

JACK™ Intelligent Agents Teams Practicals



Copyright

Copyright © 2003-2008, Agent Oriented Software Pty. Ltd.

All rights reserved.

No part of this document may be reproduced, transferred, sold, or otherwise disposed of, without the written permission of the owner.

US Government Restricted Rights

The JACK™ Modules and relevant Software Material have been developed entirely at private expense and are accordingly provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013 or subparagraph (c)(1) and (2) of the Commercial Computer Software Restricted Rights and 48 CFR 52.2270-19, as applicable.

Trademarks

All the trademarks mentioned in this document are the property of their respective owners.

Publisher Information

Agent Oriented Software Pty. Ltd.
P.O. Box 639,
Carlton South, Victoria, 3053
AUSTRALIA

Phone: +61 3 9349 5055
Fax: +61 3 9349 5088
Web: <http://www.agent-software.com>

If you find any errors in this document or would like to suggest improvements, please let us know.

The JACK™ documentation set includes the following manuals and practicals:

Document	Description
Agent Manual	Describes the JACK programming language and infrastructure. JACK can be used to develop applications involving BDI agents.
Teams Manual	Describes the JACK Teams programming language extensions. JACK Teams can be used to develop applications that involve coordinated activity among teams of agents.
Development Environment Manual	Describes how to use the JACK Development Environment (JDE). The JDE is a graphical development environment that can be used to develop JACK agent and team-based applications.
JACOB Manual	Describes how to use JACOB. JACOB is an object modelling language that can be used for inter-process transport and object initialisation.
WebBot Manual	Describes how to use the JACK WebBot to develop JACK enabled web applications.
Design Tool Manual	Describes how to use the Design Tool to design and build an application within the JACK Development Environment.
Graphical Plan Editor Manual	Describes how to use the Graphical Plan Editor to develop graphical plans within the JACK Development Environment.
JACK Sim Manual	Describes how to use the JACK Sim framework for building and running repeatable agent simulations.
Tracing and Logging Manual	Describes the tracing and logging tools available with JACK.
Agent Practical	A set of practicals designed to introduce the basic concepts involved in JACK programming.
Teams Practical	A set of practicals designed to introduce the basic concepts involved in Teams programming.

Table of Contents

JACK™ Teams Practicals	9
Exercise 1	10
Introduction	10
Instructions	10
Exercise 2	15
Instructions – Part 1	15
Instructions – Part 2	16
Exercise 3	18
Introduction	18
Instructions	18
Exercise 4	21
Introduction	21
Instructions	23
Exercise 5	25
Introduction	25
Instructions	26
Exercise 6	29
Introduction	29
Instructions	30
Teams Practicals Solutions	33
Program Solutions	33
Answers to Questions	33
Exercise 3	33
Exercise 4	33

JACK™ Teams Practicals

This set of exercises provides a basic introduction to JACK™ Teams (Teams). It is assumed that the user is already familiar with JACK™ Intelligent Agents. Teams is an extension to JACK™ Intelligent Agents (JACK) that provides a team-oriented framework. The *Overview* chapter in the *Teams manual* provides the user with an introduction to team-oriented programming and the Teams extensions.

Exercise 1

Build a simple application consisting of a team with several sub-teams.

Introduction

The intent of this exercise is to demonstrate how to build a Teams application where one team requires several sub-teams to perform roles on its behalf. This example will consist of a `Spacecraft` team which will contain 3 `Martian` sub-teams capable of performing the roles of `Pilot`, `SpokesPerson` and `Crew`.

If you have not already read the *Overview* chapter of the *Teams manual*, you should read it before beginning this exercise. Note that the introductory example in the *Overview* is similar, but not identical to the example developed in the following exercise.

Instructions

1. Create the subdirectories `martian` and `spacecraft`. The application will be organised into two packages (`martian` and `spacecraft`). The first package will contain the plans etc. for the `Martian` team and the other will contain the plans etc. for the `Spacecraft` team.
2. Create the `Martian.team` file in the `martian` subdirectory.
 - As the `Martian` team is to be capable of performing the `SpokesPerson`, `Pilot` and `Crew` roles, it must contain `#performs role` declarations for these roles. For example, the declaration for the `performs SpokesPerson` role is:

```
#performs role SpokesPerson;
```
 - The team will have plans to travel, speak a greeting and 'be on watch'. It must therefore contain `#uses plan` declarations for the plans `SpeakGreeting`, `Travel` and `WatchMonitor`.
 - It must also have the usual constructor.
3. Create the `Spacecraft.team` file in the `spacecraft` subdirectory.
 - It must contain `#requires role` declarations for the roles `SpokesPerson`, `Pilot` and `Crew`. The `spacecraft` requires 3 sub-teams capable of filling each of these roles. These declarations are of the form:

```
#requires role RoleType ref(min,max);
```
 - It must contain a `#uses plan Visit` declaration.
 - It must have a `#posts event PerformVisit ref` declaration and a public `void visit(String planet)` method to post itself a `PerformVisit` event. This method should use the `postWhenFormed` method to post the event when the team has finished building its initial role obligation structure.

