

JACK® Intelligent Agents Teams Manual



Copyright

Copyright © 2002-2009, 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

1	Overview	9
1.1	Background	9
1.2	Team-Oriented Programming	9
1.3	Team-Oriented Concepts	10
1.3.1	Team	10
1.3.2	Role	10
1.3.3	Teamdata	10
1.3.4	Teamplan	11
1.4	The Team Framework	11
1.4.1	Structure	11
	Teams and Roles	11
	Role Definition	12
	Team Formation	13
	Task Teams	14
1.4.2	Behaviour	14
	@teamAchieve	15
1.4.3	Belief Propagation	15
1.5	Example: Martian Visitors	15
2	Teams	27
2.1	Team Definition	27
2.2	Team Declarations	28
	#requires role RoleType reference(min,max)	28
	#performs role RoleType	28
	#synthesizes teamdata DataType ref (r1.s1, r2.s2, ...)	29
	#inherits teamdata DataType ref (r1.s1, r2.s2, ...)	29
2.3	Team Management	30
2.3.1	Team Construction	30
2.3.2	The Team Manager	30
2.3.3	Initial Team Formation	31
2.3.4	Dynamic Team Formation	32
2.3.5	Initialising Teams	33
2.4	The Team Base Class	34
3	Roles	37
3.1	Role Definition	37
3.2	Role Declarations	38
	#handles event EventType reference	39
	#posts event EventType reference	39
	#synthesizes teamdata DataType reference	39

	#inherits teamdata DataType reference	40
	#container method	40
	#container member	40
3.3	The Role Base Class	40
3.4	The RoleContainer Base Class	42
3.5	The Generated RoleType Class	44
3.6	The Generated RoleTypeContainer Class	44
4	TeamPlans	45
4.1	TeamPlan Definition	45
4.2	TeamPlan Declarations	47
	#requires role RoleType rolecontainer_ref as role_ref	47
	#requires role RoleType rolecontainer_ref as role_ref (size)	47
	#uses role RoleType rolecontainer_ref as role_ref	48
	#uses role RoleType rolecontainer_ref as role_ref (size)	48
	#uses role RoleType rolecontainer_ref	48
	#applicable_for role RoleType roleinstance_ref	49
	#applicable_from role RoleType roleinstance_ref	49
4.3	Task Team Establishment	49
4.4	TeamPlan Members and Methods	50
4.5	Team Goal Handling	50
4.6	TeamPlan @-statements	51
4.6.1	The @teamAchieve Statement	51
	Getting Return Values Through @teamAchieve	53
	Exception Propagation for @teamAchieve	53
5	Team Belief Connections	55
5.1	Source Data Definition	55
5.2	Target Data Definition	57
5.3	Belief Connection Dynamics	59
5.4	Synthesizing Belief Connection Definition	60
5.4.1	Role Declarations	61
5.4.2	Source Declarations	61
5.4.3	Target Declarations	61
5.4.4	An Example	62
5.5	Inheriting Belief Connection Definition	66
5.5.1	Role Declarations	67
5.5.2	Source Declarations	67
5.5.3	Target Declarations	67
5.5.4	An Example	68
6	Team Formation	71
6.1	RoleType Instance State Management	71

6.2	Role Handling Events and Messages	73
	Index	77

1 Overview

1.1 Background

JACK Teams™ (Teams) is an extension to JACK Intelligent Agents that provides a team-oriented modelling framework. As Teams builds upon the concepts of JACK, this document assumes that the reader is familiar with JACK Intelligent Agents. It also assumes that the user is familiar with the use of JACOB for initialising data.

The most immediate difference between Teams and JACK is the introduction of the `team` reasoning entity. This entity encapsulates `team` behaviour in a Teams application in the same way that the `Agent` entity encapsulates `agent` behaviour in a JACK application.

Like a JACK agent, a team is also an individual reasoning entity with its own *beliefs*, *desires* and *intentions* (BDI). It includes declarations regarding which roles the team itself may perform for other teams and which roles it offers to other sub-teams to fill. In addition to the normal knowledge-building and practical reasoning in JACK agents, team reasoning is also concerned with the coordination of sub-teams.

As with JACK, behaviour is specified in the form of plans. Teams introduces the `teamplan` construct for the specification of team-oriented behaviour. Because Teams is an extension of JACK, all the core functionality of JACK is available within a team. In particular, a team can use JACK plans as well as `teamplans`.

1.2 Team-Oriented Programming

The Teams extension provides a *team-oriented* modelling framework. Team-oriented programming is an intuitive paradigm for engineering group action in multi-agent systems. Team-oriented programming is conceptually powerful, as it allows the software engineer to specify:

- What a team is capable of doing;
- Which components are needed to form a particular type of team;
- Whether a team is willing to take on a particular role within another team;
- Coordinated behaviour among the team members; and
- Team knowledge.

In short, the concept of team-oriented programming serves to encapsulate coordination activity. It extends the agent concept by associating tasks with roles. However, the flexibility of multi-agent systems is retained. Although team members act in coordination by being given goals according to the specification, they are individually responsible for determining how to satisfy those goals.

A team's structure can contain teams in any combination and in any number. The hierarchy is not restricted to a two-level design or in fact to a hierarchy. Layers of teams can be encapsulated within other levels, and the structure can be added to or altered at any time during the process. In other words, teams can be created in many layers, where each layer is encapsulated within the next layer, and so on.

Both conceptually and explicitly in a model, teams entities exist independent of their team members. For instance, teams can reason about how they belong as members in enclosing teams, or about which teams they include as sub-teams. The teams concept encapsulates coordination activity, and extends the agent concept by associating tasks with roles.

1.3 Team-Oriented Concepts

The Teams extension introduces the new concepts of *team*, *role*, *teamdata* and *teamplan*. The Teams Model includes all the programming elements of the JACK BDI Agent Model, but with an extended semantics for some elements. Agents compiled under the JACK BDI Agent model and teams compiled under the Teams Model can communicate as peers. However, problems may arise if agents compiled under the JACK BDI Agent Model are used as elements of a team.

1.3.1 Team

In Teams, a `team` is a distinct reasoning entity which is characterised by the roles it performs and/or the roles it requires others to perform. The formation of a given team is achieved by attaching sub-teams capable of performing the roles required by the team. Note that a sub-team may be attached to more than one role in a containing team and as a sub-team in many teams. As the sub-teams of the given team may require roles to be performed on their behalf, a multi-level hierarchy (or perhaps a more complex structure) may result.

The team is automatically provided with objects to hold the actual role/sub-team selections. These objects are known as *role containers*.

1.3.2 Role

A `role` in Teams is a distinct entity which contains a description of the facilities that the participants in a team/sub-team relationship must provide. A role defines a relationship between teams and sub-teams. The role relationship is expressed in terms of the goal and belief exchanges implied by the relationship.

1.3.3 Teamdata

`Teamdata` is an addition to the JACK data model concept for change propagation declarations. This allows propagation of beliefs from team to sub-team and vice versa. A `teamdata` element defines how a propagated belief is accepted by the receiving team, and incorporated into its belief structures.

1.3.4 Teamplan

A `teamplan` specifies how a task is achieved in terms of one or more roles. It typically contains steps determining which of the sub-teams nominated to perform the roles will in fact perform each role (a process known as *task team formation*). It also dictates the steps directing each sub-team to achieve specific goals.

Teams provides additional constructs to support both activities (the `establish` reasoning method and the `@teamAchieve` statement). The JACK `@parallel` statement supports non-sequential coordination of sub-team behaviour. As team behaviour embodied in a `teamplan` is specified in terms of roles, it is decoupled from the actual sub-team behaviour. Thus team behaviour can be specified and understood independently of sub-team behaviour.

1.4 The Team Framework

1.4.1 Structure

1.4.1.1 Teams and Roles

A structural relationship between teams is catered for via the *role* concept. A role defines the means of interacting between a containing team (a *role tenderer*) and a contained team (a *role performer* or *role filler*). The role defines which goals the role tenderer may request the role performer to achieve, and it also defines the counter-goals that the role performer may require from the role tenderer.

The team-role structure is defined by statements specifying which roles a team can perform, and which roles must be performed by sub-teams. These declarations are specified in the team's type definitions, where the containing team requires certain roles to be filled, and the contained team must be able to perform certain roles.

A team can perform roles for a containing team and can also contain sub-teams which perform roles on its behalf. The sub-teams can in turn contain sub-teams which can perform roles on their behalf etc.

The following code segments illustrate how these team and role definitions may look.

```
team Company extends Team {
    #performs role CompanyRole;
    // minimum of 3 PlatoonRole fillers required. No upper limit
    #requires role PlatoonRole platoons(3,0);
    // exactly 1 Commander role filler required
    #requires role CommandRole command(1,1);
    // 0 or more ScoutRole fillers required
    #requires role ScoutRole scout(0,0);
    :
    :
}
```

```
role PlatoonRole extends Role {
    #handles event Movement m;
    #posts event Withdraw w;
    :
    :
}
```

In the above example, role definitions for `CompanyRole`, `CommandRole` and `ScoutRole` would also be required.

The team-role declarations determine which team-team structures can be built at run time.

1.4.1.2 Role Definition

The role definition does not contain implementation – only a description of the facilities that the two participants in the role relationship must provide. A role definition has two parts: first, a downwards interface that declares the events an entity must handle to take on the role, and second, an upwards interface that declares the events the team entity requiring the role needs to handle.

A role definition, such as the `PlatoonRole` definition shown above, results in two Java classes being generated by the compiler. One is named by the given `RoleType` type. The second generated class is a specialised 'container' for instances of a `RoleType` called `RoleTypeContainer`. The latter is referred to as a *role container*, as its purpose is to contain `RoleType` objects. In the case of the `PlatoonRole` definition shown above, the compiler would automatically generate the two Java classes `PlatoonRole` and `PlatoonRoleContainer`.

When the declaration is made that a team requires a given role, the result is a role-defined container to be filled by sub-teams. The `#requires role RoleType reference(min,max)` statement adds a field to the team class of name `reference` and type `RoleTypeContainer`. The `#requires role` declaration allows the specification of bounds for the container, which results in team formation constraints.

The arguments `min` and `max` in the `#requires` declaration specify the lower and upper bounds for the number of performers in order for the team to be considered formed. A zero upper bound dictates an unlimited upper bound. Note that these bounds are not enforced by the infrastructure in order to allow dynamic attachment/detachment of sub-teams. In practice, a role container can contain an unspecified number of role objects.

In the team definition illustrated above, the declarations state that a `Company` team requires three sub-teams able to perform the `PlatoonRole` role, another sub-team able to perform the `CommandRole` role, and one or more sub-teams able to perform the `ScoutRole` role.

Furthermore, the `Company` team is declared to be a performer of the `CompanyRole` role, which would be a role required by some other team type.

It should be noted that the declarations above define how an actual team structure may look, but they do not identify the actual team instances, or what the team types are in the actual team structure.

1.4.1.3 Team Formation

The overall lifetime of a team has two phases. The first phase is for setting up an initial *role obligation structure*. The second phase constitutes the actual operation of the team.

At run time, teams undergo a team formation phase intended to identify the particular sub-team instances that take on roles in a team. This first phase is initiated via a `TeamFormationEvent` that is posted by the kernel when each team is constructed. By default, the `TeamFormationEvent` is handled by a plan that identifies the role fillers according to an initialisation file in JACOB format. The following is an example of an initialisation file:

```
<Team :name "company 1"
  :roles (
    <Role :name "hq" :type "Command"
      :fillers (
        <Team :name "cmdgrp 8">
      )
    >
    <Role :name "unit" :type "Subordinate"
      :fillers (
        <Team :name "platoon 1"
          :roles (
            <Role :name "hq" :type "Command"
              :fillers (
                <Team :name "cmdgrp 23">
              )
            >
            <Role :name "unit" :type "Subordinate"
              :fillers (
                <Team :name "section 1">
                <Team :name "section 2">
                <Team :name "section 3">
              )
            >
          )
        >
      )
    >
  )
  >
  ...
  )
>
```

The Teams framework is flexible at this point, but it includes the notion of a fully formed team as a team for which all necessary role performers have been identified.

The framework will allow a team instance to complete its team formation phase without necessarily satisfying all the role filling constraints. However, the team will only be considered formed when its role containment constraints are all filled. This is a state that a program may query.

At this stage the initial role obligation structure has been constructed. It is possible to dynamically modify this structure during program execution. This is discussed in the chapter on *Team Formation*.

