CArtAgO By Example

version: 2.0

Main author: aricci
Creation date: 20100801
Last Changes date: 20100820

DEIS, Università di Bologna, Italy

Contents

Chapter 1

Introduction

In the following we describe some main features of the artifact programming model on the one side and of the integration with agent platforms on the other side by making a sequence of simple examples, focussing each on one aspect. *Jason* is used as reference agent programming language, to program agents All the examples can be found in the `examples/bridges/jason/basic` folder.

Before going with the examples, it follows a sum up of some main key point about CArtAgO (more can be found in the annotated reference):

- *Workspaces* — A CArtAgO environment is given by one or multiple workspaces, possibly spread on multiple network nodes. Multiple workspaces can be running on the same node. By default each node has a `default` workspace In order to work inside a workspace an agent must join it. By default, when booted, an agent is automatically joined to the `default workspace`. Then, the same agent can join and work simultaneously in multiple workspaces.

- *Agents' action repertoire* — By working inside a CArtAgO environment, the repertoire of an agent's actions is determined by the set of artifacts available/usable in the workspace, in particular by the operations provided by such artifacts. There is one-to-one mapping between actions and operations: if there is an artifact providing an operation `myOp`, then each agent of the workspace – modulo security constraints – can perform an external action called `myOp`. Accordingly, by performing an external action, the action completes with success or failure if the corresponding operation completes with success or fails. Since the set of artifacts can be changed dynamically by agents (creating new

artifacts, disposing existing ones), the repertoire of actions is dynamic too.

- *Default artifacts* — By default, each workspace contains a basic set of predefined artifacts that provide core functionalities to the agents. In particular:
  - `workspace` artifact (`cartago.WorkspaceArtifact`) — provides functionalities to create, dispose, lookup, link, focus artifacts of the workspace. Also it provides operations to set roles and policies related to the RBAC security model.
  - `node` artifact (`cartago.NodeArtifact`) — provides functionalities to create new workspaces, to join local and remote workspaces
  - `blackboard` artifact, type `cartago.tools.TupleSpace` – provides a tuple space that agents can exploit to communicate and coordinate;
  - `console` artifact (`cartago.tools.Console`) — provides functionalities to print messages on standard output.

Specific points related to the *Jason*+CArtAgO integration (whose semantics, however, should be preserved also in the other integrations, when possible):

- *Mapping observable properties and events into beliefs* — by focussing an artifact, observable properties are mapped into agent's belief base. So each time an observable property is updated, the corresponding belief is updated too. Percepts related to observable events adopt the same syntax of belief-update events (so `+event(Params)`), however they are not automatically mapped into the belief base;

- *Java data-binding* — Java object model is used in CArtAgO as data model to represent and manipulate data structures. That means that operations' parameters, observable properties' and signals' arguments are either Java's primitive data types or objects. To work with CArtAgO, *Jason*'s data type has been extended to work also with objects – referenced by atoms with a specific functor – and a translation between primitive data types is applied. Translation rules: From CArtAgO to *Jason*:
  - boolean are mapped into boolean
  - int, long, float, double are mapped into doubles (NumberTerm)
  - String are mapped into String

- null value is mapped into an unbound variable
- arrays are mapped into lists
- objects in general are mapped by atoms `cobj_XXX` that work as object reference

From *Jason* to CArtAgO:

- boolean are mapped into boolean
- a numeric term is mapped into the smallest type of number which is sufficient to contain the data
- String are mapped into String objects
- structures are mapped into String objects
- unbound variables are mapped into output parameters (represented by the class `OpFeedbackParam`)
- lists are mapped into arrays
- atoms `cobj_XXX` referring to objects are mapped into the referenced objects

Jason agents do not share objects: each agent has its own object pool.

Chapter 2

Examples

2.1 Example 00 - Hello World

This first example is the classic hello world. It is composed by a single agent executing a `println` action to print on the console the message. The *Jason* MAS configuration file `hello-world` is the following:

```
MAS hello_world {
  environment:
  c4jason.CartagoEnvironment

  agents:
  hello_agent agentArchClass c4jason.CAgentArch;

  classpath: "../../../../lib/cartago.jar";
             "../../../../lib/c4jason.jar";
}
```

The declarations `environment: c4jason.CartagoEnvironment` and `agentArchClass c4jason.CAgentArch` are fixed, and specify that the MAS will exploit CArtAgO environments and that agents need to have a proper predefined architecture to work within such environment. The `classpath:` declaration is needed to include CArtAgO library (`cartago.jar`) and the specific *Jason* bridge (`c4jason.jar`) in the classpath.

The program spawns a single agent (`hello_agent`) whose task is to print on standard output the classic hello message. For that purpose it exploits the `println` operation provided by the `console` artifact. Source code of the `hello_agent` (in `hello_agent.asl`):

```
!hello.
```

```
+!hello : true
  <- .my_name(Name);
     println("Hello, world! by ",Name).
```

Highlights:

- by default an agent, when booting, joins the `default` workspace on current node – this can be avoided or controlled by specifying further parameters to the `environment: c4jason.CartagoEnvironment` declaration (see later)

- `println` is an operation provided by the `console` artifact, which is available by default in the `default` workspace: so the agent external action `println` is mapped onto the operation of this artifact. This is the case of operation execution without specifying the specific target artifact.

2.2 Example 01 - Artifact definition, creation and use

This example shows the basics about artifact creation and use, including observation. Two agents create, use and observe a shared artifact.

```
MAS example01_useobs {

  environment:
  c4jason.CartagoEnvironment

  agents:
  user user agentArchClass c4jason.CAgentArch #1;
  observer observer agentArchClass c4jason.CAgentArch #1;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The `user` agent creates a `c0` artifact of type `c4jexamples.Counter` and then uses it twice, executing the `inc` action (operation) two times:

```
!create_and_use.

+!create_and_use : true
  <- !setupTool(Id);
     inc;
     inc [artifact_id(Id)].