- It must have a `#handles event PerformVisit` declaration.
- It must also have a constructor.
- As this is the containing team, it will automatically handle a `TeamFormationEvent` at construction time. This will be handled by a default plan which will use an initialisation file to build the role obligation structure. The initialisation file is described later in the practical.
- As the roles are to be defined in the `martian` package, it must also include the following import statements:

```
import martian.CrewContainer;  
import martian.Crew;  
import martian.PilotContainer;  
import martian.Pilot;  
import martian.SpokesPersonContainer;  
import martian.SpokesPerson;
```

4. In the directory above the `martian` and `spacecraft` directories, create the main Java program. This program should construct three `Martian` teams with appropriate names. It must then construct the `Spacecraft` team. The containing team should not be constructed before any of its sub-teams.

A main program called `AlienProgram` is given below:

```
import martian.Martian;  
import spacecraft.Spacecraft;  
  
public class AlienProgram {  
  
    public static void main(String [] args)  
    {  
        new Martian( "Dennis" );  
        new Martian( "Ralph" );  
        new Martian( "Jacquie" );  
        Spacecraft spacecraft = new Spacecraft( "Enterprise" );  
        spacecraft.visit("Earth");  
    }  
}
```

JACK™ Teams Practicals

Exercise 1

5. In the same directory as the main program create the initialisation file to build the role obligation structure. The name `scenario.def` is often used for this file. In this example, all 3 Martians should be capable of performing all 3 roles in the role obligation structure. The start of the initialisation file is given below:

```
<Team :name "Enterprise"
  :roles (
    <Role :type "martian.SpokesPerson" :name "sp"
      :fillers (
        <Team :name "Dennis@%portal" >
          :
          :
        )
      >
    :
    :
    etc.
  )
>
```

Note:

- the name `sp` must correspond to the reference for the `SpokesPerson` in the `#requires` declaration in the `Spacecraft` team.
 - you must take care to include the package when specifying the role types (e.g. `martian.SpokesPerson` shown in the example above).
-

6. Create the 3 role definition files (`Crew.role`, `Pilot.role` and `SpokesPerson.role`) in the `martian` package.

- The `Crew.role` must be able to handle a `DoWatch` event.
- The `Pilot.role` must be able to handle a `PilotCraft` event.
- The `SpokesPerson.role` must be able to handle a `DoGreeting` event.

In all three cases, the roles indicate the downward interface between a team that can perform that role and a team that requires a sub-team to perform the role. This indicates the events that will be posted from the containing `Spacecraft` team to the `Martian` sub-team capable of performing the role. This means that the `Martian` sub-team must have at least one plan capable of handling the specified event.

7. Create the `MessageEvents` required for the application in the `martian` package:

- `DoGreeting.event` with a `String` member for the planet and a posting method `speakGreeting(String p)` where `p` is the name of the planet.
- `DoWatch.event` with a posting method `watch`.
- `PilotCraft.event` with a `String` member for the planet and a posting method `startTrip(String p)` where `p` is the name of the planet.

8. Create the `MessageEvents` required for the application in the `spacecraft` package:
 - `PerformVisit.event` with a `String` member for the planet and a posting method `visitPlanet(String p)` where `p` is the name of the planet.
9. Create the plans used by the `Martian` sub-teams in the `martian` package:
 - `WatchMonitor.plan` handles the `DoWatch` event. Write the body of the plan. It should print out a message that the team is on watch. The team name can be obtained by including a `#uses interface Martian self` declaration at the beginning of the plan. The name can then be obtained with `self.name()`.
 - `SpeakGreeting.plan` handles the `DoGreeting` event. The body of this plan should print a suitable greeting which contains the name of the team and the name of the planet.
 - `Travel.plan` handles the `PilotCraft` event. The body of this plan should print a suitable message to indicate that the pilot sub-team has commenced the journey to the given planet. It should then contain a short delay which can be achieved using `@waitFor(elapsed(10.0))`. This should be followed by a message to indicate that the team has arrived at the planet.
10. In the `spacecraft` package, create the `Visit` plan to be used by the `Spacecraft` team.

