DSM Rules V2

|Updated today at 5:54 PM by David A. Wood|Tags: None Add tags

Introduction

The primary languages supported by the DSM rule engine is the DSM rule language (DSM/RL). DSM/RL is distinct from the lower-level DSM SQL V2 rule language (DSMSQL/RL V2). DSM/RL provides the same capabilities as DSMSQL/RL, but at a higher level of abstraction. and in fact, DSM/RL is compiled into the DSMSQL/RL DSM/RL capabilities include

- Tuple/table definitions including support for the following types of tuples
  - Input - these are non-persistent tuples that are not manipulated by the rules, but which can be used as input to rules that change other tuples..
  - Persistent - these are persistent tuples that are set through evaluation of rules
  - Transport - these are temporary tuples that are exchanged between 1 or more application nodes.
  - Action - these are tables defining Java methods to be called at the end of a rule set evaluation.
  - System - these are tables provided by the rule evaluation run-time during rule evaluation.
- Addition/deletion of tuples via rule statements, which allow the following:
  - joins between one or more tuples and one or more of their column values or constants
  - conditions on column values
  - periodicity of rule evaluations

As an example, consider a state machine, which when it receives the tuple "Hello", responds with "Goodbye". (A variation of this is available in the DSM source tree).

```
transport msg(char[128] text);              // Defines the table through which messages will be sent/received.
persistent inbound(char[128] src, char[128] text);
log(Src, x) if msg(x)@Src;              // Copy msg into a log
msg("Goodbye")@Src if inbound(Src, "Hello"); // When we receive a 'Hello' messages, respond with 'Goodbye'.
```

And, to respond with 'Goodbye' to all inbound messages, just change line four to:

```
msg("Goodbye")@Src if inbound(Src, *);
```

Next we'll explore the semantics of the rule language and its syntax.

Changes Relative to V1

The following are changes relative to version 1 of the DSM rules implementation.

1. Only tuple additions/appends are allowed now. As such **-=**operator is not available.
2. The **+=** operator is replaced by the ***if***operator.
3. Transient tuples are no longer supported.
4. A new ***prev*** operator is provided to refer to the table state prior to the current rule evaluation cycle. It is used on persistent tuples referenced within join specifications.

Semantics and Syntax

The DSM language is designed around a data model of database-like tables, columns and rows. These tables are also known and tuple sets, and the rows as tuple. The rule language allows the definition of tables and their columns, and how new tuples should be created or delete based on existing or new tuples. In general then, the rule language is designed with two sections: a table declaration section, followed by a tuple modification section. The latter are generally referred to as rules. The general layout of a rules specification is then as follows:

- Table Declarations
- Tuple Rules

We'll discuss each of these in turn.

Table Declarations

A table is declared specifying a type (one of *persistent*, *input*or *transport*), it's name and the list of columns with a name and type for each column. Table and column names are any valid identifier, which is case insensitive. Column types are one of

1. int - an integer value.
2. float - float value
3. double - a double value
4. char[N] - a string of maximum N characters.
5. timestamp - a time stamp value including the date and time.

So in general the BNF grammar for a table is as follows:

<Declaration> := <table type> <ID> <Column List> ';' ;
<table type> := 'persistent' | 'input' | 'transport' | system ;
<Column List> := '(' <type> <ID> (',' <type> <ID>)* ')' ;
<type> := 'int' | 'float' | 'double' | char '[' <integer> ']' ;

This will be covered later, but it is noted that the persistent and transport tables have a hidden column that holds the time the tuple was created. In addition, the transport tables, have two additional columns : one holding the application instance id of the sender and the other the instance id of the receiver. These can be accessed via the attribute and @ notation provided by the rule language, discussed below.

Tuple Rules

Tuple rules are defined to either add or delete tuples from the local state based on the values of new tuples received by engine. Only *input* or *transport* tuples can provide new state into the engine. Input tuples are generally provided by the JVM application that hosts the rules engine (DSMEngine). Transport tuples are received from other rule engines. All rules are generally of the form

```
<modified tuple> 'if' <tuple join> (':' <conditions> )? (':' <modifiers> )? ;
```

where,

`<modified tuple>`- is the table receiving the modification based on the operator and the evaluation of the right-hand side (RHS) of the rule.