1.4.1.4 Task Teams

Task teams are dynamically formed sub-groups within a team, created to perform a team task. When chosen to handle an event, the initial step of a teamplan is to establish the task team, by selecting which role performers to use from within the team for the various roles needed within the task/plan.

Task teams are not defined separately, but are contained within the teamplans defining the team tasks. A teamplan uses `#requires` and/or `#uses` declarations to declare the roles needed for the task team. The teamplan may also include an `establish()` reasoning method that defines how the task team is to be established for the task. This is illustrated in the code segment below:

```
teamplan CompanyFormationMove extends TeamPlan {
    #requires role PlatoonRole platoons as left;
    #requires role PlatoonRole platoons as right;
    #requires role PlatoonRole platoons as depth;
    #requires role CommandRole command as hq;

    #reasoning method
    establish()
    {
        // code to establish the task team for the task
    }

    body()
    {
        // body of the plan to perform the task
    }
}
```

The `establish` step of a teamplan is a proper plan step, and may involve any amount of reasoning by the team entity, as well as negotiations with the candidate sub-teams. The outcome is either a complete assignment of sub-teams to the roles required by the teamplan, or a plan failure allowing the team to choose an alternative plan for handling the same event. If there is a `fail()` reasoning method associated with the plan, it does not get executed if the `establish()` method fails.

There is a default `establish()` method which fills the required roles uniquely at random, if possible. However, the default `establish` method only assigns the `#requires` roles and not the `#uses` roles.

1.4.2 Behaviour

The concepts of teams requiring roles and teams performing roles provide a framework where group behaviours and individual behaviours can be clearly separated. Group behaviour is

specified in terms of the roles that are required to achieve the desired behaviour. This behaviour is specified independently of the actual teams performing the roles. However, the team has access to its sub-teams through the role container, so it is able to perform reasoning based on the actual team membership when necessary.

The team is a separate entity and has its own teamplans for the specification of team behaviour. Within these teamplans, the `@teamAchieve` statement can be used to help coordinate the behaviour of the sub-teams.

@teamAchieve

The `@teamAchieve` statement is used to activate a sub-team by sending an event to the sub-team. The team that sent the `@teamAchieve` then waits until the event has been processed by the sub-team.

In combination with the JACK `@parallel` statement, a wide range of team behaviours can be implemented.

1.4.3 Belief Propagation

In addition to communicating via the normal message/event passing in agent-oriented programming, Teams also provides a capability for the propagation of team beliefs. This propagation can be both from team to sub-team and from sub-team to team. In the latter case, the capability is provided within Teams to combine the propagated sub-team beliefs within the team. The use of Team beliefs in conjunction with the Team coordination statements enables sophisticated team behaviours to be implemented.

1.5 Example: Martian Visitors

This is a simple example to illustrate the basic steps in building a team containing several sub-teams. In this example, a team of Martians are coming to visit Earth.

The team will be contained within a spacecraft which will travel to Earth. Each spacecraft contains at least 3 sub-teams (Martians) capable of performing the role required to pilot the spacecraft. It also contains 3 Martians capable of carrying out the duties performed by a basic crew member, and 3 capable of performing the task of spokesperson when the craft arrives at its destination.

In reality, only one Martian allocated to each of these roles is required when the spacecraft performs the task of visiting Earth. However, 3 Martians are specified per role to ensure that there are backup teams, in case any Martian becomes unavailable.

In this example, the spacecraft contains 3 Martian sub-teams. Each of the Martian sub-teams is capable of performing each of the 3 roles in spacecraft team's role obligation structure. This means that in practice a Martian sub-team could be responsible for more than one role in a task

Overview

team. However, in this example, the establish method ensures that each Martian sub-team is only allocated to one role in the task team.

The steps involved in building this application are as follows:

Step 1: Create the two team types

Spacecraft.team

```
public team Spacecraft extends Team {
    #requires role Spokesperson sp(3,3);
    #requires role Pilot pi(3,3);
    #requires role Crew cr(3,3);

    #uses plan Visit;

    #handles event PerformVisit;

    public Spacecraft(String name)
    {
        super(name);
    }

    #posts event PerformVisit pfv;
    public void visit(String planet)
    {
        postWhenFormed(pfv.visitPlanet(planet));
    }
}
```

Martian.team

```
public team Martian extends Team {
    #performs role Spokesperson;
    #performs role Pilot;
    #performs role Crew;

    #uses plan SpeakGreeting;
    #uses plan Travel;
    #uses plan WatchMonitor;

    public Martian(String name)
    {
        super(name);
    }
}
```

The team definitions are very similar to the definitions for a JACK agent, except that the keyword `team` is used, and the `Team` class is extended. Most of the declarations contained in these team definitions should already be familiar from previous JACK agent programming.

The new declarations illustrated here are `#performs role` and `#requires role`. As previously discussed, this specifies that a spacecraft must contain 3 sub-teams capable of performing the role of `Spokesperson`, 3 sub-teams capable of performing the role of `Pilot`, and 3 sub-teams capable of performing the role of `Crew`. These could be 3 entirely different sub-teams for each of the roles or there could be overlap. In this example, there are only 3 Martians within the spacecraft team and each is capable of performing all 3 roles.

The Spacecraft team definition also includes a `visit` method which posts a `PerformVisit` event using the `postWhenFormed` method. The `postWhenFormed` method puts the event in a special queue so that it gets posted asynchronously when the team has completed its team formation phase and built the initial role obligation structure.

The `Martian` team contains `#performs` role declarations to indicate that teams of this type are capable of performing the roles `Spokesperson`, `Pilot` and `Crew`.

Step 2: Create the main Java program and the initialisation file

Step 2.1: Create the main Java program

The main Java program must construct instances of the spacecraft and Martian teams. These will be the instances attached to the specific roles in the initialisation file. In the main program, the sub-teams must be constructed before the containing team, so that they already exist when the containing team attempts to build its role obligation structure.

```
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");  
    }  
}
```

Step 2.2: Create the initialisation file

In this example, it is assumed that this initialisation file is called `scenario.def`. It contains the following:

```
<Team :name "Enterprise"
  :roles (
    <Role :type "Spokesperson" :name "sp"
      :fillers (
        <Team :name "Dennis@%portal" >
        <Team :name "Ralph@%portal" >
        <Team :name "Jacquie@%portal" >
      )
    >
    <Role :type "Pilot" :name "pi"
      :fillers (
        <Team :name "Dennis@%portal" >
        <Team :name "Ralph@%portal" >
        <Team :name "Jacquie@%portal" >
      )
    >
    <Role :type "Crew" :name "cr"
      :fillers (
        <Team :name "Dennis@%portal" >
        <Team :name "Ralph@%portal" >
        <Team :name "Jacquie@%portal" >
      )
    >
  )
>
```

Note the relationship between the names of instances and roles in the main program and in the team definitions in the initialisation file. Also note that the team names are in the form `name@%portal` and that if your application is organised into packages, then the package details must be included in the type specifications.

Step 3: Create the role definitions files

There are 3 role definition files required in this example. They are:

Crew.role

```
public role Crew extends Role
{
  #handles event DoWatch wm;
}
```

Spokesperson.role

```
public role Spokesperson extends Role
{
  #handles event DoGreeting dg;
}
```

Pilot.role

```
public role Pilot extends Role
{
  #handles event PilotCraft st;
}
```

In all three cases, these roles indicate the downward interface between a team that can perform that role and a team that requires a sub-team to perform that 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 each specified event.

Role definitions can also include additional declarations which will be discussed in the chapter on *Roles*.

Step 4: Create the events

DoGreeting.event

```
event DoGreeting extends MessageEvent
{
    String planet;

    #posted as
    speakGreeting(String p)
    {
        planet = p;
    }
}
```

DoWatch.event

```
event DoWatch extends MessageEvent
{
    #posted as
    watch()
    {
    }
}
```

PerformVisit.event

```
event PerformVisit extends MessageEvent
{
    String planet;

    #posted as
    visitPlanet(String p)
    {
        planet = p;
    }
}
```

PilotCraft.event

```
event PilotCraft extends MessageEvent
{
    String planet;

    #posted as
    start(String p)
    {
        planet = p;
    }
}
```

Step 5: Create the plans used by the Martian sub-teams

WatchMonitor.plan

```
plan WatchMonitor extends Plan
{
    #handles event DoWatch dw;

    body()
    {
        System.out.println(getAgent().name()+" on watch");
    }
}
```

SpeakGreeting.plan

```
plan SpeakGreeting extends Plan
{
    #handles event DoGreeting dg;

    body()
    {
        System.out.println("Hello "+dg.planet);
        System.out.println("I am "+getAgent().name());
    }
}
```

Travel.plan

```
plan Travel extends Plan
{
    #handles event PilotCraft pc;

    body()
    {
        System.out.println(getAgent().name()+" flying craft to "+
            pc.planet);
        @waitFor(elapsed(10.0));
        System.out.println(getAgent().name()+
            " arriving at "+pc.planet);
    }
}
```

The 3 plans required are implemented as agent plans. This is because there are no sub-teams within the Martian teams, so there is no requirement to establish a task team to perform the task or for the new plan statements which enable coordinated activity between the sub-teams. `Teamplans` are only required when the plan requires sub-teams to perform roles on its behalf.

Step 6: Create the plan used by the spacecraft team

```

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

teampplan Visit extends TeamPlan
{
    #handles event PerformVisit pfv;
    #uses role Spokesperson sp as speaker;
    #uses role Pilot pi as pilot;
    #uses role Crew cr as crew;

    #uses interface Team team;

    /**
     * establish the task team.
     */
    #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 (r.state == Role.ACTIVE &&
                !busy.contains(r.actor))
            {
                busy.add(r.actor);
                return r;
            }
        }
        return null;
    }

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

        @parallel(ParallelFSM.ALL,false,null)
        {
            @teamAchieve(crew, crew.wm.watch());
            @teamAchieve(pilot, pilot.st.start(pfv.planet));
        };
    }
}

```

```
        @teamAchieve( speaker ,
                    speaker.dg.speakGreeting(pfv.planet) );
    }
}
```

The spacecraft team has `RoleContainer` members (one per role that it has declared that it requires). In the `teamplan` the first task is to iterate through the appropriate role containers to select the particular role object to perform the required roles for a particular instantiation of the plan. This forms the task team for the plan. The role object selected will allow access to both the containing team and sub-team involved in the relationship.

The `#uses` role declarations indicate that within this plan three sub-teams are required – one to fill the role of `Spokesperson`, one to fill the role of `Pilot`, and one to fill the role of `Crew`. In this example, iteration to make a selection occurs through the respective role containers.

As this plan requires that each sub-team only be responsible for one particular role, a "busy" vector is used to keep track of which sub-teams are already allocated to roles. Each instance of a role has details about the containing team and the sub-team. The role's `actor` member contains the name of the instance of the sub-team that is capable of performing the role. The selection of sub-teams to perform specific roles for this task occurs in the `teamplan`'s `establish()` method.

The `RoleContainer` base class contains a method (`tags()`) which returns its current role object tags as a `java.util.Enumeration` object. The role object tags relate to the role fillers or role performers and can be passed into the role container's `find()` method to return the `Role` object that corresponds to the tag.

In this example, the `establish()` reasoning method makes use of a method called `pickRole()`. The method begins by iterating through the required role container and then performing a `find()` to return the actual role object related to the tag. When selecting a suitable sub-team, it is a case of selecting the first role object which has a value of `Role.ACTIVE`, and which does not relate to a sub-team that has already been allocated (i.e. not already in the busy vector).

The test for `Role.ACTIVE` is not strictly necessary in this application as we do not have any dynamic attachment/detachment of teams in the role obligation structure. In this application, the formation of the role obligation structure will have been completed before the event is posted to activate the plan.

During the attachment/detachment of sub-teams to the role obligation structure, the role object can be in different states. The role object exists and is added to the container before the attachment handshaking between team and sub-team has completed; however, the sub-team may still refuse the attachment. Similarly, the role object exists, but is not active when the team has initiated a detachment process, because the sub-team may still be performing tasks in the role. The detachment cannot go ahead until the sub-team has finished all of the tasks in the role. The role object is marked as active when the role attachment procedure has completed, and before the role detachment procedure has started. If dynamic attachment/detachment is occurring in an application, it is not sufficient only to look at the presence of a role object to know whether or not the team it refers to is performing in that role, one should also test whether or not the role object is `ACTIVE`.