+!setupTool(C): true
  <- makeArtifact("c0","c4jexamples.Counter",[],C).
```

Hightlights:

- *Artifact creation* – To create the artifact, the agent exploits the `makeArtifact` action, provided by the `workspace` artifact. An empty list of parameters is specified, and the artifact id is retrieved, bound to the `C` variable.
- *Operation invocation with no target artifact specified* – operation invocation – i.e. action execution – can be done either specifying or not which is the specific target artifact providing the operation. No artifact is specified in the first `inc`: the artifact is automatically selected from the workspace. If there are no artifacts providing such action, the

action fails. if more than one artifact is found, first artifacts created by the agent itself are considered. If more than one artifact is found, one is selected non deterministically. Then, the rest of the artifacts are considered, and one is selected non deterministically.

- *Operation invocation with the target artifact specified* – The second time the `inc` is executed, the target artifact is specified. This can be done by adding the annotation `[artifact_id(Id)]`, where `Id` must be bound to the artifact identifier. Alternatively, the annotation `[artifact_name(Name)]` can be used, where `Name` must be bound to the logic name of the artifact.

- *Operation invocation with the target workspace specified* – As a further variant, the workspace identifier can be specified, instead of the target artifact, by means of the `wsp_id` annotation. Ex: `inc [wsp_id(WspID)]`

The `Counter` artifact is characterised by a single `inc` operation and a `count` observable property, updated by the operation. The operation also generates a `tick` signal.

```
package c4jexamples;

import cartago.*;

public class Counter extends Artifact {

  void init(){
    defineObsProperty("count",0);
  }

  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("count");
    prop.updateValue(prop.intValue()+1);
    signal("tick");
  }
}
```

Highlights

- *Artifact definition* – an artifact template can be implemented by defining a class – whose name corresponds to the artifact template name – extending the `Artifact` base class.

- *Artifact initialization* – the `init` method in artifact classes represents artifact constructor, useful to initialize the artifact as soon as it is

created. The actual parameter of the `init` method – in this case there are no parameters – can be specified when executing the `makeArtifact` action.

- *Operations* – Operations are implemented by methods annotated with `@OPERATION` and with `void` return parameter. Methods parameter corresponds to operations parameters.

- *Observable properties* – New observable properties can be defined by the `defineObsProp` primitive. In their most general form, an observable properties is represented by a tuple, with a functor and one or multiple arguments, of any type. In this case the `count` property has a single argument value, of integer type. To retrieve the reference to an observable property the `getObsProperty` primitive is provided, specifying the property name. Then `updateValue` methods can be used to change the value of the property.

- *Signals* – like observable properties, also signals can be tuple structures, with a functor and one or multiple arguments, of any type. In this case the `tick` signal generated by the operation has no argument. The primitive `signal` is provided to generate signals. It comes in two flavours:

  - `signal(String signalName, Object... params)` – generates a signal which is perceivable by all the agents that are observing the artifact (because they did a focus)
  - `signal(AgentId id, String signalName, Object... params)` – generates a signal which is perceivable only by the specified agent. The agent must be observing the artifact, anyway.

- *Atomicity* and *transactionality* – Operations are executed *transactionally* with respect to the observable state of the artifact. So no interferences can occur when multiple agents concurrently use an artifact, since the operations are executed *atomically*. Changes to the observable properties of an artifact are made observable only when:

  - the operation completes, successfully
  - a signal is generated
  - the operation is suspended (by means of an `await`, described in next examples)

If an operation fails, changes to the observable state of the artifact are rolled back.

Finally, an `observer` agent observes the counter and prints on standard output a message each time it perceives a change in `count` observable property or a `tick` signal:

```
!observe.

+!observe : true
  <- ?myTool(C);  // discover the tool
     focus(C).

+count(V)
  <- println("observed new value: ",V).

+tick [artifact_name(Id,"c0")]
  <- println("perceived a tick").

+?myTool(CounterId): true
  <- lookupArtifact("c0",CounterId).

-?myTool(CounterId): true
  <- .wait(10);
     ?myTool(CounterId).
```

Highlights

- *Artifact lookup* – agents can discover the identifier of an artifact by means of the `lookupArtifact` action provided by the `workspace` artifact, specifying either the logic name of the artifact to discover or its type (in this last case, if multiple artifacts are found, one is chosen non deterministically. In the example, if the observer agent executes a `lookupArtifact` before the artifact has been created (by the other agent), then the `lookupArtifact` fails and the repairing plan `-?myTool(...)` is executed.

- *Focus action* – agents can select which parts (artifacts) of the environment to observe by means of the `focus` action, provided by the `workspace` artifact, specifying the identifier of the artifact to focus. Variants:

  - `focus(ArtifactId id, IEventFilter filter)` – specifies a filter to select the percepts to receive

  - `focusWhenAvailable(String artName)` – focuses the specified artifact as soon as it is available in the workspace;

- *Observable properties - Beliefs mapping* – by focussing an artifact, artifact observable properties are mapped into the agent's belief base. So changes to the observable properties are detected as changes to the belief base. In the example: `+count(V)` triggering event. Beliefs related to observable properties are decorated with annotations that can be used to select the relevant/applicable plan, in particular:

  - `source(percept)`, `percept_type(obs_prop)` – define the percept type
  - `artifact_id(Id)`, `artifact_name(id,name)`, `artifact_type(id,type)`, `workspace(id,wspname)` – provide information about the source artifact and workspace. It is important to remark that, being beliefs, the value of observable properties can be accessed by means of test goals (e.g. `?count(X)`, when specifying context conditions, and so on.
  - *Percept mixing* – due to the belief base model adopted in *Jason*, beliefs (and so observable properties) with the same functor and argument are collapsed together, mixing the annotations.

- *Signals percept* – by focussing an artifact, signals generated by an artifact are detected as changes in the belief base – in the example: `+tick` – even if in this case the belief base is not changed. As in the case of observable properties, annotations that can be used to select the relevant/applicable plan, in particular:

  - `source(ArtifactId)`, `percept_type(obs_ev)` – define the percept type
  - `artifact_id(Id)`, `artifact_name(id,name)`, `artifact_type(id,type)`, `workspace(id,wspname)` – provide information about the source artifact and workspace.

2.3 Example 02 - Action Failure

This example is a simple variation of the previous one, to show action failure. As in the previous case, two agents create, use and observe a shared artifact, in this case a *bounded counter*:

