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, there is also 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.

Previously, these concepts were represented as Strings which were parsed constantly. 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 change is most obvious in the manner that the RE handles events. Each event is essentially represented as a tree of events, where each node has a set of attributes (see implementation for a detailed description of the structure). At the leaves of the tree are simple events (eg. H50[20-40]). A composite event is somewhat analogous to a binary tree in that it references two sub events (which can be any kind of event) joined by an operator, such as the logical AND, OR, or the custom SECS operator.

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 of Strings. The truth-value of events was updated when the RE was running.

However we do not necessarily need to evaluate all events, and we 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 objects to parse input from the command line. The basic grammar of the system was supported, but arbitrarily complex commands with multiple layers of nesting could break the engine. We greatly improved the robustness, flexibility, and clarity of the grammar.

Optimization: TFM Events

New in this RE implementation is the concept of Time-Frequency-Modified Events, or TFM Events. These events allow the user greater freedom in designing rules and allow the system further opportunities for optimization.

In particular, the TFM Events need only update once every period (as defined by its frequency), which reduces communications with sensors. Also, once a TFM Event’s threshold is reached, it no longer needs to collect data from sensors, since it will simply continue to evaluate as true. Thus, communications are further reduced.

3 Implementation

Overhauling of events

Events are now represented using a tree-like structure. A simple event (eg. J37(100)) is an instance of the SimpleEvent class, and is effectively a wrapper around a Sensor object, with a filter for a certain range of sensor readings. 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 defines update() and evaluate() methods, among others.

This event tree structure is an “optimization for the optimizations”. It allows many of the described optimizations to be easily implemented and even facilitates the creation of future optimizations.

Subscription Management

The subscriptionManager() method is called upon the RUN or SET directives being parsed. A list of rules with true conditions is obtained, and those rules’ events’ sensors are subscribed to.

The sensor associated with an event can be determined by for a simple event by accessing the simple event’s getSensorNodeId() method.

To determine the sensors associated with a composite event, we call the addSensorsTo(Set s) method, which traverses the entire event tree (skipping TFM Events, for reasons expounded upon later) adding the sensors of all leaf nodes to the set as it moves along the tree.

Improved Command Parser

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 grammatically correct structure, including named and unnamed subevents.

The files parser.cup and lexer.flex contain parser and lexer specifications (respectively) that are generated when running the Ant build file build.xml. A parser and lexer are constructed for each command line string to be parsed. The parser then steps through a series of states defined by tokens passed to it from the lexer. Actions are performed in these states, mostly to scrub the command clean and catch any potential errors.

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. TFM Events, unlike Simple and Composite Events, are not handles by the subscription manager. Instead, a component of the RE dubbed the Scheduler handles TFM Events completely

The Scheduler uses Timer threads to manage all the TFM Events defined. For each TFM Event, a series of timers is scheduled and rescheduled. When a timer fires and calls for an event to be updated with a data pull from a sensor, a counter (internally called this event’s dirtyCounter, implemented with a custom, mutable Counter class) is incremented, so that when the sensor calls back with the data, the counter can be decremented and the TFM Event can properly record the data event so as to satisfy its Reporting Frequency requirements.

4 Performance Evaluation

**Reactivity Engine Test Set 1**

A tabular representation of the number of messages exchanged between the application and the reactive engine over a period of 2 minutes (120 seconds) is given below.

Since the unoptimized RE does not support some of the rules listed in Subsets 3, 4 and 5, the readings were not taken as it would be inconsistent compare old RE and the optimized RE in this case.

**Table 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Reading 1 | | Reading 2 | | Reading 3 | |
|  | Old RE | Opt RE | Old RE | Opt RE | Old RE | Opt RE |
| Subset 1 | 237 | 238 | 239 | 238 | 237 | 238 |
| Subset 2 | 238 | 237 | 238 | 238 | 237 | 236 |
| Subset 3 | - | 237 | - | 235 | - | 238 |
| Subset 4 | - | 239 | - | 237 | - | 237 |
| Subset 5 | - | 237 | - | 237 | - | 238 |

The number of messages exchanged between the application and the reactive engine are the same in both the old RE and the new RE. This is because the test set includes only 4 sensors and all the four sensors are registered in at least one rule (with its condition set to true). Hence the subscription management optimization is not effective.

However it must be pointed out that we cannot foresee any reduction in number of messages in cases when the RE has to subscribe to all sensors. In scenarios such as the Test Set 2, one should see a considerable improvement in performance in Subsets 1-4 (the number of messages exchange would be same in Subset 5).

However as we cannot compare the old RE and optimized RE using the Test Set 2 as the old RE does not support multiple instances of the same sensor, we defined a small Test Set where only 3 sensors need to be evaluated for the rules to demonstrate the subscription management optimization.

A gradual increase can be seen in the number of messages exchanged in the subsets of Test Set 2, which demonstrates the subscription management optimization.

**Custom Test Set**

|  |  |
| --- | --- |
| Rules |  |
| R1 = e1, c1, a1  R2 = e2, c1, a1  R3 = e3, c1, a1  R4 = e4, c2, a1  or   R1 = e1, c1, a1 R2 = e2, c1, a1 R3 = e3, c1, a1 R4 = e1+e2+e3, c1, a1 | Where e1 = N40(0) - Contact sensor e2 = T42(100) – Temperature sensor e3 = J4(20) – Pressure sensor e4 = G65(40) – Humidity sensor  c1 = TRUE c2 = FALSE  a1 = R45(50) – Servo |

The rules are kept simple so that an intuitive idea of the optimization is obtained. It can be observed that we do not need to subscribe to the humidity sensor until c2 becomes true or a rule is defined with e4.  
  
