Extending Production Rules with Modification Logic

Author: Peter Lin woolfel@gmail.com

Abstract

Production rules have been around for over 2 decades and has proven to be a solid foundation for a wide variety of software applications. One emergent area is handling aggregations within business applications. This paper proposes an extension to production rules semantics called modification logic. The paper will describe the problem and the proposed solution.

Acknowledgments

thanks to Daniel Selman for his feedback.

The Problem

Within financial risk management systems such as pre-trade compliance, the system needs to calculate aggregates in real-time. For example, all mutual funds must comply with 1940 Investment Act. The regulation require all retirement and mutual funds diversify the investments to limit risk. A typical diversification rule defines the boundaries in terms of portfolio weight.

The fund cannot exceed 15% weight in any sector

From an implementation perspective, we need to calculate the aggregate weight of each sector. This includes “what-if” and pending transactions, which are queued in the order management system. The fund manager or trader may conditionally execute the transaction if a pending transaction completes within a specified quantity of time. To do this effectively, the pre-trade compliance system must calculate the aggregates accurately, taking into consideration order cancellations. Some systems calculate the aggregate incrementally, while others must get the entire dataset. In some situations, getting the entire dataset is impractical, so incremental approach must be used. It's easier to see the difficulty with a sample business rule.

Rule: sector finance aggregate
If
The transaction is “buy”
The accountCode is “Growth Fund”
The sector of the stock is finance
Aggregate for the account exists
then
Add the transaction to the aggregate

On the surface, the rule looks simple, but there's hidden complexity. Say we implement the rule like this.

```
(defrule sector_finance_aggregate
        ?trans <- (Transaction
                (transactionType “buy”)
                (accountCode “Growth Fund”)
                (sectorCode “40”)
        )
        ?aggre <- (Aggregates
                (accountCode “Growth Fund”)
                (positions $?positions)
        )
        (test (not (member$ ?trans $?positions) ) )
=>
        (bind $?newpositions (union$ $?positions ?trans) )
        (modify ?aggre (positions $?newpositions) )
)
```

When we assert a transaction fact, it triggers the rule and adds the transaction to the aggregate. Another rule then checks the weight and determines if the transaction violates a rule. If the trader accidentally clicked send and decides to update the transaction, we run into a problem.

When the original transaction is retracted, the rule doesn't remove it from the aggregate. When we assert the updated transaction, it gets added again. This means the aggregate is wrong. Any calculation using the aggregate is also wrong, which results in a false positive or false negative for compliance violation. To make sure the aggregate is updated correctly, we

have to add a rule to remove the transaction from the aggregate.

```
(defrule update_sector_finance_aggregate
        ?trans <- (Transaction
                (transactionType “buy”)
                (accountCode “Growth Fund”)
                (sectorCode “40”)
                (status “update”)
        )
        ?aggre <- (Aggregates
                (accountCode “Growth Fund”)
                (positions $?positions)
        )
        (test (member$ $?positions ?trans) )
=>
        ;; remove the transaction from the list of positions
        (bind $?newpositions (remove$ ?trans $?positions) )
        (modify ?aggre (positions $?newpositions) )
)
```

The issue with using 2 rules to control the addition or removal of the transaction is tricky. If we just add the update rule, “sector_finance_aggregate” rule will still fire. This means we need to modify the transaction twice and update “sector_finance_aggregate” rule.

```
(defrule sector_finance_aggregate
        ?trans <- (Transaction
                (transactionType “buy”)
                (accountCode “Growth Fund”)
                (sectorCode “40”)
                (status ~”update”) ;; not update status
        )
        ?aggre <- (Aggregates
                (accountCode “Growth Fund”)
                (positions $?positions)
        )
        (test (not (member$ ?trans $?positions) ) )
=>
        (bind $?newpositions (union$ $?positions ?trans) )
```

```
        (modify ?aggre (positions $?newpositions) )
)
```

The application has to set the status to update, which will fire the update rule. Then we have to set the status so it isn't "update". Using this approach is inefficient and difficult to maintain. In a typical software development life cycle, the original implementation may take several months. The software may live for several years, so ease of maintenance is an important factor. Often a developer will move to another project, which means the person responsible for maintaining the application may not understand.