```
MAS example01b_useobs {

  environment:
  c4jason.CartagoEnvironment

  agents:
  user2 agentArchClass c4jason.CAgentArch #1;
  observer agentArchClass c4jason.CAgentArch #1;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The counter used by the agents has the following code:

```
public class BoundedCounter extends Artifact {
  private int max;

  void init(int max){
    defineObsProperty("count",0);
    this.max = max;
  }
  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("count");
    if (prop.intValue() < max){
      prop.updateValue(prop.intValue()+1);
      signal("tick");
    } else {
      failed("inc failed","inc_failed","max_value_reached",max);
    }
  }
}
```

Highlights:

- *failed primitive* – Differently from the non-bound case, in this case the `inc` action fails if the count already achieved the maximum value, specified as a parameter of `init`. To specify the failure of an operation the `failed` primitive is provided:
  - `failed(String failureMsg)`

- `failed(String failureMsg, String descr, Object... args)`

  An action feedback is generated, reporting a failure msg and optionally also a tuple `descr(Object...)` describing the failure.

Then, the `user2` agent creates a bounded counter with 50 as bound and tries to increment it 100 times: as soon as the maximum value is reached, the action `inc` fails and a repairing plan is executed:

```
!create_and_use.

+!create_and_use : true
  <- !setupTool(Id);
     !use(Id).

+!use(Counter)
  <- for (.range(I,1,100)){
      inc [artifact_id(Counter)];
     }.

-!use(Counter) [error_msg(Msg),inc_failed("max_value_reached",Value)]
  <- println(Msg);
     println("last value is ",Value).

+!setupTool(C): true
  <- makeArtifact("c0","c4jexamples.BoundedCounter",[50],C).
```

Highlights:

- *Failure info* – on the *Jason* side, feedback information generated by `failed` on the artifact side are included in annotations in the repairing plan. In particular:
  - `error_msg(Msg)` contains the failure message;
  - the description tuple is directly included as annotation (`inc_failed(...)` in the example.

2.4 Example 03 - Operations with output parameters (i.e. actions with feedbacks)

Operations can have output parameters, i.e. parameters whose value is meant to be computed by the operation execution. On the agent side such parameters are managed as *action feedbacks*. At the API level, output parameters are represented by the class `OpWithFeedbackParam<ParamType>`, where `ParamType` must be the specific type of the output parameter. The class provides then a `set` method to set the output parameter value.
In the following example, an agent creates and uses a `Calc` artifact, by executing operations with output parameters:

```
MAS example03_output_param {

  environment:
  c4jason.CartagoEnvironment

  agents:
  calc_user agentArchClass c4jason.CAgentArch;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The `Calc` used by the agents has the following code:

```
public class Calc extends Artifact {

  @OPERATION
  void sum(double a, double b, OpFeedbackParam<Double> sum){
    sum.set(a+b);
  }

  @OPERATION
  void sumAndSub(double a, double b, OpFeedbackParam<Double> sum,
                                     OpFeedbackParam<Double> sub){
    sum.set(a+b);
    sub.set(a-b);
  }
}
```

The source code of the agent follows:

```
!use_calc.
```

```
+!use_calc
  <- makeArtifact("myCalc","c4jexamples.Calc",[]);
     sum(4,5,Sum);
     println("The sum is ",Sum);
     sumAndSub(0.5, 1.5, NewSum, Sub);
     println("The new sum is ",NewSum," and the sub is ",Sub).
```

Highlights:

- On the agent side output parameters are denoted by bound variables, which are bound with operation execution.
- An operation can have any number of output parameters

2.5 Example 04 - Operations with guards

When defining an operation, a *guard* can be specified as a condition that must be verified to start operation execution, otherwise such execution is suspended. This can be done by including a `guard` attribute in the `@OPERATION` annotation, specifying the name of the boolean method (guard method) – annotated with `@GUARD`, representing the condition to be tested. Guard methods are called passing the same parameters of the guarded operation (so they must declare the same parameters). Typically guard methods do checks on the value of internal and observable state of the artifact, without changing it.

Operations with guards are useful to realise artifacts with synchronisation functionalities. In the following example, guards are used to implement a *bounded buffer* artifact in a producers-consumers architecture.

```
MAS example04_prodcons {

    environment:
    c4jason.CartagoEnvironment

    agents:
    producer agentArchClass c4jason.CAgentArch #10;
    consumer agentArchClass c4jason.CAgentArch #10;

    classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

Ten producers agents and ten consumers agents exchange information items by exploiting the bounded buffer. Guarded operations allow for realising a simple coordinated behaviour, such that consumers' get action is suspended if the buffer is empty, and producers' put action is suspended if the buffer is full. Bounded buffer code:

```
public class BoundedBuffer extends Artifact {

  private LinkedList<Object> items;
  private int nmax;

  void init(int nmax){
    items = new LinkedList<Object>();
    defineObsProperty("n_items",0);
    this.nmax = nmax;
  }
```

```
  @OPERATION(guard="bufferNotFull")
  void put(Object obj){
    items.add(obj);
    getObsProperty("n_items").updateValue(items.size());
  }

  @OPERATION(guard="itemAvailable")
  void get(OpFeedbackParam<Object> res){
    Object item = items.removeFirst();
    res.set(item);
    getObsProperty("n_items").updateValue(items.size());
  }

  @GUARD
  boolean itemAvailable(OpFeedbackParam<Object> res){
    return items.size() > 0;
  }

  @GUARD
  boolean bufferNotFull(Object obj){
    return items.size() < nmax;
  }
}
```

