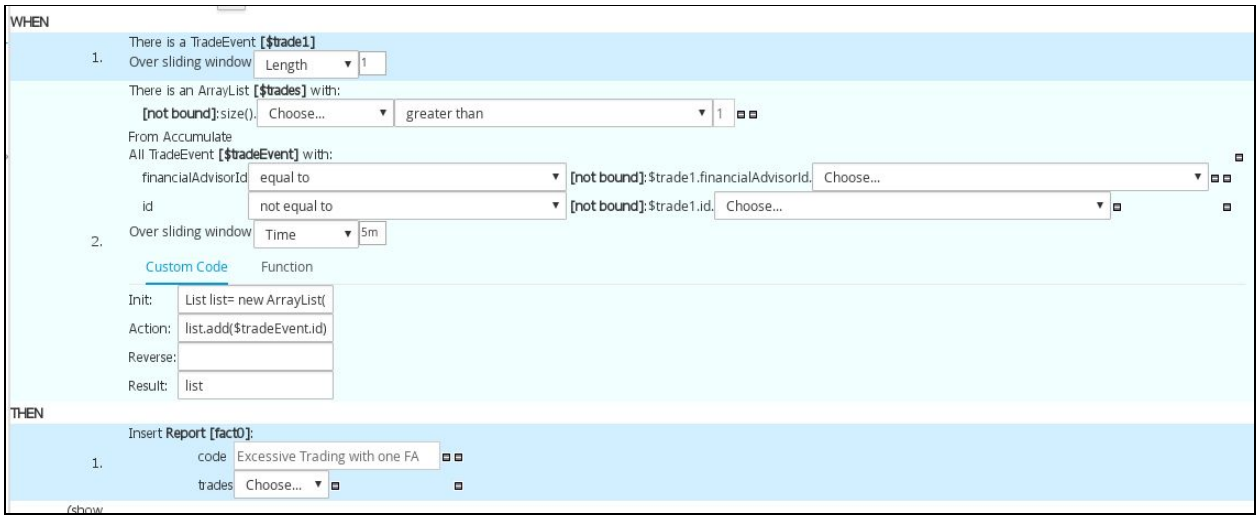
Session Clocks Code Walkthrough

Rules Project:

The ‘clocks-example’ project in the ‘drools-example-rules’ repo contains a rule that checks if more than one *TradeEvent* has happened in the last 5 minutes with the same ‘financialAdvisorId’ using a sliding windows. If this condition is met then a *Report* will be generated with all the *TradeEvent* ids that matched the condition and a code of “Excessive Trading with one FA”

clocks-example/src/main/resources/com/redhat/clocks_example/clocks-example.rdrl



Application Project:

In order to run this rule you must use a *KieBase* that is in Stream Mode and you need a Session Clock to keep track of time in the *KieSession* since the is using a sliding windows. In the ‘usecase-application’ project in the ‘drools-example-application’ repo there are two tests that run the rule in the clocks-example project. One uses the Realtime Session Clock and the other uses the Pseudo Session Clock. Both tests create *TradeEvent*s, configure the *KieBase* to use Stream Mode and the *KieSession* to use the Realtime or Pseudo Clock, and query the *KieSession* for the *Reports* generated to make sure the rules fired correctly.

Below are the tests written as Cucumber features. Cucumber is a readable way to define a test and its data. Each ‘Given’, ‘When’, and ‘Then’ statement is mapped to pieces of Java Code in the ‘CucumberSteps.java’ file shown in the next few images below.