`<tuple join>`- specifies a join between 1 or more tables. Symbols for columns are effectively defined here which can then be used in other portions of the rule. Constants may also be used in the columns of the join expression.

`<condition>`- specifies optional conditions on the tuples which can not be expressed through the join expression.

`<modifiers>`- specifies optional periodicity and/or processing semantics of the rule.

As an example, let's return to the one of the rules from the introduction:

```
log(s,msg) if inbound(msg)@s;
outbound("Goodbye")@Src if log(Src, "Hello");
```

This rule adds tuples to the *outbound* table based on the tuples found in the *inbound* table. The RHS may define a set of symbols for the columns and a predicate over those symbols that when true, results in a tuple being added to the *outbound* table. This rule defines the 'Src' symbol as a reference to the instance id that sent the tuple to the engine. Furthermore, when there is an *inbound* tuple for which the *msg* column has the value "Hello", an *outbound* tuple is added with the *msg* column having a value of "Goodbye". The @<ID> notation is used only with transport tuples to access either the instance id of the application of the sending the tuples, or the instance id of the application instance to receive the tuples. When the @<ID> is used in the LHS, it refers to the instance id of the tuple recipient (i.e. where the tuple should be sent to). When used on the RHS, @<ID> refers to the instance id of the application that sent the tuple. So, in the above rule, we send an *outbound* tuple back to the application that sent the *inbound* tuple. This rule has no condition or rule modifiers.

Lists of rules may be separated into blocks using the *'block;'* statement. Rule blocks are used to control the evaluation of rules so that during evaluation a single block is evaluated before another.

Rule Head

The modified tuple definition appears on the LHS of the rule (or rule head) and defines the rows that will be either added to are removed from the associated table. Each value assigned to a column of the tuple is an expression of constants and/or symbols defined in the join (see below). If the table being assigned is of type transport, then the specification must also include the '@<ID>' notation to define the destination of the tuple. Some examples follow.

A(x,y)

A(x+7-y, y)

B("foo", x*y)@dest

Modified tuple specifications can make use of a single 'aggregate' function that can create a single value from a column across multiple tuples. The currently supported aggregate functions are:

SUM(x) - adds the values of the column corresponding to the symbol x for all tuples matching the specification on the RHS.

AVG(x) - averages the values of the column corresponding to the symbol x for all tuples matching the specification on the RHS.

MIN(x) - finds the minimum value of the column corresponding to the symbol x for all tuples matching the specification on the RHS.

MAX(x) - finds the maximum value of the column corresponding to the symbol x for all tuples matching the specification on the RHS.

COUNT(x) - counts the number tuples matching the specification on the RHS.

For example,

A(x, COUNT(x))

A(MIN(x), y)

Rule Body - Join

Although, the join portion of the rule can specify two or more tables and no correlation between them, it will usually be the case that the join is a predicate over column values for equality with other columns in the join or constants. So for example consider the following:

```
A(z,y) if B(x,z),C(x,y);
```

First we note that the join is between tables B and C which must have been previously defined. In addition, the join defines symbols x, y and z which are associated with the 1st column of B and C, the 2nd column of C and the 2nd column of B, respectively. By using the same symbol, x, in both the B and C tables, the rule requires that the tuples under consideration are those with the first columns of B and C having the same value. So, the effect of this rule is to creates a tuple in table A whenever there is a row in B with the first column value equal to a the first column in a row of C. The corresponding second columns of B and C are taken from the rows in question and inserted to a new tuple in A. Similarly, constants can also be used as the column values of a join, as follows:

```
A(x,z) if B(x,z),C(x,5);
```

which adds a tuple to A with the values matching a tuple in table B whenever there is a tuple in C that matches B's tuple in the first column and has value 5 in the second column. If a column's value is not part of the join, the '*' (asterisk) character may be used instead of defining a dummy symbol. For example,

```
A(x,z) if B(x,z),D(x,*,*);
```