If the *task team* is successfully established, the plan body will be executed. This plan body illustrates how the `@teamAchieve` statement can be used in combination with the `@parallel` statement to coordinate the behaviour of the sub-teams. The `@parallel` statement operates like a control structure in which the body statements are executed as parallel tasks, while the `@parallel` statement itself is postponed until its termination condition holds. In this example, the arguments used in the `@parallel` statement are as follows:

mode: `ParallelFSM.ALL`

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.

termination condition: `false`

This means that the abort mechanism is turned off and not used (i.e. there is no termination condition). This is discussed in more detail in the *Agent Manual*.

notification: `null`

This argument is 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 (i.e. if the termination condition is encountered). If there is no termination condition, as in this example, this can be null.

The `@teamAchieve` declaration is used to activate a sub-team (role filler) by posting an event to the sub-team. The team that posted the event using `@teamAchieve` waits until the event has been processed.

In the Martian Visit example, the first argument is the `RoleType` instance obtained from the `RoleTypeContainer` instance. The second argument is an `event` instance being sent to the sub-team. In the example, the events are constructed using posting methods from event factories accessed via the sub-team `RoleType` instances.

Step 7: Compile and run the program

Step 7.1: Compile the program

To compile the example:

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

Step 7.2: Run the program

To run the program:

The team structure can be specified by using a Java property to specify an initialisation file that contains details of the teams. The most straightforward way of doing this is by associating a file to be read in with the `Team.Structure` property via the `-D` flag. In this example, the initialisation file was shown earlier. Assuming that the file is called `scenario.def`, the program runs as follows:

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

The output from the example is:

```
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
Hello Earth. I am Jacquie@%portal
```

2 Teams

In Teams, a team is a distinct BDI reasoning entity which is characterised by the roles it performs and the roles it requires others to perform. The formation of a given team is achieved by attaching sub-teams (either statically or dynamically) capable of performing the roles required by the team. Note that a sub-team may be attached to more than one role in a containing team. As the sub-teams of the given team may require roles to be performed on their behalf, a multi-level hierarchy (or perhaps a more complex structure) may result. The team is automatically provided with objects to hold the actual sub-team selections. These objects are known as *role containers*.

2.1 Team Definition

Team definitions take the form shown below:

```
team TeamType extends Team
{
    // team declarations and definitions
    // all JACK agent declarations can also be used

    // constructor
    public TeamType(String name)
    {
        super(name);
    }
}
```

Each component of this definition is explained in the following table:

Syntax Term	Description
<code>team</code>	A Teams Language keyword used to introduce a Team definition.
<code>TeamType</code>	The name of your derived <code>Team</code> class
<code>extends Team</code>	This part of the statement plays the same role as in Java – it indicates that the role being defined inherits from a Team's base class called <code>Team</code> . The <code>Role</code> base class implements all the underlying methods that provide a team's core functionality.

Table 2-1: Components of a Team definition

2.2 Team Declarations

The team type definition has the same range of declarations available as an agent definition. In addition, it has declarations that are specific to the team concept. These are the declarations that relate to roles required, roles performed and belief propagation in the team structure. The team declarations are described in the following sub-sections.

#requires role RoleType reference(min,max)

The statement of the form `#requires role RoleType reference(min,max)` is a declaration that teams of the type being defined require a sub-team or sub-teams in the given role, `RoleType`.

Technically, the `#requires role` statement adds a field to the team class of name `reference` and type `RoleTypeContainer`. The arguments specify the upper and lower bounds for the number of role performers. A value of zero for `max` specifies the default upper bound. A zero upper bound is an unlimited upper bound. A zero for the minimum value specifies a zero lower bound.

The `RoleTypeContainer` type is created automatically by the compiler when there is a role defined of type `RoleType`.

A `#requires role` statement:

- declares a local reference to the role container which will contain the instances of the role objects for sub-teams attached to the team in this particular capacity;
- is an implicit declaration that the team performs the *peer role*; that is, it can and must handle the events posted within the role.

#performs role RoleType

This statement is a declaration that the team of the type being defined is able to perform a given role, `RoleType`.

A `#performs role` statement:

- adds to the team all event handling and posting declarations specified in the role type definition;
- implicitly requires plans to handle the events declared as handled in the role type;
- implicitly allows plans to post the events declared as posted in the role type.

#synthesizes teamdata DataType ref (r1.s1, r2.s2, ...)

This is a declaration that can be found within a team that is a *role tenderer*. A role tenderer will have `#requires role` declarations specifying that it requires sub-teams to perform particular roles on its behalf.

The declaration has the form

```
#synthesizes teamdata DataType ref(r1.s1, r2.s2, ...)
```

where

- `DataType` is the type of the `teamdata`. This is described in the chapter on *Team Belief Connections*.
- `ref` is a reference name.
- `ri` is the required role container reference name.
- `si` is the reference name of the data used in the corresponding `#synthesizes teamdata` declaration in the `Role` definition.

Note that one or more belief connection sources are required. Belief propagation is described further in the chapter on *Team Belief Connections*.

#inherits teamdata DataType ref (r1.s1, r2.s2, ...)

This is a declaration that can be found within a team that is a *role performer*. A role performer will have `#performs role` declaration(s) specifying that it can perform a particular role on behalf of a role tenderer.

The declaration has the form

```
#inherits teamdata DataType ref(r1.s1, r2.s2, ...)
```

where

- `DataType` is the type of the `teamdata`. This is described in the chapter on *Team Belief Connections*.
- `ref` is a reference name.
- `ri` is the role type performed.
- `si` is the reference name of the data used in the corresponding `#inherits teamdata` declaration in the `Role` definition.

Note that one or more belief connection sources are required. Belief propagation is described further in the chapter on *Team Belief Connections*.

2.3 Team Management

2.3.1 Team Construction

The base class `Team` has a constructor taking a `String` argument that represents the name of the team to be constructed. A team definition must thus include a constructor that invokes the base class (`Team`) constructor with the team name provided. As part of the construction, the kernel posts a `TeamFormationEvent` for establishing the team structure (also known as the role obligation structure), and a `TeamStartEvent` to trigger application level initialisation.

The `TeamFormationEvent` extends `MessageEvent`, but uses BDI behaviour with the following BDI behaviour attributes set:

Behaviour Attribute	Setting	Effect
<code>Recover</code>	<code>repost</code>	The event is reposted on plan failure, so that another applicable plan can be tried. If no new applicable plan is found, then event processing fails.
<code>ApplicableSet</code>	<code>once</code>	The applicable set is computed only once, rather than being recomputed after each plan failure. On event failure, the next applicable plan is selected from the set computed initially for the event.
<code>ApplicableChoice</code>	<code>first</code>	The plan instance generated by the <code>#uses</code> plan declaration that occurs first in the body of the agent or capability is chosen.
<code>ApplicableExclusion</code>	<code>failed</code>	Plans that have failed are excluded from the applicable plan set.
<code>PlanBindings</code>	<code>single</code>	One applicable plan of each relevant plan type is added to the applicable plan set.

Table 2-2: The BDI behaviour attributes for the `TeamFormationEvent`

There is a default plan for the `TeamFormationEvent`. This plan uses the default team manager, described in the next section.

2.3.2 The Team Manager

The team manager is responsible for coordinating the assembly of the team structure. This can be done using team initialisation files, or in the code.

2.3.3 Initial Team Formation

The team structure can be specified by using a Java property to point to a file that contains details of the teams required. The most straightforward way of doing this is by associating a file to be read in with the `Team.Structure` property via the `-D` flag. This flag is used when running your application with Java.

An example of the use of this flag follows:

```
java -DTeam.Structure=filename ApplicationClass
```

The team initialisation file is specified using the JACOB Object Modelling Language. The format of a team initialisation file is as follows.

```
<Team :name "TeamName"
  :roles (
    <Role :type "RoleType" :name "roleinstance_ref"
      :fillers (
        <Team :name "TeamName">
      )
    )
  >
)
```

The following table details the meaning of each of the syntactical elements in the team initialisation file:

Entity	Label	Code Entity Mapping
Team	<code>:name</code>	Class name of the Team.
	<code>:roles</code>	An aggregation of role bindings.
Role	<code>:type</code>	Class name of the role.
	<code>:name</code>	A reference to the role instance as it appears in code.
	<code>:fillers</code>	A section that specifies the fillers of the role. This can be a reference to a team that is specified later in the file or the team can be fully specified inline.

Table 2-3: The syntactical elements in the team initialisation file

Note: In the above example, the `TeamName` for the `fillers` is mentioned. This format requires that the `TeamName` be fully specified in another separate `Team` entry in the team initialisation file.

Alternatively you can specify the entire team structure inline.

This allows the entire team structure to be specified as one monolithic `Team` entry, or as a number of distinct team/role relationships.

The initialisation file can specify a structure that is a superset of those team entities that you declare in code. This allows the provision of a central, predetermined and precise structure for the team. Using this method, the team structure can be better planned and changes can be made quickly and easily in the one location.

Relationships described in the initialisation file as part of role and team relationships must be legal according to the existing `#requires role`, `#performs role` and `Class (Team and Role)` declarations in code.

Note:

- The `roleinstance_ref` that occurs in the initialisation file *must* match the reference name that appears in your JACK code.
 - The `RoleTypes` must be given with their package path (i.e. their full class names).
 - `TeamNames` must be full team instance names (i.e. the name plus the portal name).
-

2.3.4 Dynamic Team Formation

It is possible to write custom team formation plans to allow any part of a team structure to be constructed dynamically at runtime. This is a powerful technique which allows, for example, dynamic specification of team structure at the top level while the lower level structure is handled by the team initialisation file.

To specify that a manually formed team is preferred, add a plan to the team's plan set to handle the `TeamFormationEvent`. The default plan has a rank of zero, so adding any plan with the standard plan rank (5) will cause this default to be overridden.

The plan to handle the team formation event is then responsible for specifying the structure of roles as they relate to the current team. Specification of team structure involves attaching teams to roles and roles to teams. This can be achieved by posting a `RoleControl` event with its `assign(String role, String container, String actor)` posting method. To detach teams the `RoleControl` event is posted with its `revoke()` posting method. This is discussed in more detail in the chapter on *Team Formation*.

2.3.5 Initialising Teams

If the plan that processes the `TeamFormationEvent` succeeds, a `StartTeamEvent` is posted by the kernel. Its purpose is to enable team-specific initialisations.

The default plan handling `StartTeamEvent` succeeds without doing anything.

This plan could be overridden to start an application. Any plan that reacts to a `StartTeamEvent` will do as the default plan has a plan rank of zero. Any new plan you write with a default rank of 5 will therefore be executed in preference to the default plan.

`StartTeamEvent` extends `Event`, but uses BDI behaviour with the following non default BDI behaviour attributes:

Behaviour Attribute	Setting	Effect
<code>ApplicableSet</code>	once	The applicable set is computed only once, rather than being recomputed after each plan failure. On event failure, the next applicable plan is selected from the set computed initially for the event.
<code>ApplicableChoice</code>	first	The first plan instance generated by the <code>#uses plan</code> declaration that occurs first in the body of the agent or capability is chosen.
<code>PlanBindings</code>	single	Only one applicable plan of each relevant plan type is added to the applicable plan set.

Table 2-4: The BDI behaviour attributes for the `StartTeamEvent`

2.4 The Team Base Class

The `Team` class is used as the base class for teams. It provides the implementations for the team 'lifetime' and includes the core team capabilities. The following methods are available within a team:

```
void canPerformRole(String role, boolean yes)
//
// A support method whereby a team marks whether it actually can
// perform a role. See also RoleContainer.active.
//

RoleContainer findContainer(String role,String cntr)
//
// Looks up the role container matching a #required role
// declaration.
//

RoleContainer findContainer(String role)
//
// Looks up the role container matching a #performed role
// declaration.
//

Role findPerformedRole(String team,String role,String cntr)
//
// Looks up the role object that for a performer represents the
// activated obligation of performing the given role for the named
// team, and that team's container.
//

Role findRequiredRole(String team,String role,String cntr)
//
// Looks up the role object that for a role tenderer represents the
// activated role obligation of a team performing the given role in
// the given container.
//

RoleInfo [] getRoles()
//
// Gets the RoleInfo table for this team. This contains
// the RoleInfo object corresponding to the #requires role
// statements, and is used by the default team formation procedure.
//

void postWhenFormed(Event e)
//
// This is the same as the postEvent method in an agent,
// but it waits until the team formation stage is
// complete before posting the event.
//

boolean rolesInitialized(RoleInfo[] roles)
//
// A reflection support method that uses the RoleInfo table as
// guide in determining whether required roles have been initialised.
//
```

```
Team(String n)
//
// The base class constructor. All team instances need to be assigned
// unique names at construction. The team will be registered with the
// communication system under that name, and other agents/teams can
// thus send messages to this team using that name.
//
```

3 Roles

A `role` in Teams is a distinct entity which contains a description of the facilities that the two participants in a team/sub-team relationship must provide. A role defines a relationship between teams and sub-teams. The relationship is expressed in terms of the event and belief exchanges implied by the relationship.

