Deployment of the Contract Compliant Checker: (User's Guide)

Carlos Molina–Jimenez[1] and Ioannis Sfyrakis[2]

[1] School of Computing Science, Newcastle University, UK,
`carlos.molina@ncl.ac.uk`
[2] Graduated MSc student, School of Computing Science, Newcastle University, UK,
`giannisfyrakis@gmail.com`

Abstract. This document is a walk through description of the deployment of version 1.1 of Contract Compliant Checker (CCC). The CCC is a software tool implemented at Newcastle University UK, in Java and Jboss Drools. It can be used for monitoring and enforcing of contract regulated interactions. Examples of such interactions are contractual agreements signed between buyers and sellers of goods and contractual agreements signed between providers of computing services and their consumers.
The CCC is loaded with the set of ECA rules that represent the contractual clauses of the contract under monitoring and deployed as a web service, for example, withing a trusted third party or within one of the business partners. As a web service, i) the CCC listens to events (RESTful messages) produced by the application under monitoring, ii) processes them using its ECA rules and iii) produces a response (a RESTful message) indicating that the event was found to be either contract compliant or non–contract compliant.
This document is aimed at potential users interested in locally deploying the CCC after downloading it from a public repository such as GitHub where it appears as the *carlos–molina/conch* project and trying it by means of running the provided examples.
The experiments discussed were conducted on a Mac platform, but users of Windows and Linux should be able to run them after minor adjustments.

1 Introduction

The CCC is a software tool that we have implemented at Newcastle University UK, in Java and Jboss Drools. It can be deployed as a contract monitor or alternatively, as a contract enforcer, By *monitor* we mean that the CCC acts as a passive observer of the interaction whereas by *enforcer* we mean that the CCC actively interferes with the interaction to prevent business partner to execute contractually illegal actions.

In both deployments, the CCC is provided with the set of Event Condition Action rules (ECA rules) that represent the contractual clauses of the contract of interest and deployed as a web service. It can be physically deployed withing a trusted third party or within one of the business partners. Its job is to listen to and process events and determine if the business partners are observing their contract clauses. We will use two examples to explain the operation of the CCC.

1.1 Monitoring Example

Let us assume that a buyer and store have agreed to trade under the following contract. This contract example is oversimplified and incomplete, yet it it good enough for explaining our ideas.

1. *The buyer can place a* ***buy request*** *with the store to buy an item.*
2. *The store is obliged to respond with either* ***buy confirmation*** *or* ***buy rejection*** *within 3 days of receiving the buy request.*
   (a) *No response from the store within 3 days will be treated as a buy rejection.*
3. *The buyer can either* ***pay*** *or* ***cancel*** *the buy request within 7 days of receiving a confirmation.*
   (a) *No response from the buyer within 7 days will be treated as a cancellation.*

Imagine that the two business partners decide to monitor their contractual interaction. A typical deployment of the CCC for addressing this question is shown in Fig. 1.

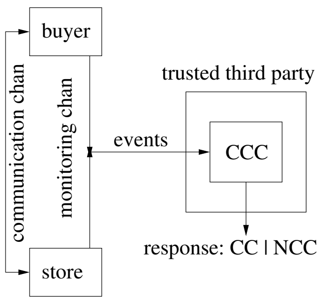


Fig. 1. The CCC deployed as a contract monitor.

In the figure, *buyer* and *store* represent the two parties involved in the contract. The *trusted third party* is a third party that operates the CCC which is assumed to be loaded with the ECA rules that represent the contractual clauses. As shown in the figure, the business partners use a communication channel (*communication chan*) for exchanging their business messages. In addition they use a monitoring channel (*monitoring chan*) for notifying events of interest to the CCC. Examples of *events* are events that notify of the execution of a contractual business operation such as the execution of a buy request operation by buyer or the execution of a confirmation operation by the store. Upon receiving an event(for example, *BuyRequest*), the CCC processes it to determine if the event is contract compliant (CC) or non–contract compliant (NCC). The results (*response: CC* | *NCC*) is sent to interested in parties such as the business partners.

1.2 Enforcement Example

Imagine service providers (providers for short) that offers services to clients under the stipulation of a contract. As a more specific example, let us think of a provider that sells pre–paid cards to clients that grant access to its service N (for example, five) times. Naturally, such a provider would need to deploy a mechanism to allow legal request reach its service and reject illegal ones (those that exceed the agreed number).

An potential solution to this problem is shown in Fig. 2, where the *client* and *provider* represent the business partners.

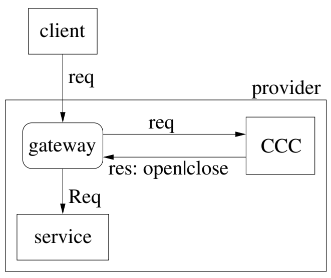


Fig. 2. The CCC deployed as an enforcer.

In this scenario, the the CCC is deployed as an enforcer—it opens or closes the *gateway* that grants access to the *service*.

1. The client sends a request (*req*) to the gateway.
2. The gateway intercepts *req* and forwards it to the CCC which is loaded with the ECA rules that represent the contract between the client and provider.
3. The CCC processes *req* and determines if the client has not exceeded yet his prepaid access (five requests in this example).
4. If *req* is declared legal by the CCC, it responds with *open*, otherwise it produces *close*.
5. The gateway forwards the request to the service only when the CCC responds with *open*.

2 Abstract Architecture of the CCC

We have implemented the CCC as a RESTful web service. Fig. 3 shows an abstract view of its architecture. In this section we will present and overview of the functionality of its components. Details about their implementations, deployments and configurations will be presented in subsequent sections.

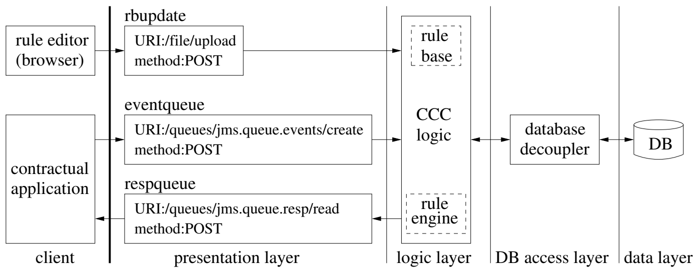


Fig. 3. Abstract architecture of the CCC.