As mentioned earlier, if a member of a join is a transport tuple, then the '@<ID>' notation may be used to define a symbol or constant for the instance id of the sender or destination of the tuple. If the @<ID> is used on the right-hand side, then it refers to the sender of the tuple. If on the left-hand side, then the destination of the tuple. The symbol can be used in condition expressions (see below) or to assign the destination of an assigned transport tuple, as in the following rule:

```
msg("Goodbye")@Src if msg("Hello")@Src;
```

or,

```
msg("Hello")@"party 2" if msg("Hello")@"party 1";
```

which might be used as a gateway to forward "Hello" messages from 'party 1' to 'party 2'.

prev Operator

The semantics of rule evaluation allow the author to refer to the state that was in effect prior to the current rule evaluation cycle. The ***prev*** operator may be used on persistent join specifications. For example,

```
A(x,y) if prev B(x,y);
```

copies the values of B, prior to this evaluation cycle, into A. The not operator may also be used with prev

```
A(x,y) if not prev B(x,y), C(x,y)
```

copies rows from C where for each row that does not exist in the previous state of B.

Rule Body - Conditions

The optional conditions section of the rule may be used to define more complex predicates over the column values (remember that only symbol definitions or constants can appear in the columns of a join specification). The condition is a logical expression in a format similar to that used by C and Java programs. The condition can make use of constants and/or the symbols defined in the join portion of the rule. For example,

```
x >= 10 && y > z*x

x == z || x == y

x != q

"foo"+x == y                    // however, both x and y must be strings
```

In addition, DSM/RL provides the ability to call functions within these expressions (see below for complete set of functions supported). So for example

```
LOCATE("foo", y) == 0

LENGTH(hostname) + LENGTH(x) > 32
```

Functions

In addition to the aggregation functions available for use in the rule head, the following functions are available for use in either the rule head or condition expressions.

String

- LOCATE (str1, str2) - returns the 1-based index of the location of str1 in str2. 0 if not found.
- LENGTH(str) - Returns the number of characters in the input value. Noncharacter data is implicitly converted to a character string.
- SUBSTR(str,start,length) - Returns part of an input character string as a string, starting at the location specified (1-based) and continuing for the given length of characters.

Date and Time

- YEAR(x) - Returns an integer that contains the year component of a date, time stamp, or character string that contains a valid date.
- MONTH(x) - Returns an integer that contains the month component of a date, time stamp, or character string that contains a valid date.
- DAY(x) - Returns an integer that contains the day component of a date, time stamp, or character string that contains a valid date.
- HOUR(x) - Returns an integer that contains the hour component of a date, time stamp, or character string that contains a valid time.
- MINUTE(x) - Returns an integer that contains the minute component of a date, time stamp, or character string that contains a valid time.

Arithmetic

- ABS(x) - Returns the absolute value of an expression, which must be one of the built-in numeric types. The return type is the same as the argument.
- MOD(x,y) - Returns the remainder when the first integer argument is divided by the second integer argument. The return type is integer. The sign of the result is determined solely from the sign of the first argument.

Arbitrary SQL Functions

- SQL(<ID><parameter list>) - allows the calling of any SQL function. The type of the function is NOT checked and so any typing errors will not be visible until run-time. For example, A(b,SQL(SQRT(x)) += B(x).

Tuple Attributes

Tuples may have attributes (non-declared column) values associated with them for which symbols may also be defined. Currently only the 'time' attribute is defined for the persistent and transport tuples. The attributes symbols are defined using a bracketed notation as in the following:

```
A(x,T) if B(x,y)[T=time];
```

in which a new tuple in A is created for every tuple in B, with the first column set to the first column of B and the second column set to the time the B tuple was created. Symbols defined for attributes (i.e. T above) can also be used in functions and/or reference in conditions. for example:

```
A(x,T) if B(x,y)[T=time] : HOUR(T) >= 12;
```

Rule Modifiers

Modifiers are used to refine the processing of the rule. The following modifiers are supported:

MSEC=n - specifies that the rule should not be evaluated more frequently than every n milliseconds, regardless of the input tuples.

choose(n[,m]) - allows the random selection of n tuples from the results of the join expression. m optionally specifies the minimum number of tuples required in the result. If m tuples can not be generated, than 0 are generated.