This plan handles the `PerformVisit` event. It also coordinates the activities among the sub-teams to allow the team to travel to the planet and speak a greeting. It will therefore require the following declarations to indicate that it requires sub-teams to perform the following roles to carry out this task:

```
#requires role SpokesPerson sp as speaker;  
#required role Pilot pi as pilot;  
#requires role Crew cr as crew;
```

Note that the `sp`, `pi` and `cr` references must correspond to the references in the `Spacecraft` team definition. By using the `#requires` declaration in the plan, we allow the plan to use the default `establish` method to select sub-teams to perform the roles within the plan.

The default `establish` method will assign a `Role` instance to `speaker` from the list of `Role` instances in the `SpokesPersonContainer`. Similarly, it will assign `Role` instances to `pilot` and `crew`.

The `Role` type has a `String` member `actor` which can be used to obtain the name of the sub-team associated with the role. In this exercise, the body of the `visit` plan should print an appropriate message to indicate the `task team` that has been established for this plan. By including a `#uses interface Team team` declaration at the beginning of the plan, the team name can be obtained with `team.name()`. For example:

```
System.out.println("Team established for craft "+
                   team.name());
System.out.println("crew member  = "+crew.actor);
System.out.println("pilot       = "+pilot.actor);
System.out.println("spokesperson = "+speaker.actor);
```

This plan must also contain the following import statements:

```
import martian.CrewContainer;
import martian.Crew;
import martian.PilotContainer;
import martian.Pilot;
import martian.SpokesPersonContainer;
import martian.SpokesPerson;
```

11. Compile the program with the following command:

```
java aos.main.JackBuild -r -map=team
```

Create a `mkit` script which contains this command.

12. Assuming your program is called `AlienProgram` and that the initialisation file is called `scenario.def`, run the program with the following command:

```
java -DTeam.Structure=scenario.def AlienProgram
```

The output will look like:

```
Team established for craft: Enterprise@%portal
crew          = Ralph@%portal
pilot        = Ralph@%portal
spokesperson = Ralph@%portal
```

You will notice that it is possible for the same sub-team to be assigned to more than one role within the task team. In the next exercise an `establish` method will be developed which restricts each martian sub-team to performing only one role within the `visit` plan's task team.

13. Create a `runit` script to run your program.

14. It is also useful to have a `cleanit` script which contains the following command:

```
java aos.main.JackBuild -r -c -map=team
```

Exercise 2

Complete the body of the `visit` plan and write an `establish` method to ensure that each martian sub-team is only responsible for performing one role within the `visit` plan.

Instructions – Part 1

1. Initially we will assume that it is possible for one sub-team to be responsible for all three roles and complete the body of the plan. The first step is to have the sub-team tasked to perform the `Pilot` role and fly the craft to the planet. This is achieved by sending a `PilotCraft` event to the sub-team responsible for the `Pilot` role by using a `@teamAchieve` statement as follows:

```
@teamAchieve(pilot, pilot.st.startTrip(eventref.planet));  
// where eventref is the event being handled by the plan
```

Note that the reference to the event factory is through the role (i.e. `st` is the reference used in the declaration of the `PilotCraft` event in the `Pilot` role definition).

2. At the same time, the sub-team responsible for performing the `Crew` role must maintain a watch to ensure that no problems arise. This is also achieved by using the `@teamAchieve` statement. This time it is used to send a `DoWatch` event to the sub-team responsible for the `Crew` role.

As the two activities are to be carried out in parallel, the two `@teamAchieve` statements should be inside an `@parallel` statement. In this example the arguments used in the `@parallel` statement are as follows:

```
@parallel(ParallelFSM.ALL, false, null)  
{  
    // the branch statements  
};
```

- The first argument is the mode. `ParallelFSM.ALL` is used. This means the `@parallel` statement will succeed after all the branches have succeeded, but fail immediately if any branch fails. All ongoing sub-statements will be notified on failure.
- The second argument is the termination condition. In this example, there is no termination condition, so this is `false`.
- The third argument is used for a user-defined Java exception object. If it is not null, the exception is thrown to active branches that are executing in parallel if they are required to terminate. Using `null` (as in this instance) means that the sub-statements will be completed without any notification.

3. Compile and run the program.

4. When the `@parallel` statement has completed it should mean that the craft has arrived at the planet. Make the following additions to land the craft:

- Create a `Land` event in the `martian` package. This is to be sent to the pilot when it is time to land the craft on the planet. It is to have a `String` member for the planet and a posting method `landcraft(String p)` where `p` is the name of the planet.
- Create a `LandCraft` plan to handle the `Land` event. This should print a message to state that the pilot is in the process of landing, wait for a short period of time and then print a message to indicate that the craft has landed on the planet.
- Add a `#handles event Land la` declaration to the `Pilot` role.
- Add a `#uses plan LandCraft` declaration to the `Martian` team definition.
- Add another `@teamAchieve` statement after the `@parallel` statement in the `Visit` plan to send a `Land` event to the pilot.