As shown in the figure, conceptually, the CCC consists of four layers (**presentation, logic, DB access** and **data** layers) and is expected to interact with external entities that are represented by a **client** tier.

Client: The *client* represents the external entity to the CCC and consists of a *rule editor* (for example, a browser) and a *contractual application.* The *rule editor* is used by rule administrators for updating the rule base of the CCC. It offers editing facilities and means for sending the edited file to the CCC as a conventional HTTP POST request. The *contractual application* represents the contractual application under monitoring or enforcement. For instance, in Figs. 1, the *contractual application* corresponds to buyer and store. Similarly, regarding Fig. 2, the *contractual application* corresponds to the client.

Presentation layer: The CCC interacts with external entities through its presentation layer which we have implemented as three RESTful endpoints:

– A *rbupdate* (rule base update) point that accepts POST request sent by administrator to update the current rule base of the CCC.
– An *eventqueue* that accepts and stores events produced by the *contractual application* and sent as RESTful POST requests. Examples of events produced by the buyer–store contract example would be *BuyReq*, *BuyConf* and *BuyPay* that correspond, respectively, to the execution of buy request, buy confirmation and payment operations. To support portability of events, the *eventqueue* accepts events tagged with XML tags. For example, the *BuyConf* and *BuyPay* events are expected to be formatted as follows:

```
<event>
 <originator>store</originator>
 <responder>buyer</responder>
 <type>BuyConf</type>
 <status>success</status>
</event>

<event>
 <originator>buyer</originator>
 <responder>store</responder>
 <type>BuyPay</type>
 <status>success</status>
</event>
```

The *originator* specifies the business partner that initiated the execution of the operation; likewise, *responder* specifies the business partner that responded to the operation; finally, *status* specifies the outcome of the operation (we will elaborate on this parameter later). Thus the *BuyConf* event notifies that the execution of a buy request operation was originated by the store, responded by the buyer and completed in success. Similarly, the *BuyPay* event notifies that the execution of a payment operation completed in success and was originated by the buyer, responded by the store.

– An *respqueue* (response queue) where the CCC stores the results (contract compliant or non contract compliant) of the evaluation of the events. To support portability of results, the CCC produces results tagged with XML tags like in the following two examples:

```
<result>
 <contractcompliance>true</contractcompliance>
</result>

<result>
 <contractcompliance>false</contractcompliance>
</result>
```

The first example is the response to an event that was declared contract compliant (*true*) by the CCC. In contrast, the second example is the response to an event declared non contract compliant (*false*) by the CCC.

Logic layer: The *Logic layer* is represented by the *CCC logic* which consist of a *rule base*, *rule engine* and ancillary Java classes (not shown in the figure). The *rule base* represents the ECA rules that encode the contractual clauses. The *rule engine* represents the rule engine (for example, Drools engine) that upon arriving of events, triggers the execution of the corresponding rules.

DB layer: The *DB layer* represents a data base that is used by the CCC for storing permanent records (for example, events notified to the CCC) about the development of the contractual interaction.

DB access layer: The *DB access layer* is represented by a *database decoupler*. Its job is to hide from the designer the details of the communication between the CCC and the particular database technology used.

The functionality of the CCC as a web service can be summarised as follows:

1. The CCC retrieve and event from the *eventqueue*, sent by the *contractual application.*
2. The *rule engine* of the CCC processes the event with the help of the rules in the *rulebase.*
3. The CCC produces a response (RESTful message) that indicates if the event is contract compliant or not, and enqueues in the *respqueue.*

3 Deployment of Components

The deployment of the CCC is platform independent. The functionality of the current version has been tested in a Mac platform; namely in a MacBook Pro with Mac OS X Version 10.6.8, 2.4 GHz Intel Core 2 Duo and 4GB of memory. We will use this settings in our discussions.

The distribution software includes two independent working folders that can be downloaded from GitHub [1] (*carlos–molina/conch* project) and then used for creating two eclipse projects, respectively:

- *CCCRest–ear*: contains the software related to the presentation, logic, DB access and data layers (see Fig. 3).
- *CCCRestClient*: contains the software related to the client.

3.1 Database Deployment

The CCC needs a data base for permanently storing records about the contractual interaction. The current version uses a MySQL data base.

Free versions of MySQL data base servers can be downloaded from [2]. We use version 5.x.

Once the MySQL Server is deployed you need to create a database and initialise it with the following tables:

1. **roles table:** A table to store allowed roles for role players to take on. To be expanded as additional checks on roles are implemented (eg, some roles could be played by only one role player at a time). It contains the following fields:
   rolename –VARCHAR(50): The name of the role (eg, buyer, seller, whatever), not the name of the person/agent playing it (eg, not John Smith, or Paper Selling Company)
2. **statusoutcome table:** A table with all acceptable status outcomes, to avoid hardcoding them in the Java code, and in case the list is expanded/shortened. It contains the following fields:
   outcomedescription –VARCHAR(30): The description of the outcome. As of now, the table comes filled with the outcomes "Success", "TechFail", "BizFail", "InitFail", "InitSuccess".
3. **eventtypes table:** A table with all acceptable Event types–basically all legitimate business operations. To be expanded as additional checks on event types and business operations are implemented.
   eventtypename –VARCHAR(80): The name of the event type, eg PurchaseOrderSubmission. These names can be quite wordy, hence the length of the field.
4. **eventhistory table:** The actual history of the transaction. Note that as of now, it is not cancelled at the end of the execution of a transaction, as it makes sense to allow historical checks on activities of past transactions. The facility to cancel it on request would in any case be useful. It contains the following fields:

type –VARCHAR(80): Same as field eventtypename of eventtypes: name of event type.

timestamp –DATETIME: The time stamp of the event.

originator –VARCHAR(50): Same as field rolename of rolename: name of originating role player.

responder –VARCHAR(50): Same as field rolename of rolename: name of responding role player.

status –VARCHAR(30): Same as field outcomedescription of statusoutcomes: outcome status of event.

Notice that the with the current version of the CCC only the *eventhistory table* is needed. The other tables are needed for performing consistency check—a feature not currently supported.

3.2 Deployment of JBoss Drools

The current version of the CCC uses Drools version 5.4.0—the latest version of Drools, which can be downloaded for free from [3]. As explained in their documentations, the drools library are copied into a local working folder.

