fully extended in order for it to successfully complete. Whereas the sub-action move arm up does not depend on the success the the hand closing on the hammer since no matter what is in the hand, it can still move its arm up.

A programmer can easily program this by grouping the sub-actions when passing them into atlasLoop. The action *pick_up_hammer* has the sub-actions (1) *open_hand*, (2) *extend_arm*, (3) *move_arm_above_hammer*, (4) *close_hand_on_hammer*, and (5) *move_arm_up* with the grouping as follows: {1}, {2, 3, 4}, {5}. This tells atlasLoop that sub-actions 1, 2, and 5 can be run without depending on any previous actions, whereas sub-actions 3 and 4 depend on the correct completion of sub-actions 2 and 3 respectively. To implement this dependency, atlasLoop sets breakpoints (b) to be at the beginning of each group: {b1}, {b2, 3, 4}, {b5}. The breakpoints mean that if any sub-action in a group fails, it will jump to the first sub-action in that same grouping. Similarly, if a whole group fails too many times, then atlasLoop will jump back to the beginning of the previous grouping. For example, if sub-action 5 fails 10 times, atlasLoop will start at sub-action 2 and try the sequence all over again. The full code for this example can be found in the section Code Example C.

Adding atlasLoop to your Project

Adding atlasLoop to your project is quite simple. Your project must first include the `AtlasPrototype.cpp`, `ActionObject.cpp`, and `pch.h` files. Then, you must include the `AtlasPrototype.h` file at the top of the code file that you would like to run atlasLoop in: `#include "AtlasPrototype.h"`. You can then use the functions that are listed in the `AtlasPrototype.h` file as if they were functions that you had written.

Interacting with atlasLoop

At its most detailed, atlasLoop takes the following arguments:

1. A `Loopstyle` telling atlasLoop which mode to use. Currently supported modes are "`Standard`" and "`Other`." Defaults to "`Standard`" if none provided.
2. A `std::list` of `ActionObjects`, which are the actions that the loop will be performing, in order.
3. An ordered `std::list` of integers, representing indexes in the list of `ActionObjects` that will serve as breakpoints. Defaults to empty if none provided

Additionally, an `ActionObject` constructor can take the following arguments:

1. A pointer to a function. (must take no arguments and return a boolean)
2. A `std::list` of pairs of ints `(A, B)` representing that atlasLoop should jump back B steps upon hitting A consecutive failures. Defaults to pairs of A=1 and B>> if none provided.
3. An integer representing the minimum number of consecutive successes for the Action to be considered passed. Defaults to 1 if none provided.

However, you do not have to specify all of these values when you invoke atlasLoop.

There are two types of ways to call atlasLoop. The first, shown above, requires you to make your own `ActionObjects`, and pass them into atlasLoop using

```
atlasLoop(Loopstyle, std::list<ActionObject>, std::list<int>)
```

This method allows maximum customizability, but less usability. Often, the second method described below will be sufficient, while being much easier to invoke.

The second method uses the class `objOrList`, referred to as `ool` in the code. Using this method, you pass pointers to functions into `atlasLoop`, and `atlasLoop` will automatically create `ActionObjects` and breakpoints. Note that this means that all breakpoints can *only* be at the beginning of a list, so the input must be constructed in such a way that all groups that depend on a single breakpoint are in the group following that breakpoint. Examples of the two methods of calling this function are shown below.

Code Example A:

```
#include “AtlasPrototype.h”

//Declare the function pointers
bool(*pointerToFunctionA)();
pointerToFunctionA = &FunctionA;

bool(*pointerToFunctionB)();
pointerToFunctionB = &FunctionB;

...etc

//Create lists of pairs
std::list<PairInt> listPairsA = { PairInt(2, 5), PairInt(3, 10) };
std::list<PairInt> listPairsB = { PairInt(1, 2)};
..etc

//Create the ActionObjects
ActionObject myObjectA(pointerToFunctionA, listPairsA, 3);
ActionObject myObjectB(pointerToFunctionB, listPairsB, 2);
...etc

//Create the list of ActionObjects before they are passed into atlasLoop
std::list<ActionObject> listOfActionObjects = { myObjectA, myObjectB };

//Call atlasLoop with the list of ActionObjects
atlasLoop(listOfActionObjects);

//nb. The above line is equivalent to
//atlasLoop(Standard, listOfActionObjects, std::list<int>{});
//as Standard and an empty list of breakpoints are the implied values.
```

Code Example B:

```
#include "AtlasPrototype.h"

//Declare the function pointers
bool(*pointerToFunctionA)();
pointerToFunctionA = &FunctionA;

bool(*pointerToFunctionB)();
pointerToFunctionB = &FunctionB;

...etc

//Create the array of ObjOrLists
ool arrayOfObjOrList[] = {
      ool(pointerToFunctionA),
      ool(pointerToFunctionB, pointerToFunctionC)
      ool(pointerToFunctionD)};

//Call atlasLoop with size of list
atlasLoop(arrayOfObjOrList, 3);

// The code will automatically place breakpoints at the beginning of each
// ool in the array. This means the above code will have breakpoints at
// actions A, B, and D.
```

Code Example C:

```
#include “AtlasPrototype.h”

bool(*pointerToOpenHand)();
pointerToOpenHand = &openHand;

bool(*pointerToExtendArm)();
pointerToExtendArm = &extendArm;

bool(*pointerToAlignWithHammer)();
pointerToAlignWithHammer = &alignWithHammer;

bool(*pointerToCloseHand)();pointerToCloseHand = &closeHand;
bool(*pointerToRaiseArm)();pointerToRaiseArm = &raiseArm;


//Create the array of ObjOrLists
ool arrayOfObjOrList[] = {
      ool(  pointerToOpenHand),
      ool(  pointerToExtendArm,
            pointerToAlignWithHammer,
            pointerToCloseHand),
      ool(  pointerToRaiseArm)
};

//Call atlasLoop with size of list
atlasLoop(arrayOfObjOrList, 3);

// The code will automatically place breakpoints at the beginning of each
// ool in the array. This means the above code will have breakpoints at
// actions pointerToOpenHand, pointerToExtendArm, and pointerToRaiseArm.
```