5. This should be followed by another `@teamAchieve` statement to get the speaker to speak the greeting. Remember that the `@teamAchieve` is synchronous, so this will not be executed until the craft has landed.

6. Compile and run this version of the program.

Instructions – Part 2

7. Create an `establish` method which restricts each sub-team to performing only one role within the `task` team. This can be achieved by iterating through the role instances inside each role container and selecting one that is associated with a sub-team that is not already being used for a role. When a role instance is selected the team name can be stored in a 'busy' vector, so that this team does not get selected to perform another role.

The `RoleContainer` base class has a method `tags()` which returns its current role object tags as a `java.util.Enumeration`. These role object tags relate to the role instances and can be used as the argument to the role containers `find` method to obtain the corresponding role instance. In this way we can iterate through the role instances in a role container. The `establish` method described (and an associated helper method) are given below:

```
#reasoning method
establish()
{
    Vector busy = new Vector();
    crew = (Crew) pickRole( busy, cr );
    crew != null;
    pilot = (Pilot) pickRole( busy, pi );
    pilot != null;
    speaker = (SpokesPerson) pickRole( busy, sp );
    speaker != null;
}

Role pickRole(Vector busy, RoleContainer rc)
{
    for (Enumeration e = rc.tags(); e.hasMoreElements(); ) {
        Role r = rc.find( (String) e.nextElement() );
        if ( !busy.contains( r.actor ) ) {
            busy.add( r.actor );
            return r;
        }
    }
    return null;
}
```

8. The `#requires` declarations in the `visit` plan should now be changed to `#uses` declarations as the default `establish` method is no longer being used to establish the task team.

9. Add the following `import` statements to `visit.plan`:

```
import java.util.Enumeration;
import java.util.Vector;
```

10. Compile and run the new version of the program.

Exercise 3

Introduction

In the current version of the program, the sub-team performing the watch only does this for a short period of time. It should actually continue this task until notified to stop (or at least until the craft has arrived). In this exercise, the `WatchMonitor` plan will continue until it is notified that it is no longer required to perform the watch.

Instructions

1. Modify the body of the `WatchMonitor` plan so that after the print statement it enters a forever loop which contains a `@waitFor(elapsed(10.0))` statement.
2. Add `pass` and `fail` reasoning methods with appropriate print statements. These are used for tracing purposes in this exercise. They will enable us to tell whether or not the plan has succeeded or failed.
3. Compile and run this version of the program. What happens? Why?
4. Modify the mode in the `@parallel` statement in the `Visit` plan to `ParallelFSM.FIRST`.
5. Compile and run this version of the program. Notice that although the `@parallel` statement ends now, there is no message to indicate that the `WatchMonitor` plan has terminated.
6. In this version of the example, we send an event to the crew member to start the watch and then send an event to stop the watch at the appropriate time. In exercise 4 we will explore an alternative mechanism for interrupting the `WatchMonitor` plan. 'Clean' the application (i.e. remove class files etc.) using the following command (or your `cleanit` script):

```
java aos.main.JackBuild -r -c -map=team
```

and copy the current version of the program into another directory to be used as the starting point for exercise 4.

7. Modify the `DoWatch` event so that it contains a boolean member `todo` to indicate whether the watch command is being started or stopped. Introduce two new posting methods `startWatch()` and `stopWatch()` which set `todo` to `true` and `false` respectively.
8. Introduce a new beliefset to the `martian` package called `CommandsStatus`. It contains one key field of type `String` and is used as a store of all the active commands.
9. Add a `#private data CommandsStatus commands()` declaration to the `Martian` team definition.

10. Modify the `WatchMonitor` plan as follows:

- Add a `#uses data CommandsStatus commands` declaration.
- Add a reasoning method `performWatch` which prints out a message to indicate that the team is on watch and then performs the watch as follows:

```
while(commands.get("PerformWatch"))
{
    // An actual task could be wrapped in @maintain.
    // Here we use @waitFor (with a sentinel) to
    // represent doing the task
    @waitFor(elapsed(10.0), !commands.get("PerformWatch"));
}
```

- Add a `relevant` method to the plan. This plan is to be relevant if the `todo` member of the `DoWatch` event is `true`.
- Modify the body of the `WatchMonitor` plan. It tests whether or not the `commands` beliefset already contains a `PerformWatch` command. If it does, we assume the crew member is already actively performing the watch. If it does not, add the `PerformWatch` command to the `commands` beliefset and begin performing the watch.

11. Write a new plan called `StopWatch`. This plan is to be relevant if the `todo` member of the `DoWatch` event is `false`. The body of this plan is to remove the `PerformWatch` command from the `commands` beliefset. Make sure you declare that the `Martian` team uses the new `StopWatch` plan.