The role construct functions at two levels:

1. To specify the requirements of a role for the tenderer (the team requiring the role) and the filler (the team providing the role). This specification allows run-time checking of the events that the role tenderers and fillers claim to handle and post. The role functions as an interface definition that declares what an entity that fills a given role must be capable of doing in terms of events handled and posted, and in terms of belief propagation. It is also necessary for the role tenderer to be able to handle events declared as posted, and post events declared as handled in the role specification.

Like an interface in Java, the role specification does not contain implementation – only a description of the facilities that the two participants in the role relationship must provide.

2. A role operates in a similar manner to a proxy by facilitating sub-tasking between participants in the role relationship. Specifically, role instances are invoked in plans to allow `@teamAchieve` statements to be issued to role performers.

3.1 Role Definition

Role definitions take the form shown below:

```
role RoleType extends Role
{
    // declarations of events handled by the role performer
    // declarations of events posted by the role performer
    // declarations of teamdata synthesized from the role
    // performer
    // declarations of teamdata inherited by the role performer
    // declarations of role container methods and members
    // other Java methods and members
}
```

Each component of this definition is explained in the following table:

Syntax Term	Description
<code>role</code>	A Teams Language keyword used to introduce a Role definition.
<code>RoleType</code>	The name of your derived <code>Role</code> class
<code>extends Role</code>	This part of the statement plays the same role as in Java – it indicates that the role being defined inherits from a Teams base class called <code>Role</code> . The <code>Role</code> base class implements all the underlying methods that provide a role's core functionality.

Table 3-1: Components of a Role definition

The compiler generates two classes from a role definition. The first class is the `RoleType`. The second generated class class is a specialised "container" for instances of the `RoleType` called `RoleTypeContainer`. This latter class is used in Teams and TeamPlans to group the performers of a role.

When a team is declared to require a role (e.g. `RoleType`), the resulting Java class for the team will include a field of the corresponding container type, `RoleTypeContainer`. The access to individual role performers is indirect through such a container.

Further, in a teamplan, the declaration of using a role results in a `RoleType` member or `RoleType` array member local to the plan. This gives a modelling advantage by allowing teamplans to operate with selected, transient sub-groupings that only exist during and for the purpose of carrying out the teamplan.

3.2 Role Declarations

The role functions as an interface definition that declares what an entity that fills a given role must be capable of doing in terms of events handled and posted, and in terms of belief propagation. It is also necessary for the role tenderer to be able to handle events declared as posted, and post events declared as handled in the role specification.

In general, a role definition will require declarations for the following:

- `Events` that the role performer must be able to handle and that the role performer may post upward to the role tenderer.
- `Teamdata` that the role performer may inherit from the role tenderer or that the role tenderer may synthesize from the role performer.

-
- `Role Container` methods that allow the definition of methods and members to be added to the automatically created `RoleTypeContainer` class.

Each declaration is described in the following sub-sections:

#handles event EventType reference

This statement declares that a role performer must be capable of handling an event of `EventType`. The `reference` becomes a data field referring to the appropriate event instance factory to be used by the JACK kernel. It is through this reference that the event's posting method can be accessed when it is necessary to create an event instance to be sent from the tendering team to the performing team.

The role events are sub-tasked through `@teamAchieve` statements. The role tenderer can sub-task a role performer with the events declared as handled.

#posts event EventType reference

This statement declares that a role tenderer must be capable of handling an event of `EventType`. `reference` becomes a data field of the generated `EventType` class initialisation to the appropriate event instance factory by the JACK kernel. It is through this reference that the event's posting method can be accessed when it is necessary to create an event instance to be sent from the performing team to the tendering team.

The role events are sub-tasked through `@teamAchieve` statements. The role performer can sub-task a role tenderer with the events declared as posted.

#synthesizes teamdata DataType reference

This is a statement for declaring a synthesizing team belief connection. `reference` identifies the beliefset (of type `DataType`) to be synthesized. There must be a corresponding declaration for the teamdata to be populated through this belief propagation in the team definition. It will be of the form:

```
#synthesizes teamdata SynthData data(role_ref.reference)
```

where `role_ref` refers to the reference in the `#requires` declaration for the role in the tendering team definition and `reference` is the reference in the `#synthesizes` declaration in the role definition. The data is directed from the role performing sub-team(s) to the tendering team.

This is described in more detail in the chapter on *Team Belief Connections*.

#inherits teamdata DataType reference

For an inheriting belief connection, a role definition needs to include an `#inherits teamdata DataType reference` statement detailing the type and reference name of the source beliefset/teamdata concerned. This statement is similar to the `#synthesizes teamdata` role statement, but is directed from the tendering team to the performing team.

This is described in more detail in the section on *Team Belief Connections*.

#container method

This statement allows the definition of methods to be added to the `RoleTypeContainer` class. The statement form is similar to a reasoning method in a JACK plan. An outline is given below:

```
#container method
public boolean doSomething(int x)
{
    ...
}
```

The generated `RoleTypeContainer` class extends a base class named `RoleContainer`. This base class provides a number of useful methods for inspecting the container and accessing the role performers. These are described in the section on the `RoleContainer Base Class`. The `#container method` statement may be used to provide user-defined methods in the role container.

#container member

This statement allows the definition of data members to be added to the `RoleTypeContainer` class.

The statement has the following form:

```
#container member <variable declaration>
```

For example,

```
#container member public MyDataType my_data = initial_value;
```

3.3 The Role Base Class

The `Role` base class provides implementations needed for maintaining role relationships between teams. Role definitions extend `Role` with specific declarations, allowing the kernel to review and enforce type safety in terms of inter-team event handling and posting.

In a program, role objects have three different uses.

1. The roles performed by a team are represented by role objects. The event handling and event posting of these roles are added to the requirements of the team instance that are checked at runtime.
2. The roles required by a team are represented by role container objects, which keep role objects representing the particular fillers attached.
3. When a team task is started, by one team issuing a `@teamAchieve` goal to a sub-team, then the team task in the sub-team is associated with a pair of role objects:
 - one role object is to represent the role that the sub-team is acting within; and
 - another role object is to represent the *peer role* that the tendering team is automatically attached to by virtue of utilising the role of the sub-team.

All three uses will use the specific role types that extend the `Role` base class. The base class in itself merely contains common data members, a few common methods and the service methods that specific role classes will override.

The `Role` class implements the following interface:

```
String actor
//
// Keeps the name of the team that the role object is a proxy for.
//

boolean mirror
//
// Is true when the role object identifies
// the role tenderer.
//

Role peer
//
// This is set only for role objects of team tasks, where it holds
// the peer role object for the team task.
//

int state
//
// Keeps the role object's activity state, which is one of INACTIVE,
// ACTIVE or DETACHED.
// This is discussed further in the section on
// Team Formation.
//

String tag
//
// Identification of the role object. This is assigned at
// role object construction to a unique identification number.
//
```

Roles

TaskJunction tasks

```
//  
// Keeps track of team tasks in progress under this role. Note that  
// for a team task, it is the peer of its role object that represents  
// the role obligation, and thus where the tasks performed under that  
// obligation are tracked.  
//  
//
```

Cursor noTasks()

```
//  
// Returns tasks.idle(). This allows a team to check whether  
// or not it is performing tasks within a particular role.  
// noTasks will be true when it is not performing any  
// tasks within the role.  
//  
//
```

void int setState(int n)

```
//  
// Method to set the role object's activity state.  
// This method would only be used explicitly for non-standard role  
// change procedures.  
//  
//
```

3.4 The RoleContainer Base Class

The `RoleContainer` class is used as the base class for all role containers. It contains common members and methods, and stubs to be overridden by specific role containers.

The `RoleContainer` class implements the following interface:

boolean active

```
//  
// This is set to true by default. A team can set it to false to  
// prevent any new tasks from being started under that role. If it  
// is set to false, any pre-existing tasks will continue to be  
// performed. The standard role assignment protocol looks at this  
// and refuses a role assignment when the performed role is not  
// active.  
//  
//
```

String name

```
//  
// This is the reference name associated with this role container.  
// For a performed role, the reference name has the form  
// "__HR_xxx_performs" where "xxx" is the role type for this  
// container. For a required role, the reference name is given by  
// the programmer.  
//  
//
```

String role

```
//  
// The type name of role objects that the container is intended for.  
//  
//
```

Team team

```
//  
// The team that the container belongs to.  
//  
//
```

```
int min = 0
//
// The minimum number of role objects that are expected to be in this
// role container.
//

int max = 0
//
// The maximum number of role objects that are expected to be in this
// role container, or zero for unlimited.
//

Role find(String actor)
//
// Returns the role object for an actor if the container contains it.
// The role container keeps roles tagged by the actor, and it can
// therefore only contain one role object for any given actor.
//

int size()
//
// Returns the number of role objects added to the role container.
//

Enumeration tags()
//
// Returns the current role object tags as a java.util.Enumeration
// object. These are also the actors defined as role fillers.
//

Role nextFiller()
//
// Uses nextTag() to find an active role filler.
//

String nextTag()
//
// The nextTag() method manages a local enumeration of tags
// to provide the available tags one at a time. If roles are added or
// removed the enumeration is reset, otherwise it will cycle through
// the tags indefinitely.
//

boolean rolesInitialized()
//
// Returns true if the min/max constraints are met.
//
```

3.5 The Generated RoleType Class

A role definition for `RoleType` results in two classes:

- a class named `RoleType` that extends the `Role` base class; and
- a class named `RoleTypeContainer` that extends the `RoleContainer` base class.

The `RoleType` classes provide runtime type checking methods that the kernel uses.

3.6 The Generated RoleTypeContainer Class

The generated `RoleTypeContainer` class extends `RoleContainer` and provides a method `createRole()` for constructing `RoleType` objects within the context of the container.

4 TeamPlans

A `teamplan` specifies how a task is achieved in terms of one or more roles. It typically contains steps determining which of the sub-teams nominated to perform the roles in the role obligation structure will in fact perform each role (a process known as *task team formation*). It also dictates the steps directing each sub-team to achieve specific goals.

Teams provides additional constructs to support both activities (the `establish()` reasoning method and the `@teamAchieve` statement). The JACK `@parallel` statement supports non-sequential coordination of sub-team behaviour. As team behaviour (embodied in a `teamplan`) is specified in terms of roles, it is decoupled from the actual sub-team behaviour. Thus team behaviour can be specified and understood independently of sub-team behaviour.

4.1 TeamPlan Definition

TeamPlan definitions take the form shown below:

```
teamplan PlanType extends TeamPlan
{
    #handles event EventType event_ref;

    // possible declarations about required roles etc.

    // Plan method definitions, reasoning methods
    // and JACK Agent Language declarations describing
    // relationships to other components etc.

    // optional relevant method
    static boolean relevant (EventType event_ref)
    {
        // code to determine if the plan
        // is relevant
    }

    // optional context method
    context()
    {
        // logical condition to determine which
        // plan instances are applicable
    }

    // optional establish method
    #reasoning method establish()
    {
        // code to establish the task team for the task
    }
}
```

```

body()
{
    // The plan body. This describes the actual steps
    // an agent performs when it executes this plan.
    // It includes Java code, JACK Agent Language
    // @-statements, and in addition @teamAchieve
    // statements for use within TeamPlans.
}
}

```

Each component of this definition is explained in the following table:

Syntax Term	Description
<code>teamplan</code>	A Teams Language keyword used to introduce a TeamPlan definition.
<code>PlanType</code>	The name of your derived <code>TeamPlan</code> class.
<code>extends TeamPlan</code>	This part of the statement plays the same role as in Java – it indicates that the <code>teamplan</code> being defined inherits from a Teams Language base class called <code>TeamPlan</code> . The <code>TeamPlan</code> base class provides the generic plan processing implementation and the overridable stub for task team formation.
<code>#handles event</code> <code>EventType event_ref</code>	Specifies the event type that this plan handles. The plan may place further constraints on its applicability via the <code>relevant()</code> and <code>context()</code> methods.
<code>static boolean</code> <code>relevant(EventType</code> <code>event_ref)</code>	Code to determine if the plan is relevant for the instance of the event being handled.
<code>context()</code>	Logical condition to determine which plan instances are applicable.
<code>#reasoning method</code> <code>establish()</code>	Code to establish the task team for the plan.
<code>body()</code>	Describes the actual work done by the team when the plan is executed. It is the plan's top-level reasoning method. If it succeeds the plan succeeds. If it fails the plan fails.