3.3 The drl Rule File

The CCC uses a *drl* (a text file that contains the ECA rules) file to create its rule base. In our examples, we use a *drl* file called *BuyerStoreContractEx.drl* and provided within the *CCCRest–ear* folder. It contains the ECA rules that encode the contract example between a buyer and seller discussed in Section 1.1.

With the current distribution, the *drl* file needs to be located under the *$JBOSS_HOME/standalone/Drools/upload* folder. This path is hardwired in the *uk.ac.ncl.conf.RulesFileEnum.java* class; we are planning to make this, more flexible in future releases.

To support the explanation of the rules, we will use a graphical representation of the contract written in BPMN choreography language [4], see Fig. 4.

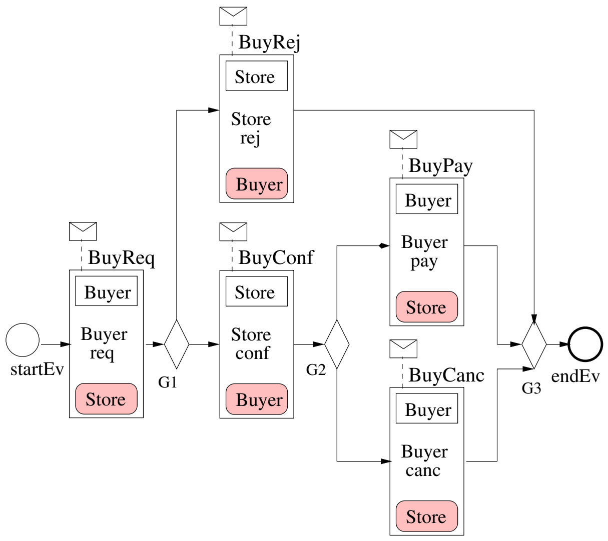


Fig. 4. Graphical view of the buyer–store contract.

The figure involves five events (*BuyReq, BuyRej, BuyConf, BuyPay, BuyCanc*) that correspond to the five business operations (*buy request, buy reject, buy confirmation, buy payment, buy cancellation*) included in the English text of the contract. We assume that the executions of all the five operations complete in success, therefore, we do not need rules for handling events that notify of executions that complete in failure. In this order, the *BuyerStoreContract.drl* file, includes only five contract–related rules (*rule Buy Request, rule Buy Reject, rule Buy Confirmation, rule Buy Payment, rule Buy Cancellation*[3]. In addition it includes an initialization rule (*rule Initialization*) and a reset rule (*Rule Reset*). The reset rule deals with reset event sent by the contractual application to signal the end of a contract run (execution path). In our example, the contractual application sends a reset event after sending *BuyRej, BuyPay* and *BuyCanc*. The main job of the reset rule is to grant the buyer a right to submit a *BuyReq* so that a new contract run can be started.

```
package BuyerStoreContractEx

# Import Java classes for EROP support
import uk.ac.ncl.erop.*;

# Global variables (persistent objects passed from outside)
global RelevanceEngine engine;
global EventLogger logger;

global RolePlayer buyer;
global RolePlayer seller;
global ROPSet ropBuyer;
```

[3] The rule names in the *BuyerStoreContract.drl* file provided in the software bundle might very slightly.

```
global ROPSet ropSeller;
global TimingMonitor timingMonitor;

global BusinessOperation buyRequest;
global BusinessOperation payment;
global BusinessOperation buyConfirm
global BusinessOperation buyReject;
global BusinessOperation cancelation;
global Responder responder;

# "rule Initialization": initialize the ROP sets for buyer and seller.
# This rule is launched only when the contract is set up.
# Initialy, the buyer has the right to submit a buy request.
rule "Initialization"
     when
        $e: Event (type == "init")
     then
        System.out.println("* Initialization when");
        #grant buyer's right to submit a buy request
        ropBuyer.addRight(buyRequest, seller, (String)null);
        System.out.println("* Initialization rule triggered ");
end

#rule Buy Request: deals with BuyReq events.
# removes buyer's right to submit buy request and
#imposes an obligation on the store to either reject
#or confirm the request.
rule "Buy Request"
    when
       $e: Event(type=="BUYREQ", originator=="buyer", responder=="store", status=="success")
       eval(ropBuyer.matchesRights(buyRequest))
    then
       # Remove buyer's right to place BuyReq
       ropBuyer.removeRight(buyRequest, seller);
       # impose seller's obligation to either accept or reject the request
       BusinessOperation[] bos = {buyConfirm, buyReject};
       ropSeller.addObligation("React To Buy Request", bos,buyer, 60,2);
       System.out.println("* Buy Request Received rule triggered");
       #The event is declared contract compliant and a response produced.
       responder.setContractCompliant(true);
end

#rule Buy Reject: deals with BuyRej events.
#removes store's obligation to react to buy request.
 rule "Buy Reject"
     when
        $e: Event(type=="BUYREJ", originator=="store", responder=="buyer", status=="success")
         eval(ropSeller.matchesObligations("React To Buy Request"));
     then
        System.out.println("* Buy Rejection");
        # Buyer's obligation is satisfied, remove it
        ropSeller.removeObligation("React To Buy Request", buyer);
        System.out.println("* Buy Rejection");
        System.out.println("* Buy Request Rejected rule triggered");
       #The event is declared contract compliant and a response produced.
        responder.setContractCompliant(true);
end

#rule Buy Confirmation: deals with BuyConf events.
#removes store's obligation to react to buy request and
#imposes buyer's obligation to pay.
rule "Buy Confirmation"
    when
       $e: Event(type=="BUYCONF", originator=="store", responder=="buyer", status=="success")
       eval(ropSeller.matchesObligations("React To Buy Request"));
    then
       # buyer's obligation is satisfied, remove it
       ropSeller.removeObligation("React To Buy Request", buyer);
       #impose buyer’s obligation to pay within 60 seconds, 2 min, 1 hour
       ropBuyer.addObligation(payment, seller, 60, 2, 1);
       ropBuyer.addRight(cancelation, seller, 60, 2, 1);
       System.out.println("* Buy Request Confirmation rule triggered");
       #The event is declared contract compliant and a response produced.
       responder.setContractCompliant(true);
end


#rule Buy Payment: deals with BuyPay events.
#removes buyer’s obligation to pay or cncel.
rule "Buy Payment"
    when
       $e: Event(type=="BUYPAY", originator=="buyer", responder=="store", status=="success")
       eval(ropBuyer.matchesObligations(payment))
    then
       #buyer’s obligation to pay is satiasfied, remove it.
       ropBuyer.removeObligation(payment, seller);
       ropBuyer.removeRight(cancelation, seller);
       System.out.println("* Payment rule triggered");
       #The event is declared contract compliant and a response produced.
       responder.setContractCompliant(true);
end

#rule Buy Cancellation: deals with BuyCanc events.
#removes buyer’s obligation to pay or cancel.
rule "Buy Cancellation"
    when
       $e: Event(type=="BUYCANC", originator=="buyer", responder=="store", status=="success")
       eval(ropBuyer.matchesRights(cancelation))
    then
       #buyer’s Obligation is satiasfied, remove it.
        ropBuyer.removeRight(cancelation, seller);
        ropBuyer.removeObligation(payment, seller);
        System.out.println("matches right cancellation: " + ropBuyer.toString());
        System.out.println("* Buy cancellation rule triggered");
        #The event is declared contract compliant and a response produced.
        responder.setContractCompliant(true);
end

#"rule Reset": deals with reset events sent by the contractual
#application to signal the end of a contract run (execution
#path). In our example, the contractual application sends a
#reset event after sending BuyRej, BuyPay and BuyCanc.
#The rule grants the buyer a right to submit a BuyReq
#so that a new contract run can be started.
rule "Reset"
    when
       $e: Event(type=="reset")
       eval(!ropBuyer.matchesRights(cancelation))
    then
       System.out.println("* reset when");
       ropBuyer.reset();
       ropSeller.reset();
       #clear business failures flag for each business operation
       buyRequest.setBusinessFailure(false);
       payment.setBusinessFailure(false);
       buyConfirm.setBusinessFailure(false);
       buyReject.setBusinessFailure(false);
       cancelation.setBusinessFailure(false);

       #grant buyer’s right to submit a BuyReq
       ropBuyer.addRight(buyRequest, seller, (String)null);
       System.out.println("* Reset rule triggered");

       #The event is declared contract compliant and a response produced.
       responder.setContractCompliant(true);
end
```