The readings of the old RE and optimized RE are shown below

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Reading 1 | | Reading 2 | | Reading 3 | |
|  | Old RE | Opt RE | Old RE | Opt RE | Old RE | Opt RE |
| Set | 237 | 180 | 239 | 180 | 237 | 180 |

As can be seen, the communications between the Reactive Engine and the sensors have been reduced by ~ 25%.

Another flaw in the old RE must be discussed here. The old RE subscribed to sensors as soon as they came online. Hence, even when the RE was not in the RUN mode, it was still exchanging messages with the sensors. So if the old RE was running for 1 minute and was open for 2 minutes, the readings would be the same as in Table 1 (not halved as it should be).

Using the test cases provided in test set 2, a second, more thorough analysis comparing the old Reactivity Engine (Old RE) to our new, optimized version (Opt) has been performed, with an emphasis on memory usage. Statistics for 5 to 25 rules, in increments of 5, evaluated over a period of 2 minutes, are recorded and discussed. Because Old RE does not implement TFM, the first 10 rules were selected so as to be able to be parsed by Old RE. If the rules do not parse correctly, that is not considered here, and any performance decrease to Old RE resulting from bad parsing remains in these results.

The 2 minute period was implemented by spawning a Timer thread which ran the STOP directive after 120,000ms. This enabled accurate timing, even when the GUI failed to respond. In Opt, a new directive was added to the grammar: “RUN n”, where n is the number of seconds to run before running the STOP directive.

Old RE was unable to function well even with only 10 rules. Though accurate data was obtained for t, Knoplerfish became unresponsive after the RUN command was given, so no further tests were performed beyond 10 rules on Old RE.

As can be seen, the number of messages per second actually decreases for Old RE after adding more rules. This is because Old RE stopped reporting data from half of the registered sensors.

Opt’s memory usage is significantly better than Old RE, consuming almost 1/3 as much memory for a comparable set of rules.

By subtracting the memory used before the RUN directive from the memory used after, the amount of memory consumed by rule evaluation process can be approximated. As can be seen, Opt consumes almost no memory for 5 rules, and increases moderately afterward, whereas Old RE uses almost all of the memory allocated to it (see graph on next page) to evaluate rules.

Opt also appears to scale well, as the amount of memory needed to evaluate rules as compared to the total memory allocated to it hovers around 30% for 15, 20, and 25 rules.

5 Challenges

Initially, the code was in one file, with almost no separation or abstraction whatsoever. Fighting through this made it difficult to think about the structure of the program, let alone conceive of optimizations for it. Refactoring the code and especially defining events as a tree structure were crucial in organizing the code and unlocking a series of ideas.

Thinking about TFM Events again posed a problem because they required waiting and/or scheduling tasks in the future. We decided to use threads to implement the TFM Events, which caused much grief initially, but ultimately seems like the appropriate course to have taken.

Finally, large changes to our engine were necessary after we were provided with the test cases. We discovered that our parser did not parse the grammar correctly and were forced to redefine several grammar rules before the input would parse. This was an unexpected and frustrating development, especially so close to the end of the project.

6 Project Status

To the best of our knowledge, our new version of the Reactivity Engine satisfies all of the requirements of the project definition. All old functionality has been retained, and much new functionality has been added, as well as a much-improved codebase. We have considered a few ways in which our version of the Engine might be further improved.

The way our command line currently parses information, poorly defined events or rules can create nameless TFM Events, which can then spawn timer threads and eat up system resources. The only way around this currently is to restart the engine. This is acceptable, thanks to the LOAD directive, but not ideal. Preferably, TFM Events would not spawn any threads until they were validated as legitimate parts of the user defined ruleset.

One might implement TFM Piggybacking. In the current implementation, this scenario can occur: Two TFM Events, each based on the same sensor, and active during the same window, can submit pullData requests to the sensor. Both TFM Events will have their dirty counters incremented, but when the first pullData call-back occurs, both TFM Events will have their dirty counters decremented. The second pullData call-back now occurs, and nothing useful happens. This is an area for potential optimization.

Since they are limited to at least a 1 second evaluation frequency, each TFM Event could submit a pullData request to a local Set of Sensor objects. At the end of each second, this Set could then pullData on each Sensor contained, thus communicating only as much as necessary.

More efficient event tree traversal when determining which events to subscribe to is another direction in which to optimize sensor communication. For example, if one event in a Boolean AND evaluates to false, there is no need to evaluate the other until the first evaluates to false. Since events are a tree structure, cutting off even one event at the top of the tree could potentially reduce the number of subscribed sensors. Algorithms for determining the minimal sensors required to evaluate the tree could also be developed.

Finally a more robust command line would be helpful. In its current state, nothing can be undefined. Other annoyances, like emulators changing node ID’s every time Knoplerfish restarts can be a pain. We actually began implementing LOAD file comments, though we put that aside in favor of higher priorities. Improvements to the command line could solve these problems, even though they are mostly mere annoyances.