Producers code:

```
item_to_produce(0).

!produce.

+!produce: true <-
  !setupTools(Buffer);
  !produceItems.

+!produceItems : true <-
  ?nextItemToProduce(Item);
  put(Item);
  !!produceItems.

+?nextItemToProduce(N) : true
 <- -item_to_produce(N);
    +item_to_produce(N+1).

+!setupTools(Buffer) : true <-
```

```
  makeArtifact("myBuffer","c4jexamples.BoundedBuffer",[10],Buffer).

-!setupTools(Buffer) : true <-
  lookupArtifact("myBuffer",Buffer).
```

Consumers code:

```
!consume.

+!consume: true
  <- ?bufferReady;
     !consumeItems.

+!consumeItems: true
  <- get(Item);
     !consumeItem(Item);
     !!consumeItems.

+!consumeItem(Item) : true
  <- .my_name(Me);
     println(Me,": ",Item).

+?bufferReady : true
  <- lookupArtifact("myBuffer",_).
-?bufferReady : true
  <-.wait(50);
     ?bufferReady.
```

Highlights:

- *Operation execution resume* – When an agent executes a guarded operation whose guard is false, the operation execution is suspended until the guard is evaluated to true.
- *Mutual exclusion* – Mutual exclusion and atomicity are enforce, anyway: a suspended guarded operation is reactivated and executed only if (when) no operations are in execution.

2.6 Example 05 - Structured Operations

In order to realise complex operations, a family of primitives (called `await`) is provided to *suspend* the execution of an operation until some specified condition is met, breaking the execution of an operation in multiple transactional steps. By suspending the execution of an operation, other operations can be invoked before the current one is terminated. When the specified condition holds and no operations are in execution, the suspended operation is resumed.

Complex operations which can be implemented by using this mechanism include:

- long-term operations which need not to block the use of the artifact;
- concurrent operations i.e. operations whose execution must overlap, which are essential for realising coordination mechanisms and functionalities.

In the following example, two agents share and concurrently use an artifact, which provides an operation using this mechanism.

```
MAS example05_complexop {

    environment:
    c4jason.CartagoEnvironment

    agents:
    complexop_userA agentArchClass c4jason.CAgentArch;
    complexop_userB agentArchClass c4jason.CAgentArch;

    classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The artifact used by the two agents has the following code:

```
public class ArtifactWithComplexOp extends Artifact {

  int internalCount;

  void init(){
    internalCount = 0;
  }

  @OPERATION void complexOp(int ntimes){
    doSomeWork();
```

```
    signal("step1_completed");
    await("myCondition", ntimes);
    signal("step2_completed",internalCount);
  }

  @GUARD boolean myCondition(int ntimes){
    return internalCount >= ntimes;
  }

  @OPERATION void update(int delta){
    internalCount+=delta;
  }

  private void doSomeWork(){}

}
```

In `complexOp` first we do some work, then we generate a signal `step1_completed`, and after that, by means of `await`, we suspend the execution of the operation until the condition defined by the guard method `myCondition` – whose name (and parameters, if needed) are specified as parameters of the `await` primitive – holds. The effect is to suspend the execution of the operation until the value of `internalCount` is greater than or equal to the value specified by the `complexOp ntimes` parameter.

Besides `complexOp`, the `update` operation is provided to increment the internal counter. In the example one agent – `complexop_userA` – executes a `complexOp` and the other agent – `complexop_userB` – repeatedly execute `update`. The action and plan of the first agent is suspended until the second agent has executed a number of updates which is sufficient to resume the `complexOp` operation.
Here it is the `complexop_userA` source code:

```
!do_test.

@do_test
+!do_test
  <- println("[userA] creating the artifact...");
     makeArtifact("a0","c4jexamples.ArtifactWithComplexOp",[],Id);
     focus(Id);
     println("[userA] executing the action...");
     complexOp(10);
     println("[userA] action completed.").

+step1_completed
```

```
  <- println("[userA] first step completed.").

+step2_completed(C)
  <- println("[userA] second step completed: ",C).
```

It is worth noting that the agent reacts to `step1_completed` signal generated by the artifact, printing a message on the console, even if the `do_test` plan execution is suspended waiting for `complexOp(10)` action completion.

`complexop_userB` source code:

```
!do_test.

+!do_test
  <- !discover("a0");
     !use_it(10).

+!use_it(NTimes) : NTimes > 0
  <- update(3);
     println("[userB] updated.");
     !use_it(NTimes - 1).

+!use_it(0)
  <- println("[userB] completed.").

+!discover(ArtName)
  <- lookupArtifact(ArtName,_).
-!discover(ArtName)
  <- .wait(10);
     !discover(ArtName).
```

The agent simply executes for 10 times the `update` operation. By running the example it is possible to see the interleaving of the agent actions.

Highlights:

- *Concurrency* – the execution of the operations overlaps in time: however always only one operation step is in execution at a time, so no interferences can occur in accessing and modifying artifact state
- *Transactionality and Observability* – by executing `await`, all the changes to the observable properties done so far by the operation are committed.

2.7 Example 05a - Implementing coordination artifacts

Here we show an example of how to exploit structured operations to implement a coordination artifact, a simple *tuple space*, and its usage to solve the dining philosophers coordination problem. The `in` and `rd` operations (that corresponds to the `in` and `rd` Linda primitives) are easily implemented exploiting the `await` mechanism:

```
public class TupleSpace extends Artifact {

  TupleSet tset;

  void init(){
    tset = new TupleSet();
  }

  @OPERATION void out(String name, Object... args){
    tset.add(new Tuple(name,args));
  }

  @OPERATION void in(String name, Object... params){
    TupleTemplate tt = new TupleTemplate(name,params);
    await("foundMatch",tt);
    Tuple t = tset.removeMatching(tt);
    bind(tt,t);
  }