12. Modify the `visit` plan so that the `@parallel` statement contains the following two branches:

- a branch that uses `@teamAchieve` to send an event to the crew member to start the watch
- a branch that contains two statements. The two statements must be enclosed in `{ and }` brackets. The first statement is an `@teamAchieve` to send an event to the pilot to start travelling to the planet. When this is finished, the second statement uses `@teamAchieve` to send an event to the crew member to stop the watch. Note that the `WatchMonitor` plan will be considered to have failed if it is terminated by the sentinel in the `@waitFor()`. This means the mode used must be `ParallelFSM.ANY` and not `ParallelFSM.ALL` or the `@parallel` statement will fail and the plan will fail.

13. Compile and run the program. The output should look similar to the following:

```
Team established for craft: Enterprise@%portal
  crew          = Ralph@%portal
  pilot         = Dennis@%portal
  spokesperson = Jacquie@%portal
Ralph@%portal on watch
Dennis@%portal flying craft to Earth
Dennis@%portal arriving at Earth
WatchMonitor plan terminating (fail)
Dennis@%portal landing craft at Earth
Dennis@%portal has landed craft at Earth
Hello Earth. I am Jacquie@%portal.
```

Exercise 4

Use the `aos.extension.parallel.ParallelMonitor` to throw an exception to interrupt the `WatchMonitor` plan.

Introduction

The arguments to the `@parallel` statement specify the success condition, termination condition and how termination is notified. In addition, an optional fourth argument is allowed, which is then an object through which the execution of the parallel statement can be monitored.

The optional monitor attribute must, if given, be an instance of the class `ParallelMonitor`. The `ParallelMonitor` class implements the following interface:

```
public int addTask(FSM)
//
// A method that can be used to add branches to an @parallel
// statement dynamically. The FSM argument is an event or
// reasoning method in the plan. Branches in a @parallel statement
// can be referred to by index, where 0 is the first branch.
// Dynamically added branches are numbered contiguously after the
// definite branches. The addTask method returns the index of the
// branch added.
//

public Cursor finished()
//
// A triggered Cursor for checking that the
// @parallel statement has finished.
//

public Cursor changed()
//
// A triggered Cursor for reacting to state changes in the
// execution of the @parallel statement, i.e. when branches finish.
//

public boolean hasFinished()
//
// Tests whether the @parallel statement has finished or not.
//

public int getStatus()
//
// Returns the current execution status of the @parallel statement.
//

public int nTasks()
//
// Returns the number of parallel branches.
//

public int getStatus(String n)
//
// Returns the execution status of a labelled branch.
```

JACK™ Teams Practicals

Exercise 4

```
// return values can be:
// -1 (active)
// 1 (finished successfully)
// 2 (failed)
// 7 (terminated with an exception)
//

public int getStatus(int n)
//
// Returns the execution status of a branch by index.
// Values can be:
// -1 (active),
// 1 (finished successfully)
// 2 (failed)
// 7 (terminated with an exception)
//

public Throwable getException(String n)
//
// Returns the exception, if any, thrown to a labelled branch.
//

public Throwable getException(int n)
//
// Returns the exception, if any, thrown to a branch by index.
//

public int findTaskIndex(String name)
//
// Returns the index for a labelled branch.
//

public void throwTo(String name, Throwable t)
//
// Throws an exception to a labelled branch.
//

public void throwTo(int n, Throwable t)
//
// Throws an exception to a branch by index.
//
```

The `ParallelMonitor` object allows the team plan to inspect the processing of parallel branches and (as in the code segment below) throw exceptions to branches selected by label or index. If the branch contains an `@teamAchieve`, the plan activated by the `@teamAchieve` will receive a `TeamAbort` exception. The plan can catch this and take appropriate action. Note that if an exception is thrown to a branch, the branch is terminated and the branch is considered to have failed. If the plan that was activated by the `@teamAchieve` does not catch the `TeamAbort` exception, you will not get any indication that the plan has been terminated. Neither the `pass` nor the `fail` reasoning method will be executed in this situation.

```
ParallelMonitor p = new ParallelMonitor();
@parallel(....., p) {
    ....;
    label: .....;
    {
        @waitFor(elapsed(100));
        p.throwTo("label", new Exception("CheckPoint"));
    };
};
```

Instructions

1. Change directory to the version saved before the previous exercise (i.e. before the version to interrupt the `WatchMonitor` plan using a `beliefset`.
2. In this version, we will interrupt the `WatchMonitor` plan by using a `ParallelMonitor` object to throw an exception.
3. In `Visit.plan`, import `aos.extension.parallel.ParallelMonitor`.
4. In `Visit.plan`, create a `ParallelMonitor` object `p` before the `@parallel` statement. Modify the `@parallel` statement to be:

```
@parallel(ParallelFSM.ANY, false, null, p)
{
    watch: @teamAchieve(crew, crew.wm.watch()); // wm is
                                                // the reference
                                                // in Crew.role
    flying: fly(p);
};
```

Note that `ParallelFSM.ANY` is used. Why?

