Rule Processing Optimization in the Atlas Reactivity Engine

Patrick Flanagan, Cheenu Madan  
{pflanag, cheenu.madan}@ufl.edu

**Abstract**

The Atlas Reactivity Engine programming tool was optimized by improving the code using a reactive model based programming paradigm and adding an optimizing layer between the command interpreter and the reactivity engine.

1 Introduction

The Atlas Reactivity Engine (RE) is a programming tool that is based on the reactive model for pervasive spaces. Rules are defined using the event, condition, action paradigm, and are maintained and triggered by the Reactivity Engine.

|  |  |
| --- | --- |
| Fig. 1.1. Front view of the Gator Tech Smart House | The Atlas RE keeps track of the latest readings of the sensors (represented by Atlas Emulators) and uses these readings to update whether an event is true or false. However, exchanging messages between the sensor and the RE increases power consumption in the sensor network. Since all these messages pass through the Atlas middleware, so there is increased consumption at the Atlas middleware.  Hence, there is a need to optimize the RE such that it retains its reliability and at the same time reduces the communications between the sensors and the RE. |

This is why we introduce an optimization layer between the Command Interpreter where the user defines events, actions, conditions and rules and the Atlas Reactivity Engine which manages keeps track of these events, actions, conditions and rules.

2 Reactivity Engine Optimizations

On investigation of the source code of the original Atlas RE, we determined that the original RE had used a procedural programming approach. Since the Atlas RE concept is based on the reactive model for programming pervasive spaces, we found the approach used in the original RE to be ill suited.

We overhauled the RE using the following approach - an object-oriented approach where the objects constitute a dependency graph such that an object re-evaluates its state whenever an object it depends upon changes.

We created classes for Sensors, Actuators, Events, Conditions, Actions and Rules so that objects could be created whenever a sensor or actuator came online or when an event, condition, action or rule was defined.

Optimization: Overhaul of Events

This changes is most obvious in the manner that the RE handles events. Each event is represented as a binary tree and each node has a set of attributes (see implementation for a detailed description of the structure). An atomic event (eg. H50[20-40]) is a single node tree. A composite event is a multiple node tree where the leaves are simple events and every internal node is a composite event of the form left\_child operator right\_child.

Optimization: Subscription Management

The old Atlas RE subscribed to the sensors as soon as they came online. Hence it continuously updated its sensor readings in a hash table (we now do not use a hash table, instead we represent each sensor as an object and store it’s latest reading in the value attribute). The truth-value of events was updated when the RE was running.

However we do not need to evaluate events and therefore maintain a list of up to date sensor readings in the following scenarios.

1. RE is not running.

2. Event is defined, but is not associated with any rule. Hence, there is no need to evaluate event

3. Event is defined and associated with at least one rule. However all the rules which the event is associated with have their conditions set to false. Hence, no need to evaluate event as the rules will not trigger.

The subscription management optimization ensures that we subscribe to sensors only when the following is true:

1. RE is running

2. Sensor is associated with an event that is associated with a rule whose condition is true

This optimization greatly reduces communications between the sensors and the applications.

Optimization: Improved Command Parser

Previously, the RE used a crude series of StringTokenizer to parse input from the command line. The basic grammar of the system was supported, but arbitrarily complex commands would break the engine.

Using the tools CUP and JFlex, we defined a precise, safe grammar for the system. Events, thanks to their tree-like nature and the new parser, can be arbitrarily large and include any conceivable structure, including named and unnamed subevents.

3 Implementation

Describe how you implemented the optimizations in section2. Describe any data structure and always explain WHY, not just WHAT.

Include any other implementation details that you find important to report.

Overhauling of events

Events are represented using a binary tree. A simple event (eg. J37(100)) can be viewed as single node. A composite event can be viewed as a binary tree with simple events at the leaves and composite events at the internal nodes. Each node has a set of attributes associated with it, depending on the type of node. The nodes all implement the Event interface, which can be updated or evaluated.

This binary tree structure is an “optimization for the optimizations”. It allows many of the described optimizations to be easily implemented.

Subscription Management

The sensor associated with an event can be determined by for a simple event by accessing the OptEvent.sensorType field (always present for simple events, see table). It then subscribes to the event.

To determine the sensors associated with a composite event, we access all the leaf nodes of the binary tree representation the composite event and get the sensor type by accessing the OptEvent.sensorType field for each node. It then subscribes to each one of these sensors.

TFM Event

The TFMEvent, like all Events, implements the Event interface. It has a somewhat linked list-like structure in that it refers to another Event object. Its evaluation is implemented with Timer threads.

4 Performance Evaluation

You will perform two experiments and collect data to measure the achieved performance gains of your optimization over an unoptimized engine. You will be given a set of “workload” files to use (to LOAD). The workload will be described clearly so you will know what emulators you need to setup before launching your RE engine and loading the workload.

**Experiment 1: Memory utilization**

You will be given a section of code to include into the implementation of the LOAD command to monitor memory usage from RUN to STOP. This will include the Runtime.getRuntime().totalMemory() as well as the Runtime.getRuntime().freeMemory() methods. You will report memory usage for plain and optimized RE when running the “memory” workload from RUN to STOP.

**Experiment 2: Scalability**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 Rules | 10 Rules | 15 Rules | 20 Rules | 25 Rules |
| 121, 121 | 121, 120 | 122, 120 | 120, 120 | 120, 121 |
| 119, 120 | 120, 120 | 121, 123 | 119, 121 | 119, 124 |
| 120, 121 | 119, 120 | 121, 122 | 119, 120 | 119, 121 |

Unfortunately, there is no difference between our implementation and the old one. This is because even the first 5 rules referenced events that required that all of the 4 sensors be subscribed to, so no optimization occurred for any of the rules. If any of the rule sets had no references to one of the sensors, an improvement would have been noticeable.

5 Project Status

Until we received the test cases, we did not realize that multiple devices of the same type should have been supported. As such, we spent the last day attempting to implement multiple devices and only finished it at the last minute, and were unable to perform the performance evaluation. It will be finished for the presentation, however.

Removing a device is unsupported. If a service is unregistered, ideally events dependent on that service would simply evaluate to be false. As it is, the program simply fails to work.