We use the symbol $\rightarrow$ to denote the *happened before relation*, thus $a \rightarrow b$ denotes that a happened before b. Them in accordance with Fig. 4, an example of a legal sequence of events would be: $BuyReq \rightarrow BuyConf \rightarrow BuyPay \rightarrow reset$ that would trigger the execution of four (without counting the initialization rule) rules, namely, *rule Buy Request, rule Buy Confirmation, rule Buy Payment, rule Reset*. Another valid sequence of events would be $BuyReq \rightarrow BuyRej \rightarrow reset$ which would trigger the execution of three rules only, namely *rule Buy Request, rule Buy Reject, rule Reset*

3.4 Deployment of JBoss Application Server

The current version of the CCC runs within a Jboss Application Server (AS) (see Fig. 5).

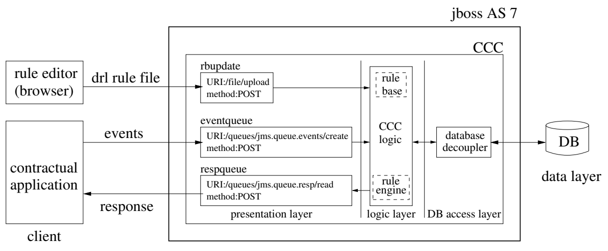


Fig. 5. Deployment of the CCC to Jboss AS7.

In our experiments we used AS version 7.1.0 which is freely available from [5]. As explained in its documentation, you can copy the AS software into a folder of your choice. For example, we copied it into */Users/ncmj2/JAVA–LIBRARIES/jboss–as–7.1.1.Final*.

More importantly, you need to set a *JBOSS_HOME* environment variable in your Linux shell. For example, since we used a *bash shell*, we included the following line in its *.bash_profile* file.

```
...
export JBOSS_HOME=/Users/ncmj2/JAVA-LIBRARIES/jboss-as-7.1.1.Final
...
```

3.5 Deployment of CCC Eclipse Project

For development purposes, it is convenient to deploy the CCC in a development environment. In our work we use eclipse (Indigo Service Release 2). The structure of the eclipse project (called *CCCRest–ear*) is shown in Fig. 6.

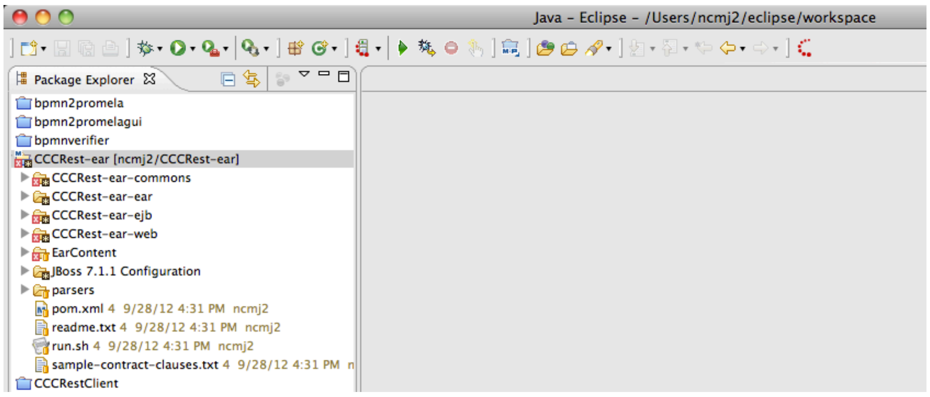

Java – Eclipse – /Users/ncmj2/eclipse/workspace
Package Explorer
bpmn2promela
bpmn2promelagui
bpmnverifier
CCCRest-ear [ncmj2/CCCRest-ear]
CCCRest-ear-commons
CCCRest-ear-ear
CCCRest-ear-ejb
CCCRest-ear-web
EarContent
JBoss 7.1.1 Configuration
parsers
pom.xml 4 9/28/12 4:31 PM ncmj2
readme.txt 4 9/28/12 4:31 PM ncmj2
run.sh 4 9/28/12 4:31 PM ncmj2
sample-contract-clauses.txt 4 9/28/12 4:31 PM
CCCRestClient

Fig. 6. CCC eclipse project.