Even in the situation where the original developer maintains the rules, the ruleset may grow to thousands of rules. To address this issue, the rule language should treat modification as a first class citizen.

The Proposed Solution

If we extend production rule with the concept of modification logic, we can avoid writing multiple rules and eliminate the need to modify the fact twice. For example, we could write "sector_finance_aggregate" rule like this.

```
(defrule sector_finance_aggregate
        ?trans <- (Transaction
                (transactionType "buy")
                (accountCode "Growth Fund")
                (sectorCode "40")
        )
        ?aggre <- (Aggregates
                (accountCode "Growth Fund")
                (positions $?positions)
        )
        (test (not (member$ ?trans $?positions) ) )
=>
        (bind $?newpositions (union$ $?positions ?trans) )
        (modify ?aggre (positions $?newpositions) )
<<=
        ;; remove the transaction from the list of positions
        (bind $?newpositions (remove$ ?trans $?positions) )
        (modify ?aggre (positions $?newpositions) )
```

```
)
```

When a Transaction fact is modified, the rule engine will retract the fact. The modification logic highlighted in green defines the appropriate compensating actions. In this case, the action removes the transaction from the list and updates the aggregate. If we retract the aggregate, it will remove the transactions from the aggregate.

The modification logic is an optional part of the right-hand side (RHS) of the rule. We indicate the beginning of the modification logic with three characters “<<=”. Depending on the semantics of the rule language, a different keyword or symbol may be more desirable. The benefit of supporting modification logic is it reduces the number of rules and improves efficiency. The abstract syntax tree for defrule in EBNF format is the following.

```
<defrule-construct> ::= (defrule <rule-name> [<comment>]
                    [<declaration>]
                    <conditional-element>*
          =>
                    <action>*
          <<=
                    <action>* )

<action> ::= <expression>
```

A proof of concept of the modification logic was implemented in Jamocha's morendo branch. The code can be found in Jamocha's Subversion repository (http://jamocha.svn.sourceforge.net/viewvc/jamocha/morendo/). There are several challenges with implementing modification logic. The next section will go into the details of the implementation and lessons learned.

Implementation

Morendo uses JavaCC (Java Compiler Compiler) to generate the CLIPS parser. The change to the grammar was straight forward. The first step was to add “<<=” has a reserved word.

```
< LEFT_ARROW: "<<=" >
```

The second step was to update the syntax tree. Since the operations of the modification logic are actions, it only needed a minor change.

```
Defrule ruleBody() :
```

```
{
  Token exp = null;
  Token rulecomment = null;
  Defrule rule;
  List dec = new ArrayList();
  List conditions = new ArrayList();
  List actions = new ArrayList();
  List modactions = new ArrayList();
}
{
  exp = <IDENTIFIER> (LOOKAHEAD(<STRING_LITERAL>)rulecomment=<STRING_LITERAL>)?
  (LOOKAHEAD(<LBRACE><DECLARE>)ruleDeclaration(dec))?
((conditionElement(conditions))+)?
  arrow()
  ruleActions(actions)
  ( leftarrow()  ruleActions(modactions) )?
  {
    rule = new Defrule(exp.image);
    if (rulecomment != null) {
      rule.setComment(rulecomment.image);
    }
    rule.setRuleProperties(dec);
    Iterator itr = conditions.iterator();
    while (itr.hasNext()) {
      rule.addCondition( (Condition)itr.next() );
    }

    itr = actions.iterator();
    while (itr.hasNext()) {
      Object acn = itr.next();
      if (acn instanceof Function) {
        FunctionAction faction = new FunctionAction();
        faction.setFunction((Function)acn);
        rule.addAction(faction);
      } else if (acn instanceof Action) {
        rule.addAction( (Action)acn );
      }
    }
    itr = modactions.iterator();
    while (itr.hasNext()) {
      Object acn = itr.next();
      if (acn instanceof Function) {
        FunctionAction faction = new FunctionAction();
        faction.setFunction((Function)acn);
        rule.addModificationAction(faction);
      } else if (acn instanceof Action) {
```

```
            rule.addModificationAction( (Action)acn );
          }
        }
        dec.clear();
        conditions.clear();
        actions.clear();
        modactions.clear();
        exp = null;
        return rule;
      }
    }
```