  @OPERATION void rd(String name, Object... params){
    TupleTemplate tt = new TupleTemplate(name,params);
    await("foundMatch",tt);
    Tuple t = tset.readMatching(tt);
    bind(tt,t);
  }

  private void bind(TupleTemplate tt, Tuple t){
    Object[] tparams = t.getContents();
    int index = 0;
    for (Object p: tt.getContents()){
      if (p instanceof OpFeedbackParam<?>){
        ((OpFeedbackParam) p).set(tparams[index]);
      }
      index++;
    }
  }
```

```
  @GUARD boolean foundMatch(TupleTemplate tt){
    return tset.hasTupleMatching(tt);
  }
}
```

(The description of `Tuple`, `TupleTemplate` and `TupleSet` classes is omitted). This is actually the implementation of the `blackboard` tuple space artifact available by default in any workspace.
It follows a solution to the dining philosophers problem using a tuple space:

```
MAS example05a_philo {

    environment:
    c4jason.CartagoEnvironment

    agents:
    waiter agentArchClass c4jason.CAgentArch;
    philo agentArchClass c4jason.CAgentArch #5;

    classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The MAS is composed by a waiter agent and five philosophers The waiter is responsible of preparing the environment, injecting the tuples representing the forks (five `fork(F)` tuples) and tickets (four `ticket` tuples), which allow for avoiding deadlocks.

```
philo(0,"philo1",0,1).
philo(1,"philo2",1,2).
philo(2,"philo3",2,3).
philo(3,"philo4",3,4).
philo(4,"philo5",4,0).

!prepare_table.

+!prepare_table
  <- for ( .range(I,0,4) ) {
       out("fork",I);
       ?philo(I,Name,Left,Right);
       out("philo_init",Name,Left,Right);
     };
     for ( .range(I,1,4) ) {
       out("ticket");
     };
     println("done.").
```

The philosophers repeatedly get a couple of forks, use them to eat, and then release them. Before taking the forks they must get a ticket, which is released then after releasing the forks.

```
!start.

+!start
  <- .my_name(Me);
     in("philo_init",Me,Left,Right);
     +my_left_fork(Left);
     +my_right_fork(Right);
     println(Me," ready.");
     !!living.

+!living
 <- !thinking;
    !eating;
    !!living.

+!eating
 <- !acquireRes;
    !eat;
    !releaseRes.

+!acquireRes :
  my_left_fork(F1) & my_right_fork(F2)
  <- in("ticket");
     in("fork",F1);
     in("fork",F2).

+!releaseRes:
  my_left_fork(F1) & my_right_fork(F2)
 <-  out("fork",F1);
     out("fork",F2);
     out("ticket").

+!thinking
  <- .my_name(Me); println(Me," thinking").
+!eat
  <- .my_name(Me); println(Me," eating").
```

Highlights:

- no one created the tuple space artifact, since it is already available in the workspace by default.

2.8 Example 06 - Internal operations and timed await: implementing a clock

Sometimes it is useful to implement operations that trigger the asynchronous execution of other operations inside the artifact, which are typically long-term. For instance: a clock artifact can have the `start` operation, triggering the execution of a long-term `counting` operation. Such operations are typically *internal*, i.e. not (necessarily) part of the usage interface, and are annotated with `@INTERNAL_OPERATION`. To trigger the execution of an internal operation the `execInternalOp` primitive is provided.
In the following example, an agent creates a clock and uses it.

```
MAS example06_clock {

  environment:
  c4jason.CartagoEnvironment

  agents:
  clock_user agentArchClass c4jason.CAgentArch;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The clock artifact has two usage interface operations – `start` and `stop` – and an internal operation `count`, triggered by `start`:

```
public class Clock extends Artifact {

  boolean counting;
  final static long TICK_TIME = 100;

  void init(){
    counting = false;
  }

  @OPERATION void start(){
    if (!counting){
      counting = true;
      execInternalOp("count");
    } else {
      failed("already_counting");
    }
  }
```

```
  @OPERATION void stop(){
    counting = false;
  }

  @INTERNAL_OPERATION void count(){
    while (counting){
      signal("tick");
      await_time(TICK_TIME);
    }
  }
}
```

Highlights:

- *Timed await* – `await_time` primitive belongs to the `await` primitives: it suspends the execution of the operation until the specified time (in milliseconds) has elapsed (from now).

  Like in the `await` case, by suspending the operation, the artifact is made accessible to agents for executing operations and possible changes to its observable state are committed and made observable.

The agent starts a clock, then reacts to ticks generated by it for a certain number of times, and finally stopping it.

```
!test_clock.

+!test_clock
  <- makeArtifact("myClock","c4jexamples.Clock",[],Id);
     focus(Id);
     +n_ticks(0);
     start;
     println("clock started.").

@plan1
+tick: n_ticks(10)
  <- stop;
     println("clock stopped.").

@plan2 [atomic]
+tick: n_ticks(N)
  <- -+n_ticks(N+1);
     println("tick perceived!").
```

Highlights:

- *Controllable processes* – these features make it possible to exploit artifacts also to implement controllable long-term processes, without the need to use agents for this purpose (e.g. clock agent).

2.9 Example 07 - Await with blocking commands: Implementing artifacts for I/O

In order to implement artifacts that provides I/O functionalities for interacting with the external world (e.g. network communication, user I/O, GUI, etc.), a further kind of `await` primitive is provided, accepting an object of type `IBlockingCommand` representing a command to be executed. The primitive suspends the execution of the operation until the specified command – which typically contains some kind of I/O and a blocking behaviour – has been executed.
In the following example, two agents communicate by means of two artifacts that function as network port, providing I/O network communication based on UDP sockets.

