A BDI-BASED MULTIAGENT SIMULATION FRAMEWORK

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MURAT YÜKSELEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2008

Approval of the thesis

A BDI-BASED MULTIAGENT SIMULATION FRAMEWORK

submitted by **MURAT YÜKSELEN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Volkan Atalay
Head of Department, **Computer Engineering**

Prof. Dr. Faruk Polat
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. İ. Hakkı Toroslu
Computer Engineering Department, METU

Prof. Dr. Faruk Polat
Computer Engineering Department, METU

Prof. Dr. Göktürk Üçoluk
Computer Engineering Department, METU

Assoc. Prof. Dr. Ahmet Coşar
Computer Engineering Department, METU

Mustafa Kemal Kaplan
Central Bank of the Republic of Turkey

Date: 05.09.2008

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Murat Yükselen

Signature :

ABSTRACT

A BDI-BASED MULTIAGENT SIMULATION FRAMEWORK

Yükselen, Murat

M.S., Department of Computer Engineering

Supervisor: Prof. Dr. Faruk Polat

September 2008, 70 pages

Modeling and simulation of military operations are becoming popular with the widespread application of artificial intelligence methods. As the decision makers would like to analyze the results of the simulations in greater details, entity-level simulation of physical world and activities of actors (soldiers, tanks, etc) is unavoidable. In this thesis, a multiagent framework for simulating task driven autonomous activities of actors or group of actors is proposed. The framework is based on BDI-architecture where an agent is composed of beliefs, goals and plans. Besides, an agent team is organized hierarchically and decisions at different levels of the hierarchy are governed by virtual command agents with their own beliefs, goals and plans. The framework supports an interpreter that realizes execution of single or multiagent plans coherently. The framework is implemented and a case study demonstrating the capabilities of the framework is carried out.

Keywords: Multiagent simulation, multiagent systems, behaviour modeling, agent-based modeling and simulation, semi-automated forces

ÖZ

BDI TABANLI ÇOKLU ETMEN SİMÜLASYON ÇATISI

Yükselen, Murat

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Faruk Polat

Eylül 2008, 70 sayfa

Askeri operasyonların modelleme ve simülasyonu yapay zeka yöntemlerinin daha yaygın uygulanması ile popülerleşmektedir. Kararverici insanların simülasyon sonuçlarını daha detaylı analiz etme isteği ile fiziksel dünyanın ve aktörlerin (asker, tank vb.) aktivitelerinin varlık seviyesi simülasyonları kaçınılmaz hale gelmiştir. Bu tezde bir aktörün veya bir grup aktörün görev tabanlı otonom davranışlarını simüle etmek için çok etmenli bir yazılım çatısı önerilmektedir. Çatı bir etmenin inanç, amaç ve planlardan oluştuğu BDI mimarisine dayanmaktadır. Bunun yanı sıra, bir etmen takımı hiyerarşik olarak organize edilir ve hiyerarşinin farklı seviyelerindeki kararları sanal etmenler kendi inanç, amaç ve planları ile yönetirler. Yazılım çatısı tek ve çoklu etmen planlarının uyumlu olarak gerçekleşmesini sağlayan bir yorumlayıcı sağlamaktadır. Önerilen yazılım çatısı gerçekleştirilmiş ve yazılımın kabiliyetlerini göstermek için örnek bir çalışma hazırlanmıştır.

Anahtar Kelimeler: Çoklu etmen simülasyonu, çoklu etmen sistemleri, davranış modelleme, etmen tabanlı modelleme ve simülasyon, yarı otonom kuvvetler

To my family and friends

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Dr. Faruk Polat and all members of Computer Engineering Department.

I would also like to thank to Tolga Can, İ. Hakki Toroslu, Göktürk Üçoluk, Burçin Sapaz, Utku Erdoğdu, Ali Galip Bayrak, Bahar Pamuk and Mehmet Gülek for their support, critics and ideas in the project of METU-TAF MODSIMMER.

This study is financially supported by TÜBİTAK.

Finally, I want to thank to my family and dear friends for their endless support during my study.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Knowledge is naively acquired through trial-error sessions. Using simulation in every aspects of these trial sessions is a cost effective way. Software simulation is applied to numerous fields that can be modelled mathematically and pushes computation limits in order to capture the most possible level of detail. As computation techniques advances, it enables simulation developers new openings to explore. Currently simulation is running on computer hardware but new openings in computation mediums such as quantum computation or biological computing are still a possibility.