4 Launch of Components

4.1 Launch of the AS

To launch the AS you need to execute the *run.sh* shell script:

```
#!/bin/sh
# run standalone Jboss 7.1.1
exec $JBOSS_HOME/bin/standalone.sh
  --server-config=standalone-full-ha.xml
```

This script is provided within the *CCCRest–ear* bundle (see Fig. 6).

To launch the AS, go to your *CCCRest–ear* working folder. In our example, it is located under */Users/ncmj2/eclipse/workspace*. Now type *./run.sh*, you should be able to see something similar to the following text:

```
{/Users/ncmj2}% pwd
/Users/ncmj2/eclipse/workspace/CCCRest-ear

{/Users/ncmj2}% ./run.sh
=========================================

  JBoss Bootstrap Environment

  JBOSS_HOME: /Users/ncmj2/JAVA-LIBRARIES/jboss-as-7.1.1.Final

  JAVA: java

  JAVA_OPTS: -d32 -client -Xms64m -Xmx512m -XX:MaxPermSize=256m -Djava.
net.preferIPv4Stack=true -Dorg.jboss.resolver.warning=true -Dsun.rmi.dg
c.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000 -Dj
boss.modules.system.pkgs=org.jboss.byteman -Djava.awt.headless=true -Dj
boss.server.default.config=standalone.xml

=========================================================================
==

17:06:46,332 INFO  [org.jboss.modules] JBoss Modules version 1.1.1.GA
17:06:46,599 INFO  [org.jboss.msc] JBoss MSC version 1.0.2.GA
```

```
17:06:46,653 INFO  [org.jboss.as] JBAS015899: JBoss AS 7.1.1.Final "Bro

...

17:06:58,668 INFO  [org.hibernate.tool.hbm2ddl.TableMetadata] (MSC serv
ice thread 1-4) HHH000108: Foreign keys: []
17:06:58,668 INFO  [org.hibernate.tool.hbm2ddl.TableMetadata] (MSC serv
ice thread 1-4) HHH000126: Indexes: [primary]
17:06:58,670 INFO  [org.hibernate.tool.hbm2ddl.SchemaUpdate] (MSC servi
ce thread 1-4) HHH000232: Schema update complete
17:06:58,696 INFO  [org.jboss.weld.deployer] (MSC service thread 1-4) J
BAS016008: Starting weld service for deployment CCCRest-ear.ear
17:06:59,566 INFO  [org.jboss.resteasy.cdi.CdiInjectorFactory] (MSC ser
vice thread 1-1) Found BeanManager at java:comp/BeanManager
17:06:59,843 INFO  [javax.enterprise.resource.webcontainer.jsf.config]
(MSC service thread 1-1) Initializing Mojarra 2.1.7-jbossorg-1 (2012022
7-1401) for context '/CCCRest-ear-web'
17:07:01,587 INFO  [org.jboss.web] (MSC service thread 1-1) JBAS018210:
 Registering web context: /CCCRest-ear-web
17:07:01,603 INFO  [org.jboss.as.server] (Controller Boot Thread) JBAS0
18559: Deployed "CCCRest-ear.ear"
17:07:01,612 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015951:
Admin console listening on http://127.0.0.1:9990
17:07:01,613 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874:
JBoss AS 7.1.1.Final "Brontes" started in 15580ms - Started 535 of 671
services (134 services are passive or on-demand)
```

At this stage, the instance of the JBoss AS is ready to accept deployments, of the CCC for example. Type *ctrl–c* to stop the AS.

4.2 Launch of the CCC Web Service

To ease the task of managing the CCC dependencies and its deployment into the AS, we use Maven facilities [6]. You might to need to download and deploy Maven in you do not have it in your local computer.

The *pom.xml* file that maven requires is provided in the CCC bundle (see Fig. 6). We have configured it with all the needed dependencies, such as *Drools, resteasy, hornet,* and *mysql.* The following lines show how we launched the CCC Web service (we assume that the database and the AS are already running).

```
{/Users/ncmj2}% pwd
/Users/ncmj2/eclipse/workspace/CCCRest-ear
{/Users/ncmj2}%
{/Users/ncmj2}% mvn clean package jboss-as:deploy
[INFO] Scanning for projects...
...
--------
[INFO] Reactor Build Order:
[INFO]
[INFO] CCCRest-ear
[INFO] CCCRest EAR: Commons Module
[INFO] CCCRest EAR: EJB Module
[INFO] CCCRest EAR: WAR Module
[INFO] CCCRest EAR: EAR Module
[INFO]
...
[INFO] Reactor Summary:
[INFO]
[INFO] CCCRest-ear ............. ..... SUCCESS [1.127s]
[INFO] CCCRest EAR: Commons Module ... SUCCESS [3.924s]
[INFO] CCCRest EAR: EJB Module ....... SUCCESS [3.595s]
[INFO] CCCRest EAR: WAR Module ....... SUCCESS [4.628s]
```

```
[INFO] CCCRest EAR: EAR Module ....... SUCCESS [27.995s]
[INFO] ----------------------------------------------
[INFO] BUILD SUCCESS
[INFO]

[INFO] Total time: 42.724s
[INFO] Finished at: Thu Jan 24 18:04:45 GMT 2013
[INFO] Final Memory: 25M/254M
[INFO] ----------------------------------
{/Users/ncmj2}%
```

The deployment of the CCC Web service within the AS is acknowleged by the AS which produces the following messages on its screen:

```
...
18:04:43,258 INFO  [org.jboss.weld.deployer] (MSC service thread 1-3) J
BAS016008: Starting weld service for deployment CCCRest-ear.ear
18:04:43,467 INFO  [org.jboss.resteasy.cdi.CdiInjectorFactory] (MSC ser
vice thread 1-2) Found BeanManager at java:comp/BeanManager
18:04:43,759 INFO  [javax.enterprise.resource.webcontainer.jsf.config]
(MSC service thread 1-2) Initializing Mojarra 2.1.7-jbossorg-1 (2012022
7-1401) for context '/CCCRest-ear-web'
18:04:44,985 INFO  [org.jboss.web] (MSC service thread 1-2) JBAS018210:
 Registering web context: /CCCRest-ear-web
18:04:45,030 INFO  [org.jboss.as.server] (management-handler-thread - 2
) JBAS018562: Redeployed "CCCRest-ear.ear"
18:04:45,031 INFO  [org.jboss.as.server] (management-handler-thread - 2
) JBAS018565: Replaced deployment "CCCRest-ear.ear" with deployment "CC
CRest-ear.ear"
```

At this stage the CCC is Web service ready for work. It is waiting for the arrival of events to the *eventqueue*.