Table 4-1: Components of a TeamPlan definition

4.2 TeamPlan Declarations

The teamplan can use any of the declarations that are available in JACK agent plans. In addition they can have declarations relating to the roles required to establish the task team for the plan and a declaration specifying the role for which the plan is applicable.

#requires role RoleType rolecontainer_ref as role_ref

This declaration states that the current plan requires an instance of the role `RoleType`. This declaration is used in the teamplans of the role `tenderer`, where the reference to the role container instance `rolecontainer_ref` can be used to access the role container to find an appropriate instance of the role `RoleType` to satisfy this role in the task team. This selection of the role instance occurs during the task team establishment phase. During this phase, `role_ref` will be set to refer to the selected performer and will subsequently be used as a reference to this role instance in the plan. `role_ref` is used to issue a `@teamAchieve`. This is because the team executing the plan needs a reference to the role filler (via the role reference) in order to issue the `@teamAchieve`.

The `#requires role` declarations are used instead of the `#uses role` declarations when the plan is to use the default `establish` reasoning method to select the role instance.

Events are posted to the role filler by the `@teamAchieve` statement via the role specification. As such, the programmer need not declare that the event being handled by the role filler is *posted* in the plan.

Each component of the `#requires role` declaration is explained in the following table:

Component	Meaning
<code>#requires role</code>	Specifies that the plan makes use of this role.
<code>RoleType</code>	The role type that is used by the plan.
<code>rolecontainer_ref</code>	A local reference that is used to perform operations on the role container.
<code>role_ref</code>	A local reference that is used to perform operations on the role instance.

Table 4-2: Components of the `#requires role` declaration

#requires role RoleType rolecontainer_ref as role_ref (size)

This form of the `#requires role` declaration is the same as the previous version except that it specifies that the plan now needs `size` performers from the `rolecontainer_ref`. During the team establishment phase, the variable `role_ref` will be set to an array that contains the selected performers.

#uses role RoleType rolecontainer_ref as role_ref

This declaration states that the current plan makes use of an instance of the role `RoleType`. The declaration is used in the `teamplans` of the role `tenderer`, where the `reference` to the role container instance `rolecontainer_ref` can be used to access the role container to find an appropriate instance of the role `RoleType` to satisfy this role in the task team. `role_ref` will be used as a reference to this role instance in the plan. `role_ref` is used to issue a `@teamAchieve`. This is because the team executing the plan needs a reference to the role filler (via the role reference) in order to issue the `@teamAchieve`.

The `#uses role` declarations are used instead of the `#requires role` declarations when the plan overrides the default `establish` reasoning method to select the role instance.

Events are posted in the role filler by the `@teamAchieve` statement via the role specification. As such, the programmer need not declare that the event being handled by the role filler is posted in the plan.

Component	Meaning
<code>#uses role</code>	Specifies that the plan makes use of this role.
<code>RoleType</code>	The role type that is used by the plan.
<code>rolecontainer_ref</code>	A local reference that is used to perform operations on the role container.
<code>role_ref</code>	A local reference that is used to perform operations on the role instance.

Table 4-3: Components of the `#uses role` declaration

#uses role RoleType rolecontainer_ref as role_ref (size)

This form of the `#uses role` declaration is the same as the previous version except that it specifies that the plan now needs `size` performers from the `rolecontainer_ref`. During the team establishment phase, the variable `role_ref` will be set to an array that contains the selected performers.

#uses role RoleType rolecontainer_ref

This last *anonymous* role usage declaration form provides direct access to the team's role container for reviewing and selecting performers, and for team-level manipulation of resources, such as assignment or revocation of roles. The container is referred to with its team reference name, i.e. `rolecontainer_ref`.

#applicable_for role RoleType roleinstance_ref

This declaration occurs in plans of the role filler. It indicates that the plan should not only react to the event it handles, it should also test for applicability based on the currently active role.

Component	Meaning
#applicable_for role	Specifies that this plan is only applicable for certain roles.
RoleType	The role type that this plan is applicable for.
roleinstance_ref	A local reference that is used to perform operations on the role instance.

Table 4-4: Components of the #applicable_for role declaration

#applicable_from role RoleType roleinstance_ref

This declaration occurs in plans of the role tenderer. It indicates that the plan should not only react to the event it handles, it should also test for applicability based on the role relationship it is currently acting under. This form of applicability declaration checks that it has a peer relationship with the team that initiated the @teamAchieve.

Component	Meaning
#applicable_from role	Specifies that this plan is only applicable for certain peer role relationships.
RoleType	The role type that this plan is applicable for.
roleinstance_ref	A local reference that is used to perform operations on the role instance.

Table 4-5: Components of the #applicable_from role declaration

4.3 Task Team Establishment

The *task team establishment* stage is an initial execution stage for a teamplan, for the purpose of establishing which particular role performers to use for the plan. Technically, the task team establishment stage is achieved by a reasoning method that is performed prior to the plan body when a plan is chosen for execution. Task team establishment may fail, in which case the plan fails.

The task team establishment method is defined in a teamplan as a reasoning method:

```
#reasoning method establish()  
  
{  
    ...  
}
```

There is a default task team establishment method, `defaultEstablish()`, which fills all the `#requires role` usages with distinct performers. They are selected by repeatedly calling the `nextFiller()` method of the appropriate role containers. The default establishment only selects active role objects (discussed in the chapter on *Team Formation*), and requires all fillers to be distinct. The `#uses role` declarations are then left unfilled; for the plural case, the array is constructed though left unfilled.

An explicit `establish()` reasoning method will override the default task team establishment method and be solely responsible for identifying and assigning task team role fillers.

Alternatively, an application may override a role container's `nextFiller()` method by means of the `#container method` statement in the role definition.

4.4 TeamPlan Members and Methods

The TeamPlan has access to the same members and methods described in the chapter on JACK Agent plans in the *Agent Manual*. In addition, the TeamPlan has the `#reasoning method establish()` which was described in the section on *Task Team Establishment*.

4.5 Team Goal Handling

The Team Modelling Framework includes all JACK BDI programming facilities, and provides extra team goal handling support through the additional `@teamAchieve` statement.

The `@teamAchieve` statement is used in a teamplan to sub-task a goal for a team member. The statement may then succeed or fail depending on whether the team member succeeded or failed in reaching the goal. The infrastructure deals with the necessary inter-team coordination, and in particular, takes care of the required task control to deal with all cases of success, failure, or exception propagation.

The JACK `@parallel` statement is used in a teamplan as a program control structure to sub-task goals for several team members in parallel, or more precisely, to progress on several branches of activity in the teamplan in parallel. The success or failure of the statement depends on the successes and failures of the parallel branches involved. The programmer specifies whether all branches need to succeed or whether it is sufficient that at least one branch succeed. The programmer also specifies whether to wait for all branches to complete before the statement completes, or whether to complete the statement as soon as possible (e.g. with the first successful branch, if the success of one branch is sufficient).

The `@parallel` statement provides a very powerful mechanism for expressing teamplans. The implied task synchronisation reduces the effort of programming coordinated activity, in particular while focusing on the "success paths". Recovery procedures, contingency planning and their effect on coordination, require careful design and use of the task control statements available in JACK.

The underlying concept is that in failing a sub-task, a sub-team may qualify the failure, i.e. "fail for a reason", for which the teamplan might include a contingency. This is a slightly different execution model than BDI, where failing to complete a plan results in a reposting of the goal, to allow another plan to be used. For team tasks, the recovery on failure needs to be dealt with at the teamplan level as it, for instance, may involve re-tasking other sub-teams as well as the sub-team failing its task. At the same time, the recovery may also allow some other parallel tasks to progress without interrupts.

Further, the choice of which particular response to make at the team level in reaction to the failure of a sub-team to complete its task is a dynamic choice that may depend both on any partial success the sub-team has had towards its goal, and the more global situation at hand, including the current state of other sub-teams.

4.6 TeamPlan @-statements

The teamplan reasoning methods can use any of the @-statements that are available in JACK agent plans. In addition they can combine the `@teamAchieve` and JACK `@parallel` statements to coordinate and sub-task sub-teams that have been selected to perform particular roles as part of the teamplan's task team.

The `@teamAchieve` statement is described in more detail in the next sections.

4.6.1 The `@teamAchieve` Statement

The `@teamAchieve` declaration is used to activate a sub-team (role filler) by posting an event to the sub-team. The team that posted the `@teamAchieve` then waits until the event has been processed.

Note: `@team_achieve` has been deprecated in favour of `@teamAchieve`.

The `@teamAchieve` has the following form:

```
@teamAchieve(roleinstance_ref, EventInstance)
```

Each parameter of the above definition is described in the following table:

Component	Meaning
@teamAchieve	Introduces the @teamAchieve statement, which issues a directive to a sub-team, via the role.
roleinstance_ref	A local reference to the role object that defines the relationship between the team (role tenderer) and sub-team (role filler). The roleinstance_ref is obtained from one of the following statements; #uses role, #requires role, #applicable_for role OR applicable_from role.
EventInstance	EventInstance is a reference to an event derived from MessageEvent. It is declared and instantiated as for a normal JACK statement (see, for instance, @send). The event being sent to the sub-team (role filler) must be declared as being #handles event by the role.

Table 4-6: Components of the @teamAchieve statement

The EventInstance used in the @teamAchieve is obtained by accessing the event declaration in the role that the plan uses. For example, a plan might contain the following lines:

```

teamplan Plan9 extends TeamPlan {
    #requires role RoleA rolecontainer as role_ref;
    body()
    {
        ...
        @teamAchieve(role_ref, role_ref.eventref.postingMethod());
        ...
    }
}

```

This example would require that a role, RoleA, had declared an event with an event reference, eventref. The posting method of this event is called to generate an event instance inline.

Optionally, the event can be pre-constructed and a reference kept locally in the plan. This local reference to the event instance could be used to check the value of event members after the @teamAchieve has returned.

As mentioned previously, @teamAchieve suspends the execution of the containing teamplan until the event has been processed in the sub-team. A @teamAchieve terminates successfully if the event has been successfully handled by the sub-team, otherwise it fails.

The `@teamAchieve` statement can also be used to post events back to the peer of the team performing the plan, with the following syntax:

```
@teamAchieve(roleinstance_ref.peer, EventInstance)
```

4.6.1.1 Getting Return Values Through `@teamAchieve`

The technique for communicating the results of the processing of a `@teamAchieve` event to the commanding team is as follows:

The sub-team plan handling an event sub-task by a `@teamAchieve` can change the fields of its local copy of the event. When the plan succeeds, these fields are copied back into the event instance that was originally used in the teamplan executing the `@teamAchieve`. Thus, if a handle to that event instance is maintained, it is then possible to retrieve all fields that have been changed.

4.6.1.2 Exception Propagation for `@teamAchieve`

During the handling of an event sub-task with `@teamAchieve`, a team may throw a Java exception. If this is not caught by the handling plan, it is propagated back to the team that is executing the `@teamAchieve` statement in the following way:

- If the exception is a `TeamException`, then the same exception is thrown to the teamplan executing the `@teamAchieve`;
- If the original exception is not a `TeamException`, then a `TeamError` is thrown to the teamplan executing the `@teamAchieve`.

A `@teamAchieve` may also be interrupted by an exception thrown to it by a parallel execution branch in the teamplan. In this case, the infrastructure notifies the active sub-team so that a `TeamAbort` is thrown to the plan handling the `@teamAchieve` event.

5 Team Belief Connections

The notion of team belief connections focuses on how beliefsets of teams and sub-teams may be connected through role relationships. Team Belief Connections are either directed 'upwards', synthesizing beliefs of sub-teams into a containing team, or 'downwards', allowing sub-teams to inherit beliefs from a containing team. In both instances, the connection is between sub-teams that fill specified roles in the containing team's Role Obligation Structure and the containing team.

The propagation dynamics of a belief connection, both synthesizing and inheriting, are *asynchronous*, and allow for source-end filtering to avoid or delay propagation, as well as target end synthesizing. Default plans for source-end filtering are provided – these plans can be overridden by the user if desired. Target end synthesis is implemented using the `teamdata` construct.

5.1 Source Data Definition

A generic capability for propagating changes is provided as part of the beliefset infrastructure. This propagation includes filtering when the `#propagates changes` declaration is used with an optional event type as described below.