5. `fly` is a reasoning method in which `@teamAchieve` is used to send an event to the pilot to start travelling to the planet. When this is complete, the `fly` reasoning method must use the `ParallelMonitor` object to test the status of the `watch` branch (using `getStatus("watch")`). If it is still active (which it should be in this example), the `ParallelMonitor` object must throw an exception to the `watch` branch (using the `throwTo(...)` method). Create this `fly` reasoning method in the `visit` plan.

JACK™ Teams Practicals

Exercise 4

6. Compile and run the program. Note that although the `pass` and `fail` reasoning methods are not executed, the `WatchMonitor` plan has been terminated by a `TeamAbort` exception.

7. In the `WatchMonitor` plan, wrap the `performWatch` activity in a `try/catch` statement that catches a `TeamAbort` exception. Print a trace statement if you catch a `TeamAbort` in this plan.

8. Add the `import aos.team.TeamAbort` statement to the `WatchMonitor` plan.

9. Compile and run the program. Your output should be similar to the following:

```
Team established for craft: Enterprise@%portal
  crew          = Ralph@%portal
  pilot         = Dennis@%portal
  spokesperson = Jacquie@%portal
Ralph@%portal on watch
Dennis@%portal flying craft to Earth
Dennis@%portal arriving at Earth
throwing exception to watch
crew received a TeamAbort exception to stop watch
WatchMonitor plan terminated normally (pass)
Dennis@%portal landing craft at Earth
Dennis@%portal has landed craft at Earth
Hello Earth. I am Jacquie@%portal
```

10. Try changing the mode in the `Visit` plan `@parallel` statement to `ParallelFSM.ALL`. Compile and run the program. What happens? Why?

11. Change the mode back to `ParallelFSM.ANY`.

Exercise 5

In this exercise we illustrate the upwards propagation of team beliefs.

Introduction

Belief propagation focuses on how the beliefsets of teams and sub-teams may be connected through role relationships. Belief connections are either directed upwards, synthesizing the beliefs of sub-teams, or downwards, allowing sub-teams to inherit beliefs from the team.

In this exercise, the pilot will maintain a flight status beliefset which, for simplicity, will consist of two non key `String` fields:

- `status` (e.g. takeoff, transit, holding, landing, landed)
- `destination` (i.e. the destination planet)

This information is to be propagated up to the spacecraft team. In some applications, data could be propagated from several sub-teams and then merged in some way. For example, sub-teams could propagate their location and the team could combine these locations into a single aggregated location. This is not illustrated in this exercise.

To achieve upward propagation of beliefs, the following components need to be provided:-

1. A data source definition

A generic capability for propagating changes is provided as part of the beliefset infrastructure. For a beliefset to be used as a source for belief propagation, it must include a `#propagates changes` declaration.

Note that the `#propagates changes` declaration may include an optional `EventType`. The use of this second form of the `#propagates changes` declaration is described in the Teams manual. It is not illustrated in this practical.

2. Source team declarations

A sub-team becomes a source in a synthesizing belief connection by filling a role that contains a `#synthesizes teamdata` declaration. Thus the sub-team must include an appropriate `#performs role` declaration and fill the role in the containing team's role obligation structure. Also, a data item with the type and the reference specified within the role must be defined within the sub-team definition, or indirectly through the sub-team's capability structure.

3. Role declarations

To associate a synthesizing belief connection with a role, the following statement form is used:

```
#synthesizes teamdata stype sref;
```

where `stype` and `sref` identify a **source** beliefset that will be involved in a synthesizing belief connection – the target for the connection is not specified.

Recall that a role defines a team/sub-team interface and as such this `synthesizes teamdata` declaration does not generate any code. Rather, it declares that any sub-team that performs this role must provide a data item named `sref` of type `stype`. Likewise any team that requires this role must have a target data declaration that involves this particular item.

4. A target data definition

The `teamdata` construct is provided to encapsulate the behaviour and data associated with the target end of a team belief connection. A template for a `teamdata` type definition is given in the instructions for this exercise.

5. Target team declarations

A team becomes a target in a synthesizing belief connection by requiring a role that contains a `#synthesizes teamdata` declaration. Thus the team must include an appropriate `#requires` role declaration and a `#synthesizes teamdata` declaration that binds the data item specified in the role with the role container that contains the sub-teams that fill the role. The `synthesizes teamdata` declaration within a team results in the creation of a `teamdata` instance that is private to that team.