5 Deployment of the Client

To test the functionality of the CCC Web service, we have implemented a client application that can play the role of the *contractual application* shown in the client tier of Fig. 3. The client application can be conveniently deployed as an eclipse project as shown in Fig. 6 where it appears as *CCCRestClient*.

The client produces and enqueues events in the *eventqueue* and retrieves events from the *respqueue*. We use it to demonstrate the use of the CCC in the monitoring of the contract example between the buyer and store discussed in Section 1.1.

Thus the client application mimics the behaviour of the buyer–store interaction of Fig. 1, in the sense that it produces the events involved in Fig. 4.

6 Launch of a Monitoring Example

To lunch the client, you need to run the *RestClient.java* as a Java application (see Fig. 7).

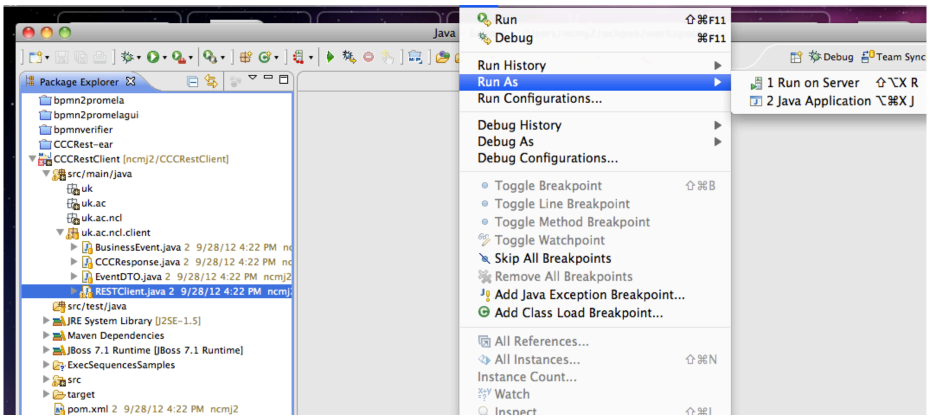
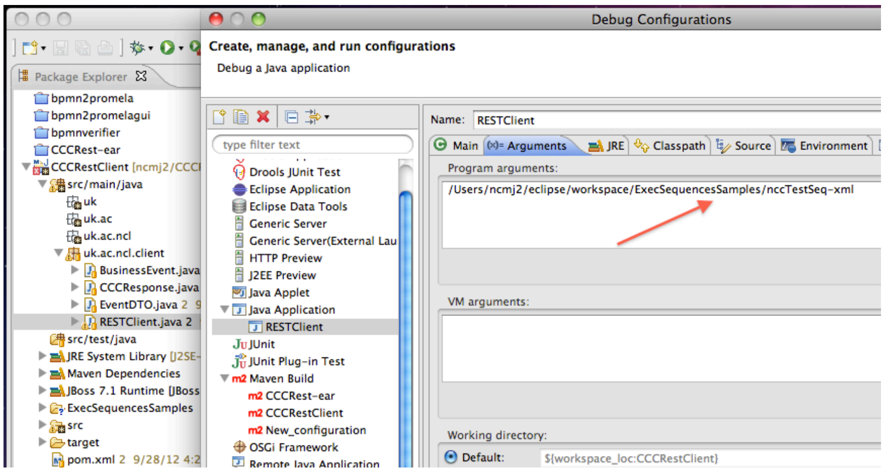


Fig. 7. Client launch as a Java application.

Once the client is launched, it looks for events stored in a local folder of your choice and enqueues them in the *eventqueue*. In our examples, we use the eclipse *Run Configurations* menu to indicate the client the location of the events. For example, imagine that the folder with the sequences of interest is *nccTestSeq-xml* and is located at
/Users/ncmj2/eclipse/workspace/ExecSequencesSamples/nccTestSeq-xml. The screen shot of Fig. 8 shows how the manipulation of eclipse *Run Configurations* to indicate this information.


Debug Configurations

Create, manage, and run configurations

Debug a Java application

Name: RESTClient

Program arguments:

/Users/ncmj2/eclipse/workspace/ExecSequencesSamples/nccTestSeq-xml

VM arguments:

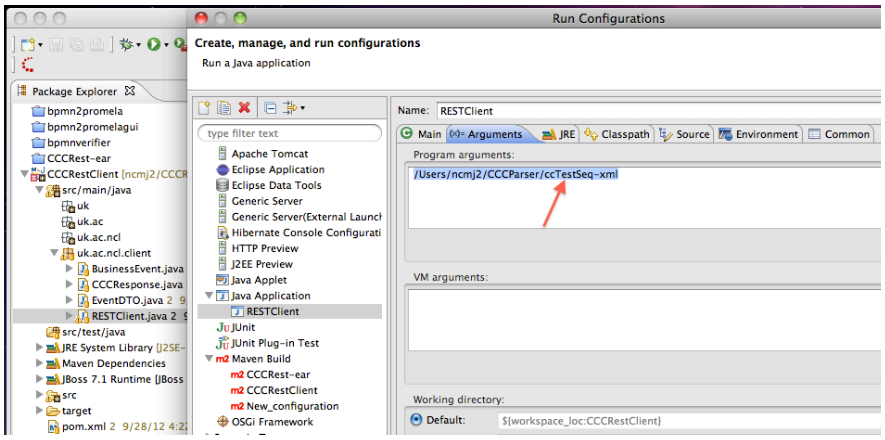
Working directory:

Default: ${workspace_loc:CCCRestClient}

Fig. 8. Specification of sequence location inside eclipse workspace.

6.1 Run with Contract Compliant Events

As shown in Fig. 9, for this experiment, we store the events in the */Users/ncmj2/CCCParser/ccTestSeq-mxl*.