So for example, the following will be executed when a tuple is sent to the engine after 30 seconds have expired since the last evaluation of this rule.

```
A(x,y) if B(x),C(x,y) : MSEC=30000;
```

or the following in which the 4 randomly chosen tuples are created from the join of tables B and C when x is greater than y.

```
A(x,y) if B(x),C(x,y) : x>y : choose(4);
```

System Tables

The rule engine provides some runtime information in the form of tables that can be used by the rule writer. The following table(s) are available and must use a matching declaration in the rules file:

- *`system peers(char[128] instanceID, char [64] topoRelationship);`* - this table contains the list of known rule engine peers and their topological relationships to the current rule engine. The valid values for `topoRelationship` are taken from the DSM Java enumerated type, `TopologicalRelationship`, and are as follows:
  - Neighbor, Reachable - these are always available independent of the application defined topology
  - Parent, Child, etc - these are available only if the application-defined topology uses them.
- *`system peer_changes(char[128] node, char[64] status);`* - this table provides information when there is a change in topology between the engines that are using the same namespace. Any change in topology triggers a new evaluation and this table will be non-empty. Status values are one of '*added*' or '*removed*'.

The *peers* table makes it easy to send information through transport tuples and identify targets by topological relationship. For example,

```
system peers(char[128] instanceID, char[64] relationship);
transport shared_observations(char[128] node, double opinion);
...
shared_observations(node,opinion)@src if my_observations(node, opinion),
peers(src,"Neighbor");
```

The *topology_change* table could be used to send messages that were not sent earlier

```
system topology_change(char[128] node, char[64] status);
persistent saved_messages(char[128] msg);
transport shared_messages(char[128] msg);
...
shared_messages(msg)@node if saved_messages(msg),topology_change(node,"added");
```

Action Tables

Action tables are used to define Java methods that are called at the end of rule evaluation, and have a minimum of two columns as follows:

```
action my_actions(int flags, char[128] methodName);
```

The 1st column holds flag bits that control action evaluation. The following are currently supported:

ASYNC bit, 0x00000001 (bit 0) - if this is set then the action will be called synchronously and rule evaluation will not complete until the action completes. The default is to call the method asynchronously so that rule evaluation may complete

before the the action does.

. The 2nd column defines the full name of the method, including the class name (e.g., `com.ibm.watson.dsm.samples.RuleActionExample.action1`). The signature of this method is then as follows:

```
public static void action1(IRuleEngine r, TupleState begin, TupleState end);
```

You may declare your action table to have additional columns. These extra column values will effectively change the expected method signature to allow these values to be passed to the method. For example,

```
action my_actions(int flags, char[128] methodName, int x, char[128] y, double z);
```

The system then expects the named method to have the following signature:

```
public static void action1(IRuleEngine r, TupleState begin, TupleState end,
Integer x, String y, Double z);
```

Rule Evaluation Algorithm

Upon acceptance of tuple state changes the DSMEngine class performs the following:

1. Make sure changes are only to Input or Transport tuples and that they are in the application name space and instance id of the engine, otherwise throw an exception.
2. Compute time since last rule evaluation.
3. Append new tuples (Input or Transport) to tables.
4. Copy thepersistent tables into temporary storage (previous persistent tables) that is referenced with the 'prev' operator
5. Clear the persistent tables.
6. Iterate over each BLOCK (if no BLOCKs are defined, all rules are in block 0), performing the following
    1. Evaluate persistent rules w/o CHOOSE modifiers,
    2. Evaluate persistent rules w/ CHOOSE modifiers, and
    3. Evaluate persistent rules w/ MSEC modifiers only on the first iteration for a block.
    4. Store tuples from 1, 2 and 3 above.
7. Goto step 6, if we added any new tuples to the persistent tables.
8. Collect tuples identified by transport rules, including those whose MSEC value is less than the time since the last evaluation. Apply choose modifiers, if present.
9. If there are 1 or more (new) transporttuples present, then send them to their destination(s).
10. Clear the Input tables.
11. Clear the transport tables.
12. Clear the previous persistent tables.
13. Evaluate action rules, if any.
14. Call actions in action tables.
15. Clear action tables.
16. Call listeners installed on the engine.

Comments

There are no comments.
Add a comment