```
MAS example07_extcommand {

  environment: c4jason.CartagoEnvironment

  agents:
  sender agentArchClass c4jason.CAgentArch #1;
  receiver agentArchClass c4jason.CAgentArch #1;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The agent `sender` creates and uses its port to send two messages:

```
!send_info.

+!send_info : true
  <- makeArtifact("senderPort","c4jexamples.Port",[23000]);
     sendMsg("hello1","localhost:25000");
     sendMsg("hello2","localhost:25000").
```

The agent `receiver` creates and uses its own port to get the messages, using two different receiving styles:

```
!receive_msgs.

+!receive_msgs : true
  <- makeArtifact("receiverPort","c4jexamples.Port",[25000],Id);
     receiveMsg(Msg,Sender);
     println("received ",Msg," from ",Sender);
     focus(Id);
```

```
    startReceiving.

+new_msg(Msg,Sender)
  <- println("received ",Msg," from ",Sender).
```

The first message is received by means of a `receiveMsg` action, while the second as a signal `new_msg` generated by the artifact.

The `Port` artifact exploits the await with a blocking command to implement its functionalities:

```
public class Port extends Artifact {

  DatagramSocket socket;
  ReadCmd cmd;
  boolean receiving;

  @OPERATION
  void init(int port) throws Exception {
    socket = new DatagramSocket(port);
    cmd = new ReadCmd();
    receiving = false;
  }

  @OPERATION
  void sendMsg(String msg, String fullAddress) {
    try {
      int index = fullAddress.indexOf(':');
      InetAddress address = InetAddress.getByName(fullAddress.substring(
          0, index));
      int port = Integer.parseInt(fullAddress.substring(index + 1));
      socket.send(new DatagramPacket(msg.getBytes(),
          msg.getBytes().length, address, port));
    } catch (Exception ex) {
      this.failed(ex.toString());
    }
  }

  @OPERATION
  void receiveMsg(OpFeedbackParam<String> msg, OpFeedbackParam<String> sender) {
    await(cmd);
    msg.set(cmd.getMsg());
    sender.set(cmd.getSender());
  }

  @OPERATION
```

```
void startReceiving() {
  receiving = true;
  execInternalOp("receiving");
}

@INTERNAL_OPERATION
void receiving() {
  while (true) {
    await(cmd);
    signal("new_msg", cmd.getMsg(), cmd.getSender());
  }
}

@OPERATION
void stopReceiving() {
  receiving = false;
}

class ReadCmd implements IBlockingCmd {

  private String msg;
  private String sender;
  private DatagramPacket packet;

  public ReadCmd() {
    packet = new DatagramPacket(new byte[1024], 1024);
  }

  public void exec() {
    try {
      socket.receive(packet);
      byte[] info = packet.getData();
      msg = new String(info);
      sender = packet.getAddress().toString();
    } catch (Exception ex) {
    }
  }

  public String getMsg() {
    return msg;
  }

  public String getSender() {
    return sender;
  }
```

```
  }
}
```

The `ReadCmd` implements a blocking command – implementing the `IBlockingCmd` interface – containing in the `exec` method the command code, in this case receiving an UDP packet from a socket.
Highlights:

- *Command implementation* – typically the class implementing a command provides methods to check success and retrieve results after the command has been executed and the `await` unblocked.

2.10 Example 07a - Programming GUI as Artifacts

An important example of artifacts encapsulating I/O functionalities is given by GUI artifacts, i.e. artifacts functioning as GUI components, enabling the interaction between human users and agents. Such artifacts allow to use Swing to define the structure of a GUI; then, they allow for defining – on the one side – operations corresponding to user actions on the GUI, so handling specific GUI events. Such operations generates signals or change some observable events to trigger agents observing the GUI; on the other side, they provide operations that can be possibly used by agents to change the GUI.

In the following example, a `gui_tester` agent creates and uses a GUI artifact to interact with the user.

```
MAS example07a_gui {

  environment:
  c4jason.CartagoEnvironment

  agents:
  gui_tester agentArchClass c4jason.CAgentArch;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

To make it easier GUI artifact development, a `cartago.tools.GUIArtifact` base artifact is provided among the CArtAgO utility tools. The implementation of `cartago.tools.GUIArtifact` – which can be checked in CArtAgO source code – exploits `await` and blocking commands The base artifact provides basic functionalities to link GUI events to the artifact operations.

In the following, `MySimpleGUI` GUI artifact creates a simple GUI with a text field and a button. Some GUI events – pressing the button, key stroke in the text field, closing the window – are linked to some artifact's internal operations, which in turn generate observable events to agents.

```
public class MySimpleGUI extends GUIArtifact {

  private MyFrame frame;

  public void setup() {
    frame = new MyFrame();
```

```
  linkActionEventToOp(frame.okButton,"ok");
  linkKeyStrokeToOp(frame.text,"ENTER","updateText");
  linkWindowClosingEventToOp(frame, "closed");

  defineObsProperty("value",getValue());
  frame.setVisible(true);
}

@INTERNAL_OPERATION void ok(ActionEvent ev){
  signal("ok");
}

@INTERNAL_OPERATION void closed(WindowEvent ev){
  signal("closed");
}

@INTERNAL_OPERATION void updateText(ActionEvent ev){
  getObsProperty("value").updateValue(getValue());
}

@OPERATION void setValue(int value){
  frame.setText(""+value);
  getObsProperty("value").updateValue(getValue());
}

private int getValue(){
  return Integer.parseInt(frame.getText());
}

class MyFrame extends JFrame {

  private JButton okButton;
  private JTextField text;

  public MyFrame(){
    setTitle("Simple GUI ");
    setSize(200,100);
    JPanel panel = new JPanel();
    setContentPane(panel);
    okButton = new JButton("ok");
    okButton.setSize(80,50);
    text = new JTextField(10);
    text.setText("0");
    text.setEditable(true);
```

```
      panel.add(text);
      panel.add(okButton);
    }

    public String getText(){
      return text.getText();
    }

    public void setText(String s){
      text.setText(s);
    }
  }
}
```

Highlights:

- *Designing GUI artifacts* – a GUI artifact is defined by extending `GUIArtifact`, wrapping the definition and creation of the structure of the GUI – using the Swing API – and then linking/mapping Swing events into artifact's internal operations by using `linkXXXtoYYY` primitives.

The agent creates an instance of the artifact and reacts to user actions on the GUI:

```
!test_gui.