Instructions

1. Create the data source type definition.

In the `martian` directory, create the `PilotFlightStatus` beliefset in the normal way. The beliefset is to have two non key `String` fields, `status` and `destination` as described in the introduction to this exercise. In addition to the normal declarations for the value fields and the queries, add the following declaration for belief propagation:

```
#propagates changes;
```

2. Add the data source to the sub-team type definition.

Add a `#private data PilotFlightStatus flightStatus();` declaration to the `Martian` team definition.

3. Add the declaration to associate a synthesizing belief with the `Pilot` role.

Add a `#synthesizes teamdata PilotFlightStatus flightStatus;` declaration to the `Pilot` role definition.

4. Create the target data definition (synthesized teamdata) for the spacecraft.

This is to be called `CraftStatus`. `CraftStatus` should be defined in a file called `CraftStatus.td`. A template for a synthesized teamdata is shown below (`DataType__Tuple` is a placeholder for the type of the incoming tuple):

```
teamdata TeamDataType extends DataType {
    #connection method(boolean added, String team)
    {
        // could have code which captures when
        // sub-teams are added to or removed from the synthesis
        // connection
    }

    #synthesis method( String team,
                       boolean asserted,
                       BeliefState tv,
                       DataType__Tuple is,
                       DataType__Tuple was,
                       DataType__Tuple lost)
    {
        // A simple case where new data added is
        // illustrated here.
        // This method could involve more complicated merging
        if(asserted && (is!=null))
            add(.. is.data fields ...);
    }
}
```

In this exercise the synthesis method simply adds a copy of any new status details propagated up from the pilot's beliefset. Note that

- `CraftStatus` is to extend `PilotFlightStatus`. The teamdata is not required to extend the source beliefset. However, it makes sense for it to do so in this example.
- The tuple type in the arguments to the synthesis reasoning method should be `PilotFlightStatus__Tuple`

- You must import

```
martian.PilotFlightStatus
```

and

```
martian.PilotFlightStatus__Tuple
```

into `CraftStatus.td`

- the body of the `connection` method is empty because we are not dynamically modifying the role obligation structure.

5. Add the declaration for the target data to the `Spacecraft` team.

Add a `#synthesizes teamdata CraftStatus status(pi.flightStatus)` declaration to the `Spacecraft` team definition. `pi` must correspond to the reference in the `#requires Pilot` role declaration in the `Spacecraft` team definition. This statement has the effect of creating an instance of the `CraftStatus` teamdata type called `status` that is private to the `Spacecraft` team. In this exercise, `status` receives beliefs propagated from a sub-team performing the pilot role and stores the propagated beliefs in a `PilotFlightStatus` tuple. (`CraftStatus` extends `PilotFlightStatus`.) The storage is performed within the `#synthesis` method of `status`.

The necessary `#requires/#performs` declarations and `scenario.def` file required to build the role obligation structure for the `spacecraft` and `martian` teams have been established in previous exercises.

The components are now in place for belief propagation to take place. The remainder of this exercise uses belief propagation to propagate belief changes from the `from` the sub-team in the pilot role to the `spacecraft`.

6. Add a `#uses data CraftStatus status` declaration to the `Visit` plan.

7. Modify `visit.plan` so that after the `@parallel` statement is executed, it checks that the `status` teamdata contains the tuple `("holding", eventref.planet)` (where `eventref` is the event being handled by the plan). If it does, then print a message and carry out the remaining `@teamAchieve` statements to land the craft and speak the greeting. If it does not, print an error message and make the plan fail.

8. Modify the `martian Travel.plan` and `LandCraft.plan` so that they assert status information in the `PilotFlightStatus` beliefset at each stage of the journey. These plans will require `#uses data PilotFlightStatus flightStatus` declarations.

9. Compile and run the program

Exercise 6

In this exercise we illustrate the downwards propagation of team beliefs.

Introduction

In this exercise we will propagate the craft status information down from the spacecraft to the sub-teams in the `CrewContainer` and `SpokesPersonContainer`.

To achieve downward propagation of beliefs, the following components need to be provided:-

1. A data source definition

A generic capability for propagating changes is provided as part of the beliefset infrastructure. For a beliefset to be used as a source for belief propagation, it must include a `#propagates changes` declaration.

2. Source team declarations

A team becomes a source in an inheriting belief connection by requiring a role that contains a `#inherits teamdata` declaration. Thus the team must include an appropriate `#requires role` declaration. Also, a data item with the type and the reference specified within the role must be defined within the team definition, or indirectly through the team's capability structure.