Simulation of rational reasoning entities is the field of multiagent systems (MAS). Multiagent systems can be defined as a system having several independent intelligent agents interacting together to accomplish their goals. MAS research include diverse topics of interest such as cooperation, coordination, communication, negotiation, social interactions. MAS applications range from training systems to online computer games.

Belief Desire Intention (BDI) model captures the mental attitudes of an agent in three distinct representations. Belief represents the knowledge of the agents. Desire represents motivations of the agent. Intentions of an agent represents the active tasks pursued by the agent. Architectures based on the BDI model represent beliefs, desires and intentions explicitly as data structures and defines the operation of the agent through an agent interpreter. BDI architecture is an abstract architecture that enables realization of autonomous agents for multiagent systems. There exist many agent frameworks realizing BDI architecture. These frameworks provide mature agent-oriented software development process for multiagent systems.

In this thesis, a BDI-based multiagent simulation framework is proposed. The framework is

mainly targeted for entity level simulations where each entity is controlled by an independent agent. A sample application area of military simulation is chosen to explore its top down command control and hierarchical nature. The framework has facilities to address possible problems that are common while developing a MAS simulation. These facilities are defined from an abstract view and related to the presented framework. An example case study is engineered to illustrate to showcase how facilities can help to solve common simulation application problems. The realization of the framework is carried out by extending a solid BDI agent interpreter called Jadex[20]. The case study not only demonstrates the framework but gives reader how to grasp the simulation problem and fit it in the proposed solution.

In this work, overview of the BDI architectures and frameworks are discussed first. Chapter 3 presents the proposed abstract simulation framework and give details about agent architecture behaviors. Chapter 4 and chapter 5 delivers the details of the implementation of the proposed simulation framework and explains the usage with a case study.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 BDI Agent Model

Belief Desire Intention (BDI) model tries to capture human practical reasoning in order to formalize developing rational agents [4]. After its introduction, BDI model is refined [23, 21] to be used in real agent based systems.

BDI model represents mental attitudes of an agent in three categories : Beliefs, desires and intentions. Beliefs corresponds to knowledge of the agent about itself and outside world. Desires are also named as goals and they define the objectives of the agent trying to reach. At a time instance, agent can not pursue all of its desires. Because of this fact, intentions of the agent describe the current behaviours in action targeted to fulfill its selected desires. Behavior of an agent is described through plans and they can be seen as pre-compiled agent actions. Intentions of the agent can be seen as the running plan instances.

Figure 2.1 describes a BDI agent architecture. Circle denotes the agent and agent is connected to the outside world by sensor input and action output. Core of the BDI architecture is the agent interpreter. This event based interpreter consumes sensory information and executes the behaviours defined in the plan library. Intentions hold the plans in execution. Plan execution can trigger belief and desire changes in the agent which are also internal events that will be processed by the interpreter. Action output of the agent is controlled by the running plan instances, which are intentions.

BDI is a widely accepted and matured model in describing agents and their behavior. There are numerous implementations of BDI formalism and surveys[16] about their properties.

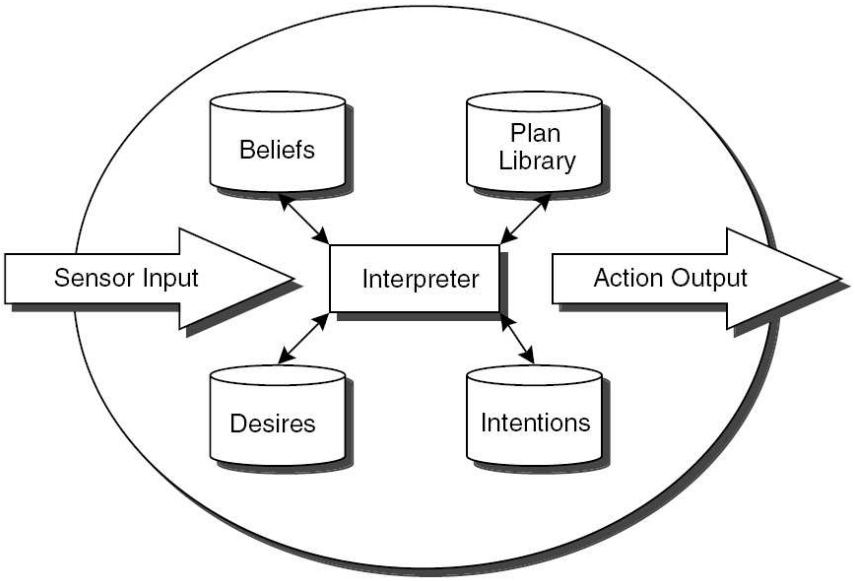

Figure 2.1: A BDI Agent Architecture

2.2 BDI-Based Agent Oriented Programming Languages and Platforms

The Procedural Reasoning System (PRS) [6] is one of the first implementation in lisp language based on BDI architecture developed by SRI International. The system is developed as a representation of an expert's procedural reasoning. It is used for evaluating maintainance procedures for the space shuttle in a simulation.