+!test_gui
  <-  makeArtifact("gui","c4jexamples.MySimpleGUI",[],Id);
      focus(Id).

+value(V)
  <- println("Value updated: ",V).

+ok : value(V)
  <-  setValue(V+1).

+closed
  <-  .my_name(Me);
      .kill_agent(Me).
```

In particular, the agent reacts to the pressing of the button by setting a new value in the GUI; it prints a message on the console as soon as a new value is observed; it shutdown as soon as the window is closed.

2.11 Example 08 - Linkability

Linkability is the mechanism that makes it possible to create interactions among artifacts, i.e. execute inter-artifacts operations. Besides the usage interface, an artifact can expose operations – to be tagged with `@LINK`. These operations are meant to be called by other artifacts. In order to allow an artifact A to execute operations over an artifact B, two options are provided:

- the artifact A must be explicitly *linked* to the artifact B by an agent, executing `linkArtifacts` action, specifying the name of an *output* port that the artifact A must expose. Then, operations of artifact A can execute operations of the linked artifact B by using the `execLinkedOp` primitive, specifying the *output* port where the linked artifact has been linked.
- without linking the two artifacts, an artifact A can execute operations over the artifact B by specifying in `execLinkedOp` the target identifier of the artifact B

In the following example, an agent creates and links together two artifacts. Then, it executes some operations of one artifact, the *linking* one, which in turns executes operations over the second one, the *linked* one:

```
MAS example08_linkability {

  environment:
  c4jason.CartagoEnvironment

  agents:
  linkability_tester agentArchClass c4jason.CAgentArch;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

Source code of the linkable artifact:

```
public class LinkableArtifact extends Artifact {

  int count;

  void init(){
    count = 0;
  }
```

```
  @LINK void inc(){
    log("inc invoked.");
    count++;
  }

  @LINK void getValue(OpFeedbackParam<Integer> v){
    log("getValue invoked");
    v.set(count);
  }
}
```

Highlights:

- *@LINK operations* – the semantics of linked operations is the same of normal operation.
- *Output parameters* – linked operations can contain also output parameters, as normal operations.

Source code of the linking artifact:

```
@ARTIFACT_INFO(
  outports = {
    @OUTPORT(name = "out-1")
  }
) public class LinkingArtifact extends Artifact {

  @OPERATION void test(){
      log("executing test.");
      try {
        execLinkedOp("out-1","inc");
      } catch (Exception ex){
        ex.printStackTrace();
      }
  }

  @OPERATION void test2(OpFeedbackParam<Integer> v){
      log("executing test2.");
      try {
        execLinkedOp("out-1","getValue", v);
        log("back: "+v.get());
      } catch (Exception ex){
        ex.printStackTrace();
      }
```

```
  }

  @OPERATION void test3(){
      log("executing test3.");
      try {
        ArtifactId id = makeArtifact("new_linked",
               "c4jexamples.LinkableArtifact", ArtifactConfig.DEFAULT_CONFIG);
        execLinkedOp(id,"inc");
      } catch (Exception ex){
        ex.printStackTrace();
      }
  }
}
```

The `test` and `test2` operations executes respectively the `inc` and `getValue` operation over the artifact linked to the `out-1` port. The operation `test3` instead creates an artifact and executes a linked operation directly using artifact identifier. Highlights:

- *Output ports* – output ports are declared in the `@ARTIFACT_INFO` annotation of the artifact class, `outports` attribute;
- *Linked operation execution* – the execution semantics is the same of normal operations. The `execOpLinked` primitive suspend the operation execution until the operation execution on the linked artifact has completed.

Finally, the agent source code:

```
!test_link.

+!test_link
  <- makeArtifact("myArtifact","c4jexamples.LinkingArtifact",[],Id1);
     makeArtifact("count","c4jexamples.LinkableArtifact",[],Id2);
     linkArtifacts(Id1,"out-1",Id2);
     println("artifacts linked: going to test");
     test;
     test2(V);
     println("value ",V);
     test3.
```

Highlights:

- *Linking artifacts* – `linkArtifacts`'s parameters include the identifier of the linking artifact, its outport and the identifier of the linked artifact.

2.12 Example 09 - Java data-binding

Since CArtAgO data model is based on Java object (POJO), a set of internal actions is provided on the agent side to create/manipulate Java objects. In particular:

- `cartago.new_obj(ClassName,ParamList,?ObjRef)` – instantiate a new object of the specified class, retrieving its reference
- `cartago.invoke_objObjRef,MethodName{(Params)},RetValue` – call a method, possibly getting the return value
- `cartago.invoke_objClassName,MethodName{(Params)},RetValue` – call a static method, possibly getting the return value

It follows a simple example:

```
MAS example09_java_data_binding {

  environment:
  c4jason.CartagoEnvironment

  agents:
  java_data_binding_tester agentArchClass c4jason.CAgentArch;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The agent source code:

```
!test_java_api.

+!test_java_api
  <- cartago.new_obj("c4jexamples.FlatCountObject",[10],Id);
     cartago.invoke_obj(Id,inc);
     cartago.invoke_obj(Id,getValue,Res);
     println(Res);
     cartago.invoke_obj("java.lang.System",currentTimeMillis,T);
     println(T);
     cartago.invoke_obj("java.lang.Class",
                   forName("c4jexamples.FlatCountObject"),Class);
     println(Class).
```

where the class `FlatCountObject` is defined as follows:

```
package c4jexamples;

public class FlatCountObject {

  private int count;

  public FlatCountObject(int v){
    count = v;
  }

  public FlatCountObject(){
    count = 0;
  }

  public void inc(){
    count++;
  }

  public void inc(int dv){
    count+=dv;
  }

  public int getValue(){
    return count;
  }

}
```

Highlights:

- *No sharing* – Java objects are not meant to be shared, each agent has its own Java library, managing its own Java objects
- *Object references* – Object references are kept track by means of atoms with a specific signature. When used outside the Java related internal actions, they are treated as normal atoms. When used in Java related internal actions, they refer to objects.
- *Null value* – Underscore (_) is used to represent the null value.

2.13 Example 10 - Working with Multiple Workspaces

In the following example, creates two workspaces, joins both and prints messages using different console artifacts, and then uses internal actions to set the current workspace.