3. Role declarations

To associate an inheriting belief connection with a role, the following statement form is used:

```
#inherits teamdata stype sref;
```

where `stype` and `sref` identify a **source** beliefset that will be involved in an inheriting belief connection – the target for the connection is not specified.

Recall that a role defines a team/sub-team interface and as such this `inherits teamdata` declaration does not generate any code. Rather, it declares that any team that requires this role must provide a data item named `sref` of type `stype`. Likewise any team that performs this role must have a target data declaration that involves this particular item.

4. A target data definition

The `teamdata` construct is provided to encapsulate the behaviour and data associated with the target end of a team belief connection. A template for a `teamdata` type definition is given in the instructions in the previous exercise.

5. Target team declarations

A sub-team becomes a target in an inheriting belief connection by filling a role that contains a `#inherits teamdata` declaration. Thus the sub-team must include an appropriate `#performs` role declaration and fill the role in the containing team's role obligation structure. The sub-team must also include a `#inherits teamdata` declaration that binds the data item specified in the role with the role type.

Instructions

1. Add a trace statement to the teamdata definition.

In this exercise the same teamdata type is used as both the source and target data type. The `Martian` team will have a private instance of `CraftStatus` teamdata that is to contain beliefs inherited from the spacecraft's `CraftStatus` teamdata. This will enable the `Martian` sub-teams to track the craft status and their current 'position'.

When data is added to the spacecraft's `status` beliefs, the beliefs will be propagated down to the sub-teams involved in the inheritance connection. Note that there is no need to have a `#propagates changes` declaration in the teamdata definition as it extends the `PilotCraftStatus` beliefset which already has a `#propagates changes` declaration.

The `CraftStatus` type was defined in exercise 5 and is in the `spacecraft` package.

To help follow the belief propagation, add a trace statement to the `CraftStatus` synthesis method to print any new data as it is added.

2. Add a `#inherits` declaration to the Role definition.

Add a `#inherits teamdata CraftStatus status;` declaration to the `Crew` role definition.

`CraftStatus` is defined in the `spacecraft` package, so it is necessary to add an import statement for `spacecraft.CraftStatus` to the `Crew` role definition.

3. Add the `#inherits` declaration to the target team.

Add a `#inherits teamdata CraftStatus craftstatus(Crew.status)` declaration to the `Martian` team definition. Each instance of a `Martian` team will now have its own private instance of a `CraftStatus` teamdata to enable it to track the craft status. The information is only propagated to the sub-teams that fill in the `Crew` role. When the propagated beliefs are inherited by a sub-team, they are again dealt with by the `#synthesis` method in the `CraftStatus` teamdata and the data is stored in a `PilotFlightStatus` tuple.

`CraftStatus` is defined in the `spacecraft` package, so it is necessary to add an import statement for `spacecraft.CraftStatus` to the `Martian` team definition.

The necessary `#requires/#performs` declarations and `scenario.def` file required to build the role obligation structure have been established in previous exercises.

The components are now in place for downward propagation to take place.

4. Compile and run the program.

5. You should have noticed that **all** sub-teams in the `Crew` role container receive the new data from the `Spacecraft` team. If there were any martians in the `SpokesPerson` role container that were not in the `Crew` role container, they would be unaware of the craft status. To overcome this deficiency:

- add the appropriate declaration and import statement to the `SpokesPerson` role definition
- modify the `#inherits` in the `Martian` team as follows:

```
#inherits teamdata CraftStatus craftstatus(Crew.status,  
                                              SpokesPerson.status);
```

6. Compile and run the program. Note that the information should only have been propagated to each sub-team once, not once/role that the sub-team is involved in.

Teams Practicals Solutions

Program Solutions

The solutions to the programming exercises can be found in the `practicals/teams/solutions` subdirectory. There is a separate directory for each of the exercises.

Answers to Questions

Exercise 3

Instruction 3: The spacecraft does not arrive. This is because the `@parallel` statement does not terminate now that the `watch` branch does not terminate.

Exercise 4

Instruction 4: `ParallelFSM.ANY` is used so that the `@parallel` statement succeeds if the `flying` branch terminates successfully.

Instruction 10: The statements after the `@parallel` statement are not executed. The `watch` branch is considered to fail – it is terminated when the `fly` reasoning method throws it an exception. When the `@parallel` statement has a mode of `ParallelFSM.ALL` and one of the branches fails the `@parallel` statement fails.

Note that the actual `WatchMonitor` plan is interrupted in this exercise. The plan passes because the `TeamAbort` exception is caught in the `try/catch` statement. Without the `try/catch` statement, the plan would be interrupted and terminate immediately – it would not even execute the `fail` reasoning method to indicate that it had terminated.