usecase-application/src/test/java/com/redhat/usecase/features/realtime_example/realtimeClockExample.feature

```
Feature: Test Feature to demonstrate realtime clock usage

Scenario: Multiple trades in last 5 minutes with same FA
Given Trade Events:
|Id|Investor Id|Advisor Id| Price |Quantity|Symbol|Type    |
|1 | 1         |  3       | 20.00 | 10     | ANS  | Buy    |
|2 | 1         |  5       | 20.00 | 11     | ABS  | Sell   |
|3 | 2         |  2       | 20.00 | 15     | HNG  | Sell   |
|4 | 3         |  1       | 20.00 | 20     | TYI  | Buy    |
|5 | 3         |  1       | 20.00 | 20     | JKJ  | Buy    |
|6 | 3         |  1       | 20.00 | 20     | ANS  | Buy    |
When I run the realtime clock example
Then I expect the following Reports to be created:
|Code                              | Trades          |
|Excessive Trading with one FA     | 4,5,6           |
```

usecase-application/src/test/java/com/redhat/usecase/features/pseudo_example/pseudoClockExample.feature

```
Feature: Test Feature to demonstrate pseudo clock usage

Scenario: Multiple trades in last 5 minutes with same FA
Given Trade Events:
|Id|Investor Id|Advisor Id| Price |Quantity|Symbol|Type    |
|4 | 3         |  1       | 20.00 | 20     | TYI  | Buy    |
|5 | 2         |  1       | 20.00 | 20     | JKJ  | Buy    |
|6 | 3         |  1       | 20.00 | 20     | ANS  | Buy    |
When I run the pseudo clock example
Then I expect no reports to be created
```

The ‘Given’ statement code parses the table of data below the statement in the feature file and creates a list of *TradeEvents*.

usecase-application/src/test/java/com/redhat/usecase/CucumberSteps.java

```
@Given("^Trade Events:$")
public void trade_events(DataTable table) throws Throwable {

    for (Map<String, String> row : table.asMaps(String.class, String.class)) {
        TradeEvent trade = new TradeEvent();

        trade.setId(row.get("Id"));
        trade.setAccountId(row.get("Investor Id"));
        trade.setFinancialAdvisorId(row.get("Advisor Id"));
        trade.setPrice(Double.parseDouble(row.get("Price")));
        trade.setQuantity(Integer.parseInt(row.get("Quantity")));
        trade.setSymbol(row.get("Symbol"));
        trade.setType(row.get("Type"));

        if (row.get("Timestamp") != null) {

            trade.setTimestamp(parseDate(row.get("Timestamp")));

        }

        trades.add(trade);

    }
}
```

Below is the ‘When’ statement code for both tests. Each calls ‘fireAllRules’ in the *ReportService* class. This function in the class takes in the artifactId of the Maven Project the rule belongs to, the *TradeEvent* list created in the ‘When’ statement, the processing mode type, and the session clock type.

usecase-application/src/test/java/com/redhat/usecase/CucumberSteps.java

```
@When("^I run the pseudo clock example$")
public void i_run_the_pseudo_clock_example_rules() throws Throwable {
    reportsGenerated = reportService.fireAllRules("clocks-example", trades, "stream", "pseudo");
}
```

```
@When("^I run the realtime clock example$")
public void i_run_the_realtime_clock_example_rules() throws Throwable {
    reportsGenerated = reportService.fireAllRules("clocks-example", trades, "stream", "realtime");
}
```

The ‘fireAllRules’ function uses another class called *BRMSUtil* to created a *KieBase* using the processing mode and artifactId passed in as parameters. It then creates a Stateful *KieSession* from that *KieBase* that uses the session clock type passed in as a parameter. It then adds all the *TradeEvents* to the session, fires all rules, and queries for the reports generated. If the clock type is ‘pseudo’t hen it adds six minutes to the clock between inserts. This means the timestamp on every *TradeEvent* will be six minutes apart. If the clock type is ‘realtime’ then the timestamp of every *TradeEvent* will be populated with the current time. This means the timestamp on every *TradeEvent* will be milliseconds apart.