The change to the grammar is highlighted in red. Note it reuses the existing function to parse the actions. In JavaCC, question mark “?” indicates the expressions within the parenthesis is optional. Once the rule is parsed, the actions are added to Defrule object highlighted in red.

Rather than put all the actions in one list, the rule object keeps a separate list for the modification logic. This was done for convenience and makes it easier for the rule compiler and rule analysis. The following methods were added to Rule interface.

```
    void addModificationAction(Action act);
    Action[] getModificationActions();
```

When the rule is compiled, the compiler checks the modification action list. If it is not null and greater than zero, the rule compiler uses MLTerminalNode instead of the normal TerminalNode. This was done for efficiency purposes. There's several draw backs to this decision. The first is that we can't dynamically “turn off” the modification logic for a rule at runtime. The second is it introduces another type of terminal node, which adds a little bit more complexity to the rule compiler. Jamocha's DefaultRuleCompiler creates the terminal node in a central method, so the change was minor. Creating a specialized node increases the cost of maintenance, so implementors needs to weigh the benefit against the long term cost.

```
protected TerminalNode createTerminalNode(Rule rl) {
          if (rl.getModificationActions() != null &&
rl.getModificationActions().length > 0) {
                    return new MLTerminalNode(engine.nextNodeId(), rl);
          } else if (rl.getNoAgenda() && rl.getExpirationDate() == 0) {
                    return new NoAgendaTNode(engine.nextNodeId(),rl);
          } else if (rl.getNoAgenda() && rl.getExpirationDate() > 0) {
                    return new NoAgendaTNode2(engine.nextNodeId(),rl);
          } else if (rl.getExpirationDate() > 0) {
                    return new TerminalNode3(engine.nextNodeId(),rl);
```

```
        } else {
                return new TerminalNode2(engine.nextNodeId(),rl);
        }
}
```

The main difference between a normal TerminalNode2 and MLTerminalNode is the retract method. A normal terminal node will look for the activation and remove it. Modification logic terminal node will remove the activation if it's still in the agenda. If the rule fired, it will add a new ModificationActivation.

```
public void retractFacts(Index inx, Rete engine, WorkingMemory mem) {
        long time = System.currentTimeMillis();
        Map tmem = (Map) mem.getTerminalMemory(this);
        LinkedActivation act = (LinkedActivation) tmem.remove(inx);
        if (act != null) {
                engine.getAgenda().removeActivation(act);
        } else {
                // we add a new ModificationActivation to the agenda
                ModificationActivation modact =
                        new ModificationActivation(this.theRule, inx);
                modact.setTerminalNode(this);
                tmem.put(modact.getIndex(), modact);
                engine.getAgenda().addActivation(modact);
        }
}
```

The main difference between the default Activation and ModificationActivation is the execute method. Instead of getting the actions from the rule, it gets the modification actions.

```
public void executeActivation(Rete engine) throws ExecuteException {
        remove(engine);
        try {
                getRule().setTriggerFacts(getFacts());
                Action[] actions = getRule().getModificationActions();
                for (int idx = 0; idx < actions.length; idx++) {
                        if (actions[idx] != null) {
                                actions[idx].executeAction(engine,
getFacts());
                        } else {
                                throw new
ExecuteException(ExecuteException.NULL_ACTION);
                        }
                }
        } catch (ExecuteException e) {
```

```
                throw e;
        }
}
```