Name: RESTClient

Program arguments: /Users/ncmj2/CCCParser/ccTestSeq-xml

VM arguments:

Working directory:
Default: ${workspace_loc:CCCRestClient}

Fig. 9. Specification of sequence location.

The current implementation of the client expects the events folder to contain one or more subfolders—one for each potential contract run. Consequently, each subfolder contains one or more **.xml* files where each of them represents an event. The following lines show the structure of the *ccTestSeq–mxl* folder that we use in one of our experiments.

The *ccTestSeq–mxl* folder contains two subfolders: *correctchoreExecSeq1* and *correctchoreExecSeq12*. The *correctchoreExecSeq1* subfolder contains four events: *event1.xml*, *event2.xml*, *event3.xml* and *event4.xml*. Similarly, tthe *correctchore-ExecSeq12* subfolder contains only three events: *event1.xml*, *event2.xml* and *event3.xml*.

```
{/Users/ncmj2}% pwd
/Users/ncmj2/CCCParser/ccTestSeq-xml
{/Users/ncmj2}%
{/Users/ncmj2}% ls -lR
total 0
drwxr-xr-x  6 ncmj2  staff  204  8 Nov 15:46 correctchoreExecSeq1
drwxr-xr-x  5 ncmj2  staff  170  8 Nov 15:46 correctchoreExecSeq12

./correctchoreExecSeq1:
total 32
-rw-r--r--  1 ncmj2  staff  130  8 Nov 15:46 event1.xml
-rw-r--r--  1 ncmj2  staff  131  8 Nov 15:46 event2.xml
-rw-r--r--  1 ncmj2  staff  130  8 Nov 15:46 event3.xml
-rw-r--r--  1 ncmj2  staff  127  8 Nov 15:46 event4.xml

./correctchoreExecSeq12:
total 24
```

```
-rw-r--r--  1 ncmj2  staff  130  8 Nov 15:46 event1.xml
-rw-r--r--  1 ncmj2  staff  130  8 Nov 15:46 event2.xml
-rw-r--r--  1 ncmj2  staff  127  8 Nov 15:46 event3.xml

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchore
                     ExecSeq1/event1.xml
<event>
  <originator>buyer</originator>
  <responder>store</responder>
  <type>BUYREQ</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchore
                     ExecSeq1/event2.xml
<event>
  <originator>store</originator>
  <responder>buyer</responder>
  <type>BUYCONF</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchore
                     ExecSeq1/event3.xml
<event>
  <originator>buyer</originator>
  <responder>store</responder>
  <type>BUYPAY</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchore
                     ExecSeq1/event4.xml
<event>
  <originator>reset</originator>
  <responder>reset</responder>
  <type>reset</type>
  <status>reset</status>
</event>
{/Users/ncmj2}%



{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchore
                     ExecSeq12/event1.xml
<event>
  <originator>buyer</originator>
  <responder>store</responder>
  <type>BUYREQ</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchore
                     ExecSeq12/event2.xml
<event>
  <originator>store</originator>
  <responder>buyer</responder>
  <type>BUYREJ</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchore
                     ExecSeq12/event3.xml
<event>
  <originator>reset</originator>
  <responder>reset</responder>
  <type>reset</type>
  <status>reset</status>
```

```
</event>
```

As we can see from the lines shown above, the events are XML tagged. The *correctchoreExecSeq1* folder contains the contract run that includes $BuyReq \to BuyConf \to BuyPay \to reset$, similarly the *correctchoreExecSeq12* folder contains the contract run that includes $BuyReq \to BuyRej \to reset$. In accordance with Fig. 4, both contract runs include only contract compliant events.

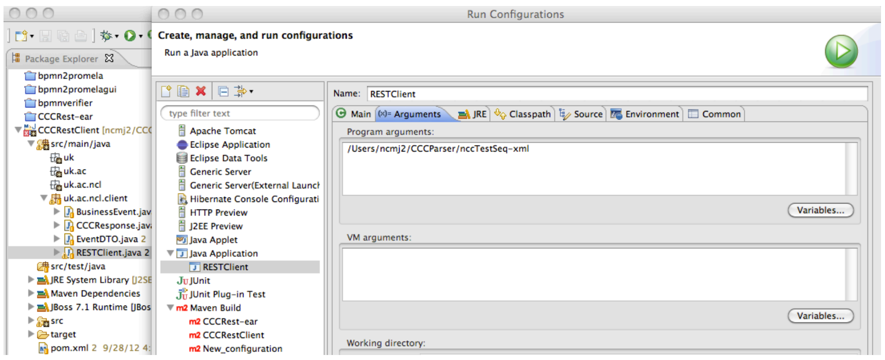
As shown in the lines below, a run of the client confirm our expectations: the events from both execution runs are declared contract compliant by the CCC.

```
log4j: ...

folder: /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchoreExecSeq1
filename: event1.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=buyer, responder=store, type=BUYREQ, status
=success]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------



folder: /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchoreExecSeq1
filename: event2.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=store, responder=buyer, type=BUYCONF, statu
s=success]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


folder: /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchoreExecSeq1
filename: event3.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=buyer, responder=store, type=BUYPAY, status
=success]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


folder: /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchoreExecSeq1
filename: event4.xml
```

```
-------- Begin Request to CCC service ----------
BusinessEvent [originator=reset, responder=reset, type=reset, status=
reset]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------




folder: /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchoreExecSeq12
filename: event1.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=buyer, responder=store, type=BUYREQ, status
=success]
-------- End Request to CCC service ----------


-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------



folder: /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchoreExecSeq12
filename: event2.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=store, responder=buyer, type=BUYREJ, status
=success]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


folder: /Users/ncmj2/CCCParser/ccTestSeq-xml/correctchoreExecSeq12
filename: event3.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=reset, responder=reset, type=reset, status=
reset]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------
```

6.2 Run with Non–Contract Compliant Events

As shown in Fig. 10, for this experiment, we store the events in the */Users/ncmj2/CCCParser/nccTestSeq-mxl*.



Name: RESTClient

Program arguments:
/Users/ncmj2/CCCParser/nccTestSeq-xml

VM arguments:

Fig. 10. Specification of sequence location.