usecase-application/src/main/java/com/redhat/usecase/services/ReportService.java

```
public ArrayList<Report> fireAllRules(String artifactId, ArrayList<TradeEvent> trades, String mode,
        String clockType) {

    ArrayList<Report> reports = new ArrayList<Report>();
    TrackingAgendaEventListener listener = new TrackingAgendaEventListener();

    BRMSUtil brmsUtil = new BRMSUtil(artifactId, mode);
    KieSession ksession = brmsUtil.getStatefulSession(clockType);

    // ksession.addEventListener(listener);

    for (TradeEvent trade : trades) {

        ksession.insert(trade);
        if (clockType.equals("pseudo")) {
            SessionPseudoClock clock = ksession.getSessionClock();
            clock.advanceTime(6, TimeUnit.MINUTES);
        }
    }
    try {
        ksession.fireAllRules();
    } finally {
        QueryResults queryResults = (QueryResults) ksession.getQueryResults("reports generated");

        // System.out.println(listener.getMatchList());

        for (QueryResultsRow row : queryResults) {
            Report report = (Report) row.get("$report");
            reports.add(report);
        }
        if (ksession != null) {

            ksession.dispose();
        }
    }
    return reports;
}
```

usecase-application/src/main/java/com/redhat/usecase/utile/BRMSUtil.java

```
public BRMSUtil(String artifactId, String mode) {

    KieServices kServices = KieServices.Factory.get();

    ReleaseId releaseId = kServices.newReleaseId("com.redhat", artifactId, "LATEST");
    kContainer = kServices.newKieContainer(releaseId);

    // configure process mode type
    KieBaseConfiguration kBaseConfig = KieServices.Factory.get().newKieBaseConfiguration();
    if (mode.equals("stream")) {
        kBaseConfig.setOption(EventProcessingOption.STREAM);
    } else {
        kBaseConfig.setOption(EventProcessingOption.CLOUD);
    }

    kBase = kContainer.newKieBase(kBaseConfig);

    KieScanner kScanner = kServices.newKieScanner(kContainer);

}
```

```
 public KieSession getStatefulSession(String clockType) {
     KieSessionConfiguration kSessionConfig = KieServices.Factory.get().newKieSessionConfiguration();
     kSessionConfig.setOption(ClockTypeOption.get(clockType));
     return kBase.newKieSession(kSessionConfig, null);
 }
```

The timestamp of a *TradeEvent* will only be updated with the session clock time upon insert if the timestamp attribute is not mapped to a field on the *TradeEvent* object. In the current domain model for these projects we do have the timestamp attribute mapped to a field so we need to comment that out in order for these tests and our rule to work. The domain model is in the ‘usecase-domain’ project in the ‘drools-example domain’ repo. You will need to comment out the line of code above the *TradeEvent* class declaration.

usecase-domain/src/main/java/tradeEvent/TradeEvent.java

```
@org.kie.api.definition.type.Role(org.kie.api.definition.type.Role.Type.EVENT)
@org.kie.api.definition.type.Timestamp("timestamp")

public class TradeEvent implements java.io.Serializable {
    private String accountId;
    private String financialAdvisorId;
    private Double price;
    private int quantity;
    private String symbol;
    private String type;
    private Date timestamp;
    private String id;
```

Once the ‘fireAllRules’ function in the *ReportService* class is done it will return all the *Reports* created in the *KieSession*. The ‘Then’ statement code compares the *Reports* returned with the *Reports* expected (parsed from the feature file).

usecase-application/src/test/java/com/redhat/usecase/CucumberSteps.java

```
@Then("^I expect the following Reports to be created:$")
public void i_expect_the_following_Report_to_be_created(DataTable table) throws Throwable {
    ArrayList<Report> reports = new ArrayList<Report>();
    for (Map<String, String> row : table.asMaps(String.class, String.class)) {
        Report report = new Report();
        report.setCode(row.get("Code"));

        ArrayList<String> list = new ArrayList<String>(Arrays.asList(row.get("Trades").split(",")));
        report.setTrades(list);
        reports.add(report);
    }

    Assert.assertTrue(arraysMatch(reports, reportsGenerated));

}
```

You can run this feature files by locating *src/test/com/redhat/usecases/runners/RunRealtimeExample.java* and *src/test/com/redhat/usecases/runners/RunPseudoExample.java* These can be run as Junit Tests in your IDE.