Beliefset types to be connected must include declaration statements of the following form:

```
#propagates changes ;
```

or

```
#propagates changes EventType ;
```

in their definition.

A `#propagates changes` statement marks that the beliefset may be a source beliefset in a team belief connection, and it provides an implementation of the connection dynamics, so that changes to the beliefset are propagated correctly.

Team Belief Connections

If an event type is specified in a `#propagates` changes statement, it will be sub-tasked to allow the team to block propagation selectively. Event types used for propagation in this way must implement the `PropagationEvent` interface:

```
public interface PropagationEvent {  
  
    public Event propagate(  
        String team,  
        Tuple newTuple,  
        Tuple keyDiscard,  
        Tuple negateDiscard,  
        BeliefState truthValue,  
        boolean wasAssert  
    );  
}
```

The parameters are described below:

Parameter	Description
team	The name of the sub-team whose change is propagated.
wasAssert	Whether the tuple concerned was asserted or retracted.
truthValue	The resulting <code>BeliefState</code> .
newTuple	The tuple concerned.
keyDiscard	The opposing tuple retracted by virtue of the key constraint, if any.
negateDiscard	The contradictory tuple being retracted, if it was believed.

Table 5-1: Parameters for the `propagate` method

Note that `propagate` returns an event – this will typically be achieved by defining a suitable posting method in the usual way and then invoking the posting method from within `propagate`. An example of how to construct such an event is provided in the synthesizing belief connection example presented later in this chapter.

If changes to a belief type are to be propagated through the team hierarchy, the fields must be transportable. This means that either the fields must be declared as `JACOB` objects in an api file (refer to the *JACOB Manual* for more details) or they must be defined as a Java class that implements `java.io.Serializable`.

Note: If a propagated belief type is instantiated in a capability rather than at the agent level, then the capability must declare that it posts the propagation event. If the user has not defined the propagation event, then the event to be posted is `aos.team.ChangePropagation`.

5.2 Target Data Definition

The *teamdata* construct is provided to encapsulate the behaviour and data associated with the target end of a team belief connection. Each teamdata is defined as a type level entity using the keyword `teamdata`, and it has the following form:

```
teamdata Type ... {  
    ... // declarations  
}
```

A teamdata is usually an extension of a normal JACK beliefset, which provides the declarations of fields and queries. The extension part defines the behaviours associated with the receipt of change propagations from the sources involved in the connection.

A teamdata definition includes two reasoning methods:

- a `#connection` method which defines the behaviour when teams are added to or removed from the connection, and
- a `#synthesis` method which defines the computation to be performed on receipt of a propagated belief. This method is invoked **regardless** of whether the connection is synthesizing or inheriting.

The `#synthesis` method is invoked to receive a propagated belief. It has the following prototype:

```
#synthesis method(  
    String team,  
    boolean wasAssert,  
    BeliefState truthValue,  
    XXXX__Tuple newTuple,  
    XXXX__Tuple keyDiscard,  
    XXXX__Tuple negateDiscard  
)
```

`XXXX__Tuple` is a placeholder for the type of the incoming tuple; it must be replaced with the actual type. The parameters are described in the following table:

Parameter	Description
team	The name of the sub-team whose change is propagated.
wasAssert	Whether the tuple concerned was asserted or retracted.
truthValue	The resulting <code>BeliefState</code> .
newTuple	The tuple concerned.
keyDiscard	The opposing tuple retracted by virtue of the key constraint, if any.
negateDiscard	The contradictory tuple being retracted, if it was believed.

Table 5-2: Parameters for the `#synthesis` method

The typical behaviour is for the synthesis method to add (selectively) the propagated tuple to the beliefset part of the teamdata. This beliefset can be of a different type to that of the propagated beliefset. However, if multiple sources are involved in a connection, those sources must all be of the same type.

The synthesis method should be written for optimal performance. If a change propagation update requires any lengthy computation, then the synthesis method should defer that computation and instead post an asynchronous event for that purpose.

The `#connection` method is a reasoning method which is invoked asynchronously when a sub-team is added to or removed from a role, and when this results in a change to whether or not there is a belief connection for that team. If the team belief is connected through multiple roles, then only the first addition or the last remove will result in a belief connection change and an associated `#connection` method invocation. It has the following prototype:

```
#connection method(boolean added, String team)
```

The parameters are described below:

Parameter	Description
added	Whether the sub-team has been added to or removed from the role.
team	The name of the sub-team that has been added to or removed from the role.

Table 5-3: Parameters for the `#connection` method

`#connection` method and `#synthesis` method invocations are synchronized, ensuring that a team performs only one such method at a time.

5.3 Belief Connection Dynamics

The computation flow in a team belief connection is as follows:

1. The starting point is that a beliefset which propagates change is updated. This defines the *belief detail* to be propagated.
2. The beliefset involved needs to include a `#propagates changes` statement in its definition. That statement results in a `moddb()` callback that posts a kernel event, named `ChangePropagation`, for propagating the change.
 - The kernel event is a `BDIGoalEvent`, so as to allow user code to override the whole propagation procedure at the source end.
 - The kernel plans for handling `ChangePropapgtion` have a rank of 4 or less, to allow a user's plan at the default plan rank, which is 5, to have precedence. If the user plan fails, the default handler will be invoked.
3. The change propagation runs as a parallel task for the source beliefset team. The default handler reviews the role relationship and determines the set of teams to which the belief detail should be propagated. It then spawns a new, parallel task for each target team to deal with per-target-team filtering and the actual inter-team transfer.
4. The `#propagates changes` statement may nominate an event type for the kernel to use so as to perform a per-target-team source end propagation filtering. When an event type is nominated, the default handler will `@subtask` that event as a means of deciding whether or not to propagate the change to a given team, and when this propagation is to occur. The former is decided by the event succeeding or failing, and the latter is decided by means of delaying success.
5. The per-team change propagation task next transfers the belief detail to the target team. This is achieved by sending a `PropagationMessage` event to the target team. The standard way of handling this event uses the `teamdata` elements as described in points 6 and 7 that follow. However, it is possible for a user to provide a team with special purpose plans to handle `PropagationMessage` events in other ways. Such special purpose plans should be of higher precedence rank than the standard plan which has a precedence rank of 5. The fields of the `PropagationMessage` event are described in the following table.

Parameter	Description
from	The name of the sub-team whose change is propagated.
source	The reference name of the source beliefset.
wasAssert	Whether the tuple concerned was asserted or retracted.
truthValue	The resulting <code>BeliefState</code> .
newTuple	The tuple concerned.
keyDiscard	The opposing tuple retracted by virtue of the key constraint, if any.
negateDiscard	The contradictory tuple being retracted, if it was believed.

Table 5-4: Fields of the `PropagationMessage` event

6. Upon receiving a change propagation notification, the target team first ensures change propagation sequencing by waiting on a semaphore. This semaphore is also used by connection method calls, to ensure that a team only performs one `#connection` method or `#synthesis` method at a time.
7. In turn, the target team propagation task completes the propagation by invoking the `synthesis` method of the teamdata involved, one at a time, followed by a signal to the sequencing semaphore.

All distribution filtering is done at the source end of a change propagation, and all synthesizing computation is done at the target end. Further, at the target end, the change propagations are sequential, allowing only one change propagation at a time to occur.

5.4 Synthesizing Belief Connection Definition

A synthesizing team belief connection maps sub-teams' beliefs into corresponding beliefs at the containing team level. This is achieved by propagating information from the sub-team beliefsets to the containing team(s). In order to create a synthesizing team belief connection, appropriate declarations must be included in the

- role that provides the sub-team/team linkage
- the sub-teams that are the source for the connection
- the team that is the target for the connection.

In addition

- a teamdata definition must be provided for the target team
- the source beliefsets must include `#propagate changes` statements.

5.4.1 Role Declarations

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

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

stype and *sref* identify a **source** beliefset that will be involved in a synthesizing belief connection – the target for the connection is **not** specified. Multiple declarations are allowed within a role definition.

Recall that a role defines a team/sub-team interface. Within a role type definition, the `#synthesizes teamdata` declaration 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 should have a target data declaration that involves this particular data item or it will be unable to receive the propagated beliefs.

5.4.2 Source 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 either directly within the sub-team definition, or indirectly through the sub-team's capability structure. The data item can be defined either through a `#private data` declaration, a `#exports data` declaration in a capability, a `#synthesizes teamdata` or through a `#inherits teamdata` declaration. The latter two cases require that the sub-team is the target for another belief connection.

5.4.3 Target 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.

A `#synthesizes teamdata` declaration has the following form in the team definition:

```
#synthesizes teamdata ttype tref(rcref1.sref1,rcref2.sref2,...);
```

where

`ttype` is the type of the target teamdata

`tref` is the name of the target teamdata reference

`rcrefi` is the name of the *i*th role container reference

`srefi` is the name of the *i*th source data item reference which is to be synthesized.

As indicated above, teamdata can be synthesized from beliefs specified in more than one role. In this case, multiple `#requires role` statements will be required in the team and the types of the source beliefs must be the same. Note that `ttype` refers to the type of the target teamdata, not the type of the source data.

Recall that the teamdata type is typically achieved by extending a beliefset type. Depending on the application, that beliefset type may be the same as the source data type or it may be different.

The above declaration results in the creation of a teamdata instance. The intention is that the data to be contained in this instance will be provided solely from the data sources for the connection – hence there is no mechanism to populate the instance at construction time. This teamdata instance is then accessible to the target team and through the `#uses data` declaration, to the target team's capabilities and plans as though it had been declared as `#private data`. In particular, a teamdata instance can be used as a source belief for another belief connection.

The `#synthesizes teamdata` statement results in code that ensures that when role fillers are added to or removed from any of the indicated role containers the corresponding beliefset change propagation path is added or removed. The actual synthesizing computation is defined separately (via the `#synthesis` method of the teamdata definition). Although a connection is defined in terms of role filling, it is maintained on a sub-team basis. Thus if a connection involves multiple roles and one sub-team fills more than one of the roles, a change to that sub-team's beliefset is propagated only once to the teamdata, and not once for each role container that contains the sub-team.

5.4.4 An Example

Suppose that a `Section` team requires a `Soldier` role and a `Private` team performs the `Soldier` role. Furthermore, suppose that the `Private` team maintains its current location and ammunition level in beliefsets of type `Location` and `Ammunition` respectively. A synthesizing team belief connection that will enable the `Section` team to monitor the location and ammunition levels of its members is to be established. This data will be stored in teamdata of type `SectionLocation` and `SectionAmmunition` respectively.

1. Source data definition

The `Location` and `Ammunition` beliefsets could be defined as follows:

```

beliefset Location extends OpenWorld {
    #value field double x;
    #value field double y;

    #linear query get(logical double x,logical double y);
    #propagates changes;
}

beliefset Ammunition extends OpenWorld {
    #key field String type;
    #value field int count;
    #indexed query get(logical String t,logical int c);
    #indexed query get(String t,logical int c);
    #propagates changes AmmoChangePropagation;
}

```

Note that the latter beliefset, `Ammunition`, propagates changes through the `AmmoChangePropagation` *filter event*. When a change occurs, the kernel will sub-task an instance of this event, for deciding whether and when the change propagation is to occur. The `AmmoChangePropagation` event will be similar to the following:

```

event AmmoChangePropagation extends BDIGoalEvent
    implements PropagationEvent {
    ...

    #posted as
    report( ... )
    {
        ...
    }

    public Event propagate(String team,
        Tuple newTuple,
        Tuple keyDiscard,
        Tuple negateDiscard,
        BeliefState truthValue,
        boolean wasAssert )
    {
        // just call the posting method ...
        return report( ... );
    }
}

```

The application can then include a plan to handle this `AmmoChangePropagation` event. The change propagation will then occur only if and when the plan succeeds.

Note: The `#propagates changes` declaration results in a `moddb()` method associated with the beliefset. This means that the programmer must not include their own `moddb()` method within the beliefset.