```
MAS example10_workspaces {

    environment:
    c4jason.CartagoEnvironment

    agents:
    wsp_tester agentArchClass c4jason.CAgentArch;

    classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The agent source code:

```
!test_wsp.

+!test_wsp
  <- ?current_wsp(Id0,Name,NodeId);
     println("current workspace ",Name," ",NodeId);
     println("creating new workspaces...");
     createWorkspace("myNewWorkspace1");
     createWorkspace("myNewWorkspace2");
     joinWorkspace("myNewWorkspace1",WspID1);
     ?current_wsp(_,Name1,_);
     println("hello in ",Name1);
     makeArtifact("myCount","c4jexamples.Counter",[],ArtId);
     joinWorkspace("myNewWorkspace2",WspID2);
     ?current_wsp(_,Name2,_);
     println("hello in ",Name2);
     println("using the artifact of another wsp...");
     inc [artifact_id(ArtId)];
     cartago.set_current_wsp(WspID1);
     println("hello again in ",WspID1);
     println("quit..");
     quitWorkspace;
     ?current_wsp(_,Name3,_);
     println("back in ",Name3);
     quitWorkspace;
     cartago.set_current_wsp(Id0);
```

```
?current_wsp(_,Name4,_);
println("...and finallly in ",Name4," again.").
```

Highlights:

- *Working with mutiple workspaces* – Agents can create, join and work in multiple workspace at a time. However there is always a *current* workspace, to which are routed actions with no artifact id or workspace id specified. Current workspace info are automatically tracked by the `current_wsp(WspId,Name,NodeId` belief.

- *Setting the current workspace* – The `cartago.set_current_wsp(WspID)` internal action makes it possible to set the current workspace, specifying its id.

- *Actions on workspaces* – Actions on workspaces include `createWorkspace` – to create a new workspace in current node, provided by the `NodeArtifact`, `joinWorkspace` – to join a workspace on the node, provided by the `NodeArtifact`, `quitWorkspace` to quit the workspace, provided by the `WorkspaceArtifact`.

2.14 Example 11 - Working in Remote Workspaces

Agents can join workspaces that are hosted on remote nodes, by means of a `joinRemoteWorkspace` action (provided by the `NodeArtifact`. As soon as the join succeed, the interaction within remote workspaces is the same as local workspace.

In the following example, a *Jason* agent joins the default workspace of a CArtAgO node running on localhost. The following Java program installs a CArtAgO node on localhost, to make it reachable (also) by remote agents:

```
package examples;

import cartago.*;
import cartago.util.BasicLogger;

public class Ex00a_HelloRemoteWorld {

  public static void main(String[] args) throws Exception {
    CartagoService.startNode();
    CartagoService.installInfrastructureLayer("default");
    CartagoService.startInfrastructureService("default");
    CartagoService.registerLogger("default",new BasicLogger());
    System.out.println("CArtAgO Node Ready.");
  }
}
```

Highlights:

- *Starting a* CArtAgO *Node* – to execute CArtAgO artifact-based environment, first of all a CArtAgO node must be started. This is done by the `startNode` service of `CartagoService`. The node functions as virtual machine for running workspaces and artifacts, it does not include any infrastructural (network) support.
- *Installing infrastructure layers* – in order to allow agents working in the local node to interact also with remote nodes – or to allow linking with remote artifacts – a proper infrastructural layer must be installed by means of `installInfrastructureLayer`, specifying the protocol(s) to be used (`default` means the default protocol of the platform, which is RMI in the case of Java SE desktop environments).
- *Intalling infrastructure services* – in order to make the node reachable from remote agents, then infrastructure services must be started by means of `startInfrastructureService`, again specifying the protocol.

Then, it follows the *Jason* program which creates a standalone CArtAgO node with a single agent:

```
MAS example11_remote {

  environment:
  c4jason.CartagoEnvironment

  agents:
  voyager agentArchClass c4jason.CAgentArch;

  classpath: "../../../lib/cartago.jar";"../../../lib/c4jason.jar";
}
```

The `voyager` agent boots in the standalone node, then it joins a remote workspace, where he creates and uses an artifact.

```
!test_remote.

+!test_remote
  <- ?current_wsp(Id,_,_);
     +default_wsp(Id);
     println("testing remote..");
     joinRemoteWorkspace("default","localhost",WspID2);
     ?current_wsp(_,WName,_);
     println("hello there ",WName);
     !use_remote;
     quitWorkspace.

+!use_remote
  <- makeArtifact("c0","examples.Counter",[],Id);
     focus(Id);
     inc;
     inc.

+count(V)
  <- ?default_wsp(Id);
     println("count changed: ",V)[wsp_id(Id)].

-!use_remote [makeArtifactFailure("artifact_already_present",_)]
  <- ?default_wsp(WId);
     println("artifact already created ")[wsp_id(WId)];
     lookupArtifact("c0",Id);
     focus(Id);
     inc.
```

Highlights:

- *Infrastructure options* – By default, *Jason* programs using CArtAgO environment create a standalone CArtAgO node, i.e. not accessible through the network To install a CArtAgO node accessible also to remote agents further parameters can be specified to the `c4jason.CartagoEnvironment`, in particular:

  - `c4jason.CartagoEnvironment("infrastructure"{, WspName, protocol(ProtName, Address), ...})`
    installs an infrastructure layer specifying the protocols to support and the local address where to start the service;
  - `c4jason.CartagoEnvironment("remote"{, WspName, protocol(ProtName, Address), ...})`
    does not install any node – agents directly join the specified remote workspace;
  - `c4jason.CartagoEnvironment("local"{,WspName})`
    does not install any node – agents directly join the specified locak workspace.

TODO:

- example on RBAC security model