An Overview of XRobots

A Hierarchical State Machine Based Language

Steve Tousignant, Eric Van Wyk, and Maria Gini
Department of Computer Science and Engineering
University of Minnesota
Outline

- The problem
- Background on Hierarchical State Machines
- Augmented Hierarchical State Machines
- An example
- Conclusions
Some Problems with Robot Programming

• In a general purpose programming language, the problem space is different from the solution space
• Programmers tend to use global variables
• Modularity is difficult and code reuse limited
• Robot programs are event driven, based on states, with parallel execution, and hence are very different from the imperative style of programming
• Tracing and understanding how the program operates is difficult
Approach

• We want to bring to robotic software development researchers from different communities who do not traditionally work in robot programming:
  • Domain Specific Languages are becoming popular in multiple application areas. They use formal semantics and generation of code for the underlying support language.
  • Safety and Critical Systems approaches enables verification of properties, such as correctness and termination. They are used extensively in areas such as avionics and medical devices where software is regulated. They use Finite State Machines or Hierarchical State Machines.
Hierarchical State Machines (HSMs)

A set of states

Each state has a (possibly empty) set of actions that occur on Entry and on Exit

Each state has a (possibly empty) set of transitions each represented by a boolean condition-state pair

States may be nested inside other states
Problems With HSMs

- HSMs are used to describe problems with multiple states and complex state transitions because the hierarchical state structure helps handling complexity.
- However, there is no good method to pass data between states.
- Redundant sub-states (e.g. Straight Line) often occur.
Our Approach: Augmented HSMs

- States specify behaviors for the robot
- We make two extensions:
  - Behaviors are parameterized such that arguments can be provided upon state transition
  - Behaviors are first class constructs, so they can be passed around as parameters
- As a consequence, we can pass a behavior (such as Straight Line) to another behavior (such as Triangle or Square) instead of having to rewrite it. This increases reusability
An Example Behavior

/** #1 name and parameter list **/

Behavior driveStraightFor(float duration){

/** #2 Declaration Block **/
float newDuration;
/** #3 Entry Block */
Entry{
    rVel := 200.0; lVel := 200.0; newDuration = duration - 5.0;
}
/** #4 Transition block **/
Under Condition duration > 0
    Apply Behavior driveStraightFor(newDuration)
Under Condition duration <= 0
    Apply Behavior Stop() /** behavior stop not shown **/
/** #5 Exit block **/
Exit{
    rVel := 0.0; lVel := 0.0;
}
}
Executing a program

- We define all behaviors which have been entered but not exited as Active Behaviors
- We define the last behavior which has been entered but not exited as the Current Behavior
- All programs have a root behavior that is the ancestor of all the behaviors
- The order in which behaviors are entered is first-in-last-out. Behaviors are pushed on a stack on entering and popped on exit

To start a program
- Find the top-level behavior marked Initial and execute its entry block
- Repeat the process for all its sub-behaviors marked Initial
Executing a program (2)

- For each *Active Behavior*, from first entered to last, check the conditions of its transitions in order
  - The first condition found true enables its transition, which is executed by
    - Executing the exit block of all active behaviors back to the parent of the target behavior
    - Executing the entry block of the target behavior
- Only one behavior can be entered per cycle, so the target behavior must be a child or sibling of the active behavior
  - A cycle consisting of reading the sensor data, checking the transitions of the active behaviors, and writing the actuator values
Parameter Passing

- Parameters can be passed in one of two ways:
  - by reference, where we pass a reference to the original data storage location
  - by value, where a copy of the data is used locally
- We pass a behavior by reference when we want to use the static location of the behavior for the transition
- We pass a behavior by value when we want the behavior to be a sub-behavior
- This enables building behaviors out of simpler behaviors and reusing behaviors
Example

• The robot has to move on a square path 4 times, then a triangle path 4 times, and to continue indefinitely unless it hits an obstacle

• We can define the following hierarchy of behaviors,
  • Root
    • StraightLine
    • Start
    • Square
      • Right corner
    • Triangle
      • Triangle corner
    • Stop
/** #1 Root behavior with sensor and actuator parameters **/

Behavior root (sensor bool rBumb, sensor bool lBumb, actuator float rVel, actuator float lvel) {
  Behavior straightLine (  /** #2 parameter list to pass arguments, including behavior**/
      ByRef Behavior nextBehavior(int, ByVal Behavior), ByVal Behavior avoid(), int count) {
          Entry { /** #3 setting the clock and differential-drive**/
                   clock := 0.0; rvel := 100.0; lvel := 100.0;
               }
  /** #4 calling a behavior specified by a by-value parameter **/
  Under Condition rBump || lBump Apply behavior avoid()
  /** #5 calling the behavior specified by the parameter nextBehavior with parameter count and avoid **/
  Under Condition clock > 25.00 Apply Behavior nextBehavior(count, avoid)
  }
  /** #6 the initial behavior start initializes the system calling the behavior square **/
  Initial Behavior start() {
      Under Condition True Apply Behavior square(0, stop)
  }
  /** #7 the behavior square takes the count of the squares drawn so far and an obstacle avoidance behavior **/
    Behavior square (int sqCount, ByVal Behavior avoid) {
      Behavior rightCorner (int cornerCount) {
          Entry {
               clock := 0.0; rVel := 250.0; lVel := 25.0;
             }
        Under Condition clock > 7.0 && cornerCount < 4 Apply Behavior straightLine(square, cornerCount)
        UnderCondition clock > 7.0 && cornerCount == 4 Apply Behavior square(sqCount, stop)
        Exit {
               cornerCount := cornerCount + 1;
            }
        }
  } ....
Technologies Used to Develop XRobots

- Silver – an Attribute Grammar Language used to develop the compiler
- Haskell – used to implement the algorithms and data structures of the language
- C based Haskell to robot driver interface
- Player/Stage & iRobot's Create Open Interface
Challenges

• Ensuring that invoking a by-reference behavior follows standard rules for behavior transition
• Error detection when passing by-value behaviors
  ◦ Static vs. dynamic
• Dealing with the pragmatics of the language
• Define the formal semantic of the language to ensure the meaning of each construct in the language is properly defined
Conclusions

• Developed XRobots – a language for programming mobile robots based on HSM
• Presented an augmented model for HSM
  • States are first class constructs
  • States can be parameterized
• Narrowed the distance between the solution space and the problem space