The final change to the rule engine is the agenda. Normally, reset is implemented by retracting all the facts and then re-asserting them. If we don't modify the agenda, the engine will produce incorrect results when the user calls reset. The solution to this problem is to prevent the addition of activations during the retract phase of reset.

```
public void resetFacts() {
        try {
                // call startReset
                this.workingMem.getAgenda().startReset();

                List facts = new
ArrayList(this.workingMem.getDeffactMap().values());
                Iterator itr = facts.iterator();
                while (itr.hasNext()) {
                        Fact ft = (Fact) itr.next();
                        this.workingMem.retractFact(ft);
                }

                // call endReset
                this.workingMem.getAgenda().endReset();

                itr = facts.iterator();
                while (itr.hasNext()) {
                        Fact ft = (Fact) itr.next();
                        this.workingMem.assertFact(ft);
                }
        } catch (RetractException e) {
                log.debug(e);
        } catch (AssertException e) {
                log.debug(e);
        }
}
```

Before the facts are retracted, resetFacts method calls Agenda.startReset, which sets a boolean flag to true. This prevent addActivation from adding new activations.

```
public void addActivation(Activation actv) {
        if (!this.startReset) {
                // and then add the activation to the Module.
```

```
                    // the implementation should get the current focus from Rete
                    if (profAdd) {
                              addActivationWProfile(actv);
                    } else {
                              if (watch) {
                                        engine.writeMessage("=> " + actv.toPPString()
+ Constants.LINEBREAK, "t");
                              }
                              actv.getRule().getModule().addActivation(actv);
                    }
          }
}
```

Truth Maintenance System

In Clips, JESS and many other production rule engines, logical conditional elements (CE) provide truth maintenance functionality. The idea is that two facts are dependent on each other. When a rule uses logical CE and asserts new facts in the actions, the facts created by the rule are dependent on the fact that triggered the rule. For example:

```
(defrule logical_test
          (logical
                    (person
                              (age ?age&:(> ?age 20) );; age greater than 20
                    )
          )
=>
          (assert (purchase (alcohol true) ) )
)
```

If we assert a person fact with age greater than 20, the rule will create a new purchase fact and assert it. If we retract the person fact, the engine will retract the purchase fact. Logical CE provide one form of truth maintenance. There are several different forms of truth maintenance, but Logical truth maintenance (LTM) is more popular among existing production rule engines.

Although modification logic looks similar to LTM, they are different. With LTM, the engine will retract any facts created by the rule actions. In the case of changes, LTM does not define corrective actions when one or more facts matching the left-hand side change. Likewise, modification logic does not retract any facts created by the rule and does not maintain a truth

table. These two features are complementary. When used together, LTM and ML provide concise semantics for handling dependencies between facts.

Potential Side-Effects

This feature isn't without issues. Take the following example.

```
(defrule new_order
        (Order
                (orderId ?id)
                (status “new”)
        )
        ?item <- (Item
                (orderId ?id) ;; join on order
                (sku ?sku)
        )
        (Product
                (productSku ?sku) ;; join on the item
                (InStock true)
        )
=>
        (modify ?item (readyToShip true) )
<==
        (send-event “order updated ” ?id )
)
(defrule order_update_shipments
        ?order <- (Order
                (orderId ?id)
                (status “new”)
                (numberOfShipments ?shipments)
        )
        (Item
                (orderId ?id) ;; join on order
                (sku ?sku)
        )
        (exist
                (Product
                        (productSku ?sku) ;; join on the item
```

```
                                        (inWarehouse ?count&:(> ?count 10 ) ) ;; less than 10 instock
                        )
        )
=>
        (modify ?order (numberOfShipments (+ ?shipments 1) )
)
```