2. Target data definition

The `SectionAmmunition` beliefset is teamdata that accumulates the count of sub-team's `Ammunition` beliefsets. A possible definition for this is outlined below:

```
teamdata SectionAmmunition extends Ammunition {  
  
    #connection method(boolean added, String team)  
    {  
    }  
  
    #synthesis method  
    (String team,  
     boolean asserted,  
     BeliefState tv,  
     Ammunition__Tuple is,  
     Ammunition__Tuple was,  
     Ammunition__Tuple lost )  
    {  
        logical int current;  
        if (get(is.type,current)) {  
            // binds current  
        } else {  
            current.unify(0);  
        }  
        if (is != null) {  
            int delta = is.count;  
            if (lost != null)  
                delta -= lost.count;  
            add(is.type, current.getValue() + delta);  
        }  
    }  
}
```

`SectionLocation` makes use of explicit replication as follows:

```
teamdata SectionLocation extends Location {  
  
    Hashtable locations = new Hashtable();  
  
    Location location(String team)  
    {  
        return (Location) locations.get(team);  
    }  
  
    #connection method(boolean added, String team)  
    {  
        if (added) {  
            if (locations.get(team) == null ) {  
                Location location = new Location();  
                location.attach(handler);  
                locations.put(team, location);  
            }  
        } else {  
            locations.remove(team);  
        }  
    }  
}
```

```

#synthesis method
    (String team,
     boolean asserted,
     BeliefState tv,
     Location__Tuple is,
     Location__Tuple was,
     Location__Tuple lost)
{
    Location location = (Location) locations.get(team);
    if (asserted)
        location.add(is, tv);
    else
        location.remove(is, tv);

    double sum_x = 0;
    double sum_y = 0;
    int n = locations.size();

    for (Enumeration e = locations.elements();
         e.hasMoreElements(); ) {
        Location location = (Location) e.nextElement();
        logical double x;
        logical double y;
        location.get(x,y);
        sum_x += x.getValue();
        sum_y += y.getValue();
    }
    sum_x /= n ;
    sum_y /= n ;
    add(sum_x,sum_y);
}
}

```

Note: The code in the above example also invokes the `attach()` method when a belief replication beliefset is created, providing the local `handler` as the argument. This statement is a JACK detail that is hidden in the generated code for beliefsets, but which must be dealt with explicitly for belief replication. The purpose is to attach the new beliefset object to the correct `EventRecipient` (i.e. the entity that is to handle any event being posted by the beliefset), which in practice is the enclosing team.

In the example, the local `handler` is inherited from the ultimate base class, `BeliefSet`, via the explicit base class, `Location`. The local `handler` is thus available since the synthesized belief extends a beliefset. In the general case, the synthesized belief may need to capture the `EventRecipient` explicitly by implementing the `EventSource` interface, which in fact is the `attach()` method.

3. Role declarations

The `Soldier` role could contain the following declarations:

```

role Soldier extends Role {
    #synthesizes teamdata Location location;
    #synthesizes teamdata Ammunition ammo;
    ...
}

```

4. Source declarations

The `Private` team needs to perform the `Soldier` role and to define the data sources specified within that role:

```
team Private extends Team {
    #performs role Soldier;

    #private data Location location();
    #private data Ammunition ammo();
    ...
}
```

5. Target declarations

The `Section` team could incorporate declarations similar to the following:

```
team Section extends Team {
    #requires role Soldier left(3,3);
    #requires role Soldier right(3,3);
    #requires role Soldier depth(3,3);

    #synthesizes teamdata SectionLocation location
        (left.location, right.location, depth.location);
    #synthesizes teamdata SectionAmmunition ammo
        (left.ammo, right.ammo, depth.ammo);
    ...
}
```

5.5 Inheriting Belief Connection Definition

An inheriting team belief connection maps a team belief into separate sub-team beliefs. Conceptually this is done by means of a distribution computation that translates the team belief individually for each sub-team, followed by a (virtual) replication of the translated belief into the corresponding sub-team's belief. Often a sub-team will perform a role for one team only, but in the general case the sub-team may fill the same role for many teams and the inherited belief connection will combine belief updates from all the teams in the same way as a synthesizing belief connection.

In order to create an inheriting team belief connection, appropriate declarations must be included in the

- role that provides the team/sub-team linkage
- the teams that are the source for the connection
- the sub-teams that are the target for the connection.

In addition

- a `teamdata` definition must be provided for the target sub-teams
- the source beliefsets must include `#propagates changes statements`.

5.5.1 Role Declarations

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

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

`stype` and `sref` identify a **source** beliefset that will be involved in an inheriting belief connection – the target for the connection is **not** specified. Multiple declarations are allowed within a role definition.

Recall that a role defines a team/sub-team interface. Within a role type definition, a `#inherits teamdata` declaration 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 should have a target data declaration that involves this particular data item or it will be unable to receive the propagated beliefs.

5.5.2 Source Declarations

A team becomes a source in an inheriting belief connection by requiring a role that contains a `#inherits teamdata` declaration. The team must therefore include an appropriate `#requires role` declaration. Also, a data item with the type and the reference specified within the role must be defined either directly within the team definition, or indirectly through the team's capability structure. The data item can be defined either through a `#private data` declaration, a `#exports data` declaration in a capability, a `#synthesizes teamdata` declaration or through a `#inherits teamdata` declaration. The latter two cases require that the team is the target for another belief connection.

5.5.3 Target 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. Note that this binding differs to that in a synthesizing belief connection, as a role performer does not have access to the role container.

A `#inherits teamdata` declaration has the following form in the sub-team definition:

```
#inherits teamdata ttype tref (rtype1.sref1, rtype2.sref2, ... );
```

where

`ttype` is the type of the target teamdata,

`tref` is the name of the target teamdata reference,

`rtypei` is the type of the performed role, and

`srefi` is the name source data item in the performed role that is to be inherited.

As indicated above, teamdata can be inherited from beliefs contained in more than one role. In this case, multiple `#performs role` statements will be required in the sub-team and the types of the source beliefs must be the same. Note that `ttype` refers to the type of the target teamdata, not the type of the source data.

Recall that the teamdata type is typically achieved by extending a beliefset type; depending on the application, that beliefset type may be the same as the source data type or it may be different.

The above declaration results in the creation of a teamdata instance. The intention is that the data to be contained in this instance will be provided solely from the data sources for the connection – hence there is no mechanism to populate the instance at construction time. This teamdata instance is then accessible to the sub-team, and its capabilities and plans as though it had been declared as `#private data` – in particular, it can be used as the source belief of another belief connection.

The `#inherits teamdata` statement results in code that ensures that when role fillers are added to or removed from any of the indicated role containers the corresponding beliefset change propagation path is added or removed. The actual synthesizing computation is defined separately (via the `#synthesis` method of the teamdata definition). Although a connection is defined in terms of role filling, it is maintained on a sub-team basis. Thus, if a connection involves multiple roles and one sub-team fills more than one of the roles, a change to the containing team's beliefset is propagated only once to the sub-team's teamdata, and not once for each role container that contains the sub-team.

5.5.4 An Example

Suppose that a `Company` team requires a `FireSupport` role and a `Platoon` team can perform the `FireSupport` role. Furthermore, suppose that the `Company` team maintains the current enemy location in a beliefset of type `Location`. An inheriting team belief connection that will enable the `Platoon` team to monitor the enemy location is to be established. This data will be stored in teamdata of type `EnemyLocation`.

1. Source data definition

The `Location` beliefset definition developed for the synthesizing belief connection example can be used:

```
beliefset Location extends OpenWorld {
    #value field double x;
    #value field double y;

    #linear query get(logical double x,logical double y);
    #propagates changes;
}
```

2. Target data definition

The `EnemyLocation` beliefset is teamdata that mirrors the enemy location maintained by the Company team. A possible definition for this is outlined below:

```
teamdata EnemyLocation extends Location {

    #connection method(boolean added, String team)
    {

    }

    #synthesis method
    (String team,
     boolean asserted,
     BeliefState tv,
     Location__Tuple is,
     Location__Tuple was,
     Location__Tuple lost )
    {
        if (is != null) {
            add(is.x, is.y);
        }
    }
}
```

3. Role declarations

The `FireSupport` role could contain the following declaration:

```
role FireSupport extends Role {
    #inherits teamdata Location enemyLocation;
    ...
}
```

4. Source declarations

The `Company` team requires a team to perform the `FireSupport` role and it must define the data sources specified within that role:

```
team Company extends Team {
    #requires role FireSupport fireSupport;
    ...

    #private data Location enemyLocation();
    ...
}
```

5. Target declarations

The `Platoon` team needs to perform the `FireSupport` role and to create the `teamdata` instance to receive the enemy location:

```
team Platoon extends Team {
    #performs role FireSupport;
    ...

    #inherits teamdata EnemyLocation reportedEnemyLocation
        (FireSupport.enemyLocation);
    ...
}
```

6 Team Formation

This chapter describes team formation. It includes a description of the process that occurs when sub-teams are attached to the role obligation structure of a containing team. During the process of attachment/detachment, the associated role instances pass through various states. This process of attachment is carried out during the initialisation phase when the team is constructed. In addition, it is possible to have sub-teams dynamically attached/detached to the role obligation structure while the application is running.

Teams have a capability, `TeamCap`. This capability includes plans that handle the initial team formation with the sub-tasking and default handling of a `TeamFormationEvent`, and the posting of a `StartTeamEvent`. The initialisation process is triggered automatically as part of the team instance construction.

An application may override the handling of a `TeamFormationEvent` simply by defining a plan that handles the event (the default plan is of precedence rank 0, and therefore a new plan – with standard rank 5 – will have higher precedence.)

The `TeamCap` capability also includes handling of `RoleControl` events. These events are posted by the application level to assign or revoke a sub-team as a role performer. When they are posted, the infrastructure layer posts additional events that result in any negotiation and detachment/attachment required to connect a sub-team to a containing team. The application layer may include plans that are involved in the negotiation and/or react to the success or failure of attachment/detachment. The `RoleControl` events therefore allows dynamic modification of role obligation structures during the lifetime of an application.

If the application layer requires a *non-JACK* agent/team (e.g. an agent written in an alternative language) to perform a role, the application code would need to carry out the attachment and negotiation on both sides of the relationship without the use of `RoleControl` events.

6.1 RoleType Instance State Management

A `RoleType` instance can be in one of three states (`INACTIVE`, `ACTIVE` OR `DETACHED`). When the `RoleType` object is created, it is initially in the `ACTIVE` state. However, it will not be available/visible until it is actually added to a role container. The posting of a `RoleControl` event constructed using the `assign` posting method results in an attempt to add the `RoleType` object to the specified role container. If the object is successfully added to the role container, then the object will be in the `ACTIVE` state.

When the team revokes a role, the `RoleType` object is placed in the `DETACHED` state. This means that any plans that already have access to the object can continue to access the object but no new activity should be allowed access the object. It is the users responsibility to check the state of the `RoleType` object.

Team Formation

The `RoleType` object will change from the `DETACHED` to the `INACTIVE` state when the team filling the role finishes processing all tasks associated with the role. It will also be removed from the role container at that point.

The interaction diagram in the following figure illustrates the interaction between a tendering team and a suggested role performer for processing an `ASSIGN` action.

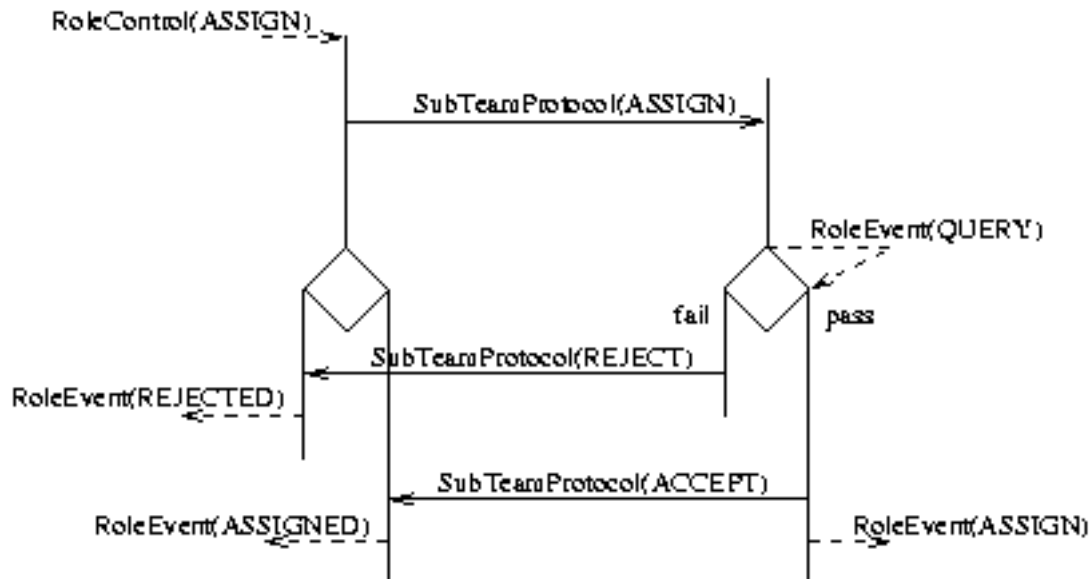


Figure 6-1: Interactions for adding a role performer

On the tendering team side, the `ASSIGN` action is initiated via a `RoleControl(ASSIGN)` event, that asks for the addition of a role performer to a given `RoleContainer`. This is a synchronous request, and the `RoleControl` handler does not return control to the invoking task after having issued the `SubTeamProtocol(ASSIGN)` message, but waits until the actor team has processed and replied to it.