The structure of the *nccTestSeq-mxl* folder is similar to the of *ccTestSeq-mxl*. The *correctchoreExecSeq1* folder contains the contract run that includes $BuyReq \rightarrow BuyConf \rightarrow BuyPay \rightarrow reset$, which contains, in accordance with Fig. 4, only contract compliant events. However, the folder *correctchore-ExecSeq12* contains a non contract compliant sequence, namely $BuyReq \rightarrow BuyPay \rightarrow reset$.

```
{/Users/ncmj2}% pwd
/Users/ncmj2/CCCParser/nccTestSeq-xml
{/Users/ncmj2}%
{/Users/ncmj2}% ls -lR
total 0
drwxr-xr-x  6 ncmj2  staff  204  8 Nov 16:17 correctchoreExecSeq1
drwxr-xr-x  5 ncmj2  staff  170  8 Nov 16:19 correctchoreExecSeq12

./correctchoreExecSeq1:
total 32
-rw-r--r--  1 ncmj2  staff  130  8 Nov 16:17 event1.xml
-rw-r--r--  1 ncmj2  staff  131  8 Nov 16:17 event2.xml
-rw-r--r--  1 ncmj2  staff  130  8 Nov 16:17 event3.xml
-rw-r--r--  1 ncmj2  staff  127  8 Nov 16:17 event4.xml

./correctchoreExecSeq12:
total 24
-rw-r--r--  1 ncmj2  staff  130  8 Nov 16:17 event1.xml
-rw-r--r--  1 ncmj2  staff  130  8 Nov 16:19 event2.xml
-rw-r--r--  1 ncmj2  staff  127  8 Nov 16:17 event3.xml

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchore
                     ExecSeq1/event1.xml
<event>
  <originator>buyer</originator>
  <responder>store</responder>
  <type>BUYREQ</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchore
                     ExecSeq1/event2.xml
<event>
  <originator>store</originator>
  <responder>buyer</responder>
  <type>BUYCONF</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchore
                     ExecSeq1/event3.xml
<event>
  <originator>buyer</originator>
  <responder>store</responder>
  <type>BUYPAY</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchore
                     ExecSeq1/event4.xml
<event>
  <originator>reset</originator>
  <responder>reset</responder>
  <type>reset</type>
  <status>reset</status>
</event>



{/Users/ncmj2}% pwd
/Users/ncmj2/CCCParser/nccTestSeq-xml
{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchore
                     ExecSeq12/event1.xml
<event>
  <originator>buyer</originator>
  <responder>store</responder>
  <type>BUYREQ</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchore
                     ExecSeq12/event2.xml
<event>
  <originator>store</originator>
  <responder>buyer</responder>
  <type>BUYPAY</type>
  <status>success</status>
</event>

{/Users/ncmj2}% cat /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchore
                     ExecSeq12/event3.xml
<event>
  <originator>reset</originator>
  <responder>reset</responder>
  <type>reset</type>
  <status>reset</status>
</event>
```

As we expected (see results below), a run of this experiment shows that the CCC declares the event *BuyPay* non–contract compliant.

```
log4j: ...
```

```
folder: /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchoreExecSeq1
filename: event1.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=buyer, responder=store, type=BUYREQ, status
=success]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


folder: /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchoreExecSeq1
filename: event2.xml


-------- Begin Request to CCC service ----------
BusinessEvent [originator=store, responder=buyer, type=BUYCONF, statu
s=success]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


folder: /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchoreExecSeq1
filename: event3.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=buyer, responder=store, type=BUYPAY, status=succ
ess]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------


folder: /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchoreExecSeq1
filename: event4.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=reset, responder=reset, type=reset, status=reset]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------



folder: /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchoreExecSeq12
filename: event1.xml
```

```
-------- Begin Request to CCC service ----------
BusinessEvent [originator=buyer, responder=store, type=BUYREQ, status=succ
ess]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------

folder: /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchoreExecSeq12
filename: event2.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=buyer, responder=store, type=BUYPAY, status=succ
ess]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>false</contractCompliant>
</result>
-------- End Response from CCC service ----------

folder: /Users/ncmj2/CCCParser/nccTestSeq-xml/correctchoreExecSeq12
filename: event3.xml

-------- Begin Request to CCC service ----------
BusinessEvent [originator=reset, responder=reset, type=reset, status=reset
]
-------- End Request to CCC service ----------

-------- Begin Response from CCC service ----------
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<result>
    <contractCompliant>true</contractCompliant>
</result>
-------- End Response from CCC service ----------
```

7 Implementation

Details about the technologies used in the implementation of the CCC and the client can be found in Chapter 6 of the MSC dissertation that originated this work [7]. UML class diagrams are also available.

8 Licence

The CCC is released under the Apache License, Version 2.0[8], which is available from Apache's web pages. Also, you can find a *txt* copy from our home page [9].

9 Implementation History

Table 1. BPMN verifier–implementation history.

Version	Date	Contributors	Key features
1.1	Aug 2012	Ioannis Sfykaris	Implementation of presentation and data access layer. Implementation of a client module for testing purposes.
1.0	Oct 2010	Massimo Strano	CCC logic implemented.

Acknowledgment

The first author was partially funded by EPSRC grant KTS-EP/H500332/1. Discussions with Gary Brown from Red Hat concerning the tool framework and SAVARA and constructive comments from Stuart Wheater from Arjuna Technologies, have been useful.

References

1. Inc., G.: Github distributed version control system. https://github.com (2012)
2. Corporation, O.: Mysql data base. http://www.mysql.com (2012)
3. JBoss: Drools. http://www.jboss.org/drools/
4. OMG: Documents associated with business process model and notation (bpmn) version 2.0. http://www.omg.org/spec/BPMN/2.0 (Jan 2011)
5. Community, J.: Jboss application server 7. http://www.jboss.org/jbossas/downloads/ (2013)
6. Foundation, T.A.S.: Apache maven project. http://maven.apache.org/ (2013)
7. Sfyrakis, I.: Implementing a contract compliance checker for monitoring contracts. http://homepages.cs.ncl.ac.uk/carlos.molina/home.formal (visited in Nov 2012 2012) MSc Dissertation Project, Aug 2012.
8. Foundation, T.A.S.: Apache license version 2.0, january. http://www.apache.org/licenses (2004)
9. Molina-Jimenez, C.: Carlos molina–jimenez home page. http://homepages.cs.ncl.ac.uk/carlos.molina (2012)