In a normal RETE engine, the activations are added to the agenda. When “order_update_shipments” fires, the engine retracts the Order fact, updates the shipment field and then asserts the fact again. Internally, Morendo would remove “(modify ?item (readyToShip true))” action from the agenda during retract and add the action again on assert. If the user assigns salience values, all activations from new_order rule could have already executed. In that case, when order_update_shipments rule fires, it would send the “order updated” message during the retract and execute “(modify ?item (readytoShip true))” again. Even if we modify the rule to check the Item.readyToShip field, new_order rule still modifies the fact and potentially causes a side-effect.

```
(defrule new_order
        (Order
                (orderId ?id)
                (status “new”)
        )
        ?item <- (Item
                (orderId ?id) ;; join on order
                (sku ?sku)
                (readyToShip false)
        )
        (Product
                (productSku ?sku) ;; join on the item
                (InStock true)
        )
=>
        (modify ?item (readyToShip true) )
<==
        (send-event “order updated ” ?id )
)
```

To avoid these types of issues, the actions performed in the right-hand side should be

idempotent. Meaning repeated executions of the actions does not negatively affect the state of the facts. One solution to this complex issue is to remove the order conditional element from new_order rule like this.

```
(defrule new_order
        ?item <- (Item
                (orderId ?id) ;; join on order
                (sku ?sku)
                (readyToShip false)
        )
        (Product
                (productSku ?sku) ;; join on the item
                (InStock true)
        )
=>
        (modify ?item (readyToShip true) )
<==
        (send-event “order updated ” ?id )
)
```

The problem with that solution is the rule applies to all orders regardless of the status. In some situations, removing a conditional element is acceptable. In situations where the order conditional element is needed, the side-effect may be unavoidable.

If the rule engine is running in reactive mode like run-until-halt in JESS or no-agenda in Morendo, Morendo would immediately execute “(modify ?item (readyToShip true))” when it matched. When “order_update_shipment” rule fires, it would cause new_order rule to execute both sets of actions again. This type of side effect is a serious concern. At this time, there is no formal proof or solution to this issue, since the problem is unbounded. At this time, the only recommended usage of modification logic is updating aggregates.

Conclusion

The addition of modification logic to Jamocha took approximately 2 weeks of coding, but the feature is quite powerful. It simplifies how production rule engines handle modifications to aggregates. The real benefit is it reduces the number of rules a developer has to write and reduces the cost of maintaining large rulesets that utilize aggregations. More research is needed to explore the usage of modification logic and identify common patterns. Another

important factor to consider is mixing modification logic with LTM within a single rule. Jamocha currently does not implement LTM, therefore users cannot use both within a single rule. Although combining both within a rule may be useful, rigorous research is needed to provide a thorough proof. Broadly applying techniques like modification logic should be used with care, due to potential side effects.

References

At this time, I could not find any references or prior art for modification logic. This doesn't mean this paper is the first time the topic has been explored. If anyone knows of prior art, please email and let me know.

http://en.wikipedia.org/wiki/Truth_maintenance_system
http://www.cis.temple.edu/~ingargio/cis587/readings/tms.html
http://www.jessrules.com/jess/docs/71/rules.html#logical_ce

Addendum 6/2/2010

According to Grindwork, their rising/falling edge feature was implemented in 2008 and has been in production for a year. Although I am aware of other people exploring these concepts and optimizations prior to Grindwork, I'm unable to find prior research papers on Acmqueue or other academic journals. This isn't meant to be definitive, since many of these same issues have existed for over two decades. It is also unclear if rising/falling edge in Grindwork is exactly the same as modification logic. At this time, Grindwork source code is closed, so no meaningful comparison can be made.

In large part, modification logic was inspired by OPSJ language and Paul Haley's backward chaining paper (http://haleyai.com/wordpress/2008/03/11/goals-and-backward-chaining-using-the-rete-algorithm/). OPSJ language (http://www.pst.com/opsjbro.htm) uses the same technique to collapse multiple rules into one rule. The implementation of modification logic is specific to production rule engines that implement RETE algorithm. The approach does not apply to engines that do not have conflict resolution or implement RETE.