On the (suggested) performer side, the infrastructure layer queries the application level whether the suggested addition is to be accepted. This is done by sub-tasking a `RoleEvent(QUERY)` event, to be handled by the application level and passed or failed. If the event handling fails, the suggested action is understood as rejected, and a `SubTeamProtocol(REJECT)` message is sent to the tendering team. This results in a transition for the `RoleType` instance to `Void` state (i.e. the `RoleType` instance is marked as `Detached` and it is removed from the `RoleTypeContainer`). Further, a `RoleEvent(REJECTED)` is posted on the tenderer team side, so as to allow application level reasoning about the rejected addition.

If the handling of the `RoleEvent(QUERY)` event succeeds, the suggested action is understood as accepted, and a `SubTeamProtocol(ACCEPT)` message is sent to the tendering team. This results in a `RoleType` instance state transition to `Active`, and the posting of a `RoleEvent(ASSIGNED)` event, so as to allow application level reasoning about the accepted addition. On the performer side, a `RoleEvent(ASSIGN)` event is posted to notify about a successful assignment.

The following figure illustrates the interaction between a tendering team and a suggested role performer for processing a `REVOKE` action.

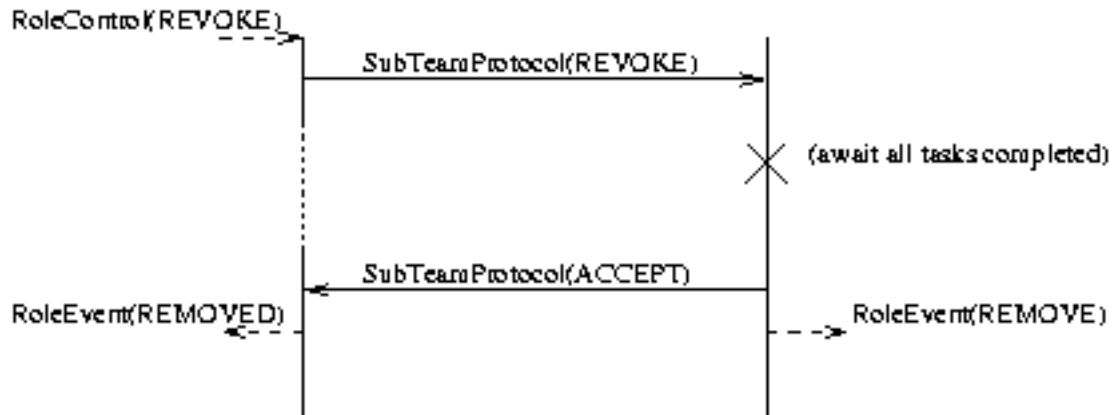


Figure 6-2: Interactions for removing a role performer }

On the tendering team side, the `REVOKE` action is initiated via a `RoleControl(REVOKE)` event, that asks for the removal of a role performer from a given `RoleContainer`. This is a synchronous request, and the `RoleControl` handler does not return control to the invoking task directly after having issued the `SubTeamProtocol(REVOKE)` message, but waits until the actor team has processed and replied to it. First, however, the `RoleType` instance concerned is marked as `Detached`, to block subsequent attempts to issue further team tasks to the role performer.

On the performer side, the infrastructure then waits until all tasks under the role have been completed before returning a `SubTeamProtocol(CONFIRM)` message to the role tendering team. As application level notifications, `RoleEvent(REMOVE)` and `RoleEvent(REMOVED)` events are posted on the performer and tendering sides respectively.

6.2 Role Handling Events and Messages

`SubTeamProtocol` [sent and handled event]

The `SubTeamProtocol` event is used for the role handling protocol between teams. This is an infrastructure message event that is **not** used at application level.

RoleControl [handled event]

The `RoleControl` event is posted by the application level for assigning or revoking a sub-team to be a role performer. The same event type is used for both `ASSIGN` and `REVOKE` actions, and it is defined with a range of posting methods.

```
event RoleControl extends Event {
  #posted as
  assign(String role, String container, String actor)
  // role is the role type
  // container is the reference to the role container
  // actor is the team name (to be assigned)

  #posted as
  revoke(String role, String container, String actor)
  // role is the role type
  // container is the reference to the role container
  // actor is the team name (to be revoked)
}
```

`RoleControl(ASSIGN)`

The `ASSIGN` action event is constructed through the `assign` posting method. This is handled by the infrastructure, resulting in the team interaction to establish the given actor as role filler. For example:

```
import martian.Crew;

teampplan RescueMartian extends TeamPlan {
  #handles event Rescue re;
  #posts event RoleControl rc;
  #uses interface Team team;

  body()
  {
    // post a RoleControl event to add a sub-team
    // to the role obligation structure
    if(@subtask(rc.assign("martian.Crew", "cr", re.name)))
      System.out.println("rescued "+re.name);
    else
      System.out.println("could not rescue "+re.name);
  }
}
```

`RoleControl(REVOKE)`

The `REVOKE` action event is constructed through the `revoke` posting method. This is handled by the infrastructure, resulting in the team interaction to revoke the given actor as role filler.

RoleEvent [posted event]

The infrastructure posts RoleEvent events to connect with the application level for notifications and application level reasoning. The event is a polymorphic event that is posted in different modes for different purposes.

```

event RoleEvent extends BDIGoalEvent {
    #set behavior Recover never;

    public String team;
    public String container;
    public String role;

    public int mode;
    public final static int QUERY = -1;
    public final static int REJECT = 0;
    public final static int ASSIGN = 1;
    public final static int REJECTED = 2;
    public final static int ASSIGNED = 3;
    public final static int REMOVE = 5;
    public final static int REMOVED = 7;
}

```

RoleEvent(QUERY)

The QUERY mode is *sub-tasked* by the infrastructure while handling a role performer ASSIGN action, and the purpose is for the application level to decide whether or not the proposed addition should be accepted. If the event handling succeeds, the addition is accepted, and if it fails, the addition is rejected.

RoleEvent(REJECT)

The REJECT mode is *posted* by the infrastructure for the role performer as a notification that the addition has been rejected.

RoleEvent(ASSIGN)

The ASSIGN mode is *posted* by the infrastructure for the role performer as a notification that the addition has been accepted.

RoleEvent(ASSIGNED)

The ASSIGNED mode is *posted* by the infrastructure for the role tenderer as a notification that the addition has been accepted.

RoleEvent(REJECTED)

The REJECTED mode is *posted* by the infrastructure for the role tenderer as a notification that the addition has been rejected.

RoleEvent(REMOVE)

The REMOVE mode is *posted* by the infrastructure for the role performer as a notification that the team has been removed as performer of a role.

Team Formation

RoleEvent (REMOVED)

The REMOVED mode is *posted* by the infrastructure for the role tenderer as a notification that a team has been removed as performer of a role.

Index

Symbols

#applicable_for role 49
#applicable_from role 49
#connection method 57, 58, 60
#container member 40
#container method 40
#container method() 50
#handles event 39
#inherits teamdata 29, 40, 67
#performs role 17, 18, 28
#posts event 39
#propagates changes 55
#reasoning method establish 50
#requires role 12, 17, 28, 47, 50
#synthesis method 57, 60
#synthesizes teamdata 29, 39, 61
#uses role 24, 48, 50
@parallel 11, 25, 45, 50
@teamAchieve 11, 25, 39, 45, 50, 51
 exception propagation 53
 returning values 53

A

Active 73
active 42
actor 41
assign posting method 32
attach method 65

B

BDI 9
behaviour 14
belief 9
belief connection dynamics 55, 59
belief exchange 10
belief propagation 10

C

canPerformRole() 34
compile example 26
contained team 11

containing team 11
create sub-team 18

D

-D flag 26
defaultEstablish() 50
desire 9
Detached 73
dynamic team formation 32

E

establish 14
establish reasoning method 11, 14, 24, 45,
 50
establish() 50
EventRecipient 65

F

fail reasoning method 14
find() 43
findContainer() 34
findPerformedRole() 34
findRequiredRole() 34

G

getRoles() 34
goal exchange 10
group behaviour 14

H

handler 65
hierarchy 10

I

inheriting team belief connection 66
inherits teamdata 68
initialisation file 13, 18, 19, 26
 example 13
intention 9

J

JACK Intelligent Agents 9
JACOB 9, 13

M

max 43
min 43
mirror 41
moddb() 63
multi-level hierarchy 10

N

name 42
nextFiller 50
nextFiller() 43
nextTag() 43
noTasks 42

P

peer 41
plan body 25
plan failure 14
postWhenFormed() 34
propagate sub-team belief 15
propagate team belief 15
propagated belief types 56
 in capabilities 56
propagates changes 59
propagation 10
PropagationEvent 56
PropagationMessage 59

R

revoke posting method 32
Role 37
role 10, 11, 20, 37, 42
 declaration 11
 declarations 38
 #container member 40
 #container method 40
 #handles event 39
 #inherits teamdata 40
 #posts event 39

 #synthesizes teamdata 39

 definition 12, 37

Role Base Class 40

 actor 41
 mirror 41
 noTasks() 42
 peer 41
 setState() 42
 state 41
 tag 41
 tasks 42

role container 10, 12

role filler 11

role object 25

role obligation structure 13, 14, 18, 30, 71

role performer 11, 13

role relationship 10

role tenderer 11

Role.ACTIVE 24

RoleContainer 24

RoleContainer Base Class 42

 active 42
 find() 43
 max 43
 min 43
 name 42
 nextFiller() 43
 nextTag() 43
 role 42
 rolesInitialized() 43
 size() 43
 tags() 43
 team 42

RoleControl 74

RoleControl event 32, 71

 assign posting method 32

 revoke posting method 32

RoleControl(ASSIGN) 72

RoleControl(REVOKE) 73

RoleEvent 75

RoleEvent(ASSIGN) 72

RoleEvent(ASSIGNED) 72

RoleEvent(QUERY) 72

RoleEvent(REJECTED) 72

RoleEvent(REMOVE) 73
 RoleEvent(REMOVED) 73
 rolesInitialized() 34, 43
 RoleTypeContainer 40
 run example 26

S

setState() 42
 size() 43
 StartTeamEvent 33, 71
 state 41
 sub-team 10, 11, 18
 SubTeamProtocol 73
 SubTeamProtocol(CONFIRM) 73
 SunTeamProtocol(ACCEPT) 72, 73
 SunTeamProtocol(ASSIGN) 72
 SunTeamProtocol(REJECT) 72
 SunTeamProtocol(REVOKE) 73
 synthesizes teamdata 62
 synthesizing team belief connection 60

T

tag 41
 tags() 43
 task team 14, 25
 task team establishment 49
 task team formation 11, 45
 task teams 14
 tasks 42
 team 10, 27, 42

- construction 30
- declarations 28
 - #inherits teamdata 29
 - #performs role 28
 - #requires role 28
 - #synthesizes teamdata 29
- definition 12, 17, 27
- formation 31
- initialisation file 13, 31
- management 30
- manager 30

Team Base Class 34

- canPerformRole() 34
- findContainer() 34
- findPerformedRole() 34
- findRequiredRole() 34
- getRoles() 34
- postWhenFormed() 34
- rolesInitialized() 34
- Team() 35

team belief connections 55
 team extension 10
 team formation 10, 13, 71
 team formation constraints 12
 team goal handling 50
 team hierarchy 10
 team structure 10, 26
 team type definition 11
 Team() 35
 Team.Structure property 26, 31
 TeamAbort 53
 TeamCap 71
 teamdata 10, 57

- declarations 57
 - #connection method 57
 - #synthesis method 57
- definition 57

TeamError 53
 TeamException 53
 TeamFormationEvent 13, 30, 32, 71
 team-oriented 9
 Teamplan 9
 teamplan 11, 14, 45

- @-statements 51
 - @teamAchieve 51
- declarations 47
 - #applicable_for role 49
 - #applicable_from role 49
 - #requires role 47
 - #uses role 48
- definition 45
- members 50
- methods 50

team-role declaration 12
 team-role structure 11
 Teams 9
 Teams framework 13
 teams reasoning 9

TeamStartEvent 30
termination condition 25