After the success of PRS, it is rewritten in C++ known as distributed Multi-Agent Reasoning System (dMARS) [5] at Australian AI Institute. The implementation of the platform also includes graphical editors, compiler and interpreter for a goal-oriented logical language. dMARS is used in industrial applications such as Oasis air traffic management system handling over 100 aircraft arrival to an airport and Swarmm [17] agent-based simulation system to simulate air mission dynamics and pilot reasoning. A summary of dMARS applications can be found in [7].

AgentSpeak(L) [22] starts by evolving from PRS and dMARS and formalizes its operation. It is based on a restricted first-order language with events and actions. Jason [3] is one of the implementations of AgentSpeak(L) that extends with speech-acts [25].

3APL [9] architecture has many similarities with other architectures such as PRS. Different from other architectures, 3APL is designed to control and revise todo goals of agent. 3APL also incorporates practical reasoning rules to revise mental attitudes. A 3APL agent is defined with a set of actions and a set of rules.

Dribble [24] is a propositional language that constitutes a synthesis between the declarative features of the language GOAL [10], and the procedural features of 3APL.

Coo-BDI (Cooperative BDI) [1] is based on the dMARS specification and extends it by introducing cooperations among agents to retrieve external plans for achieving desires. The cooperation strategy is defined by a set of agents to cooperate, plan retrieval policy and plan acquisition policy. The mechanism for retrieving relevant external plans involves cooperation with trusted agents.

JAM is an intelligent agent architecture that grew out of academic research and extended during the last five years of use, development, and application. JAM combines ideas drawn from the BDI theories, the PRS system and its UMPRS and PRS-CL implementations, the SRI International's ACT plan interlingua [18], and the Structured Circuit Semantics (SCS) representation [13]. It also addresses mobility aspects from Agent Tcl [8], Agents for Remote Action (ARA) [19], Aglets [12] and others.

Jack is a commercial and mature Java implementation of BDI architecture [14, 11]. Jack introduces agent oriented programming concepts on top of object oriented Java language and supplies a runtime to support this agent oriented extensions. Jack agents are defined in a Java like language. The language files are compiled to various intermediate Java files in the process and a runtime library helps agent interpretation. Agent definitions can use notion of capability that allows modularity by encapsulation and promotes code/design reuse. Jack treats goals as a special kind of event in its event based interpretation.

Jadex [20] is an open source BDI architecture implementation in Java. Jadex emphasizes use of goal as a first class data structure unlike other BDI implementation where a goal is treated as a special event type whose handling results in plan activation. Jadex is a BDI interpreter that is not bound to underlying agent software middleware. Currently Jadex supports its standalone middleware, Jade[2] and Diet-agents[15] platform. Jadex is also flexible in terms of runtime adaptability which allows an agent to add any belief, goal and plan definition in

runtime.

JACK has an extension that provides dynamic team formation and team based agent programming to support team-oriented modelling. A JACK team also has all the properties of an agent. It uses the notion of roles in which a team can require specific roles to accomplish its assignment. Team are then dynamically formed by fulfilling role containers and execution starts after the team formation. JACK Teams also provides teamdata, a way to communicate belief between agents either from bottom up or top down. Bottom up aproach is used for information fusion for higher level of abstraction in teams. Team plans are slightly extended to support these extensions.

In this thesis, Jadex is chosen as an extension point because it is open source, has explicit goal processing and solid codebase.

2.3 Jadex

Jadex is an agent-oriented reasoning engine supporting different agent middlewares. An agent middleware is a software platform for agents and deals with agent management and communication services. From the perspective of the agent middleware, a jadex agent is a black box that can only receive and send messages. Jadex provides a reasoning engine for agent implementation and it is based on BDI-model.

Jadex enables writing rational agents with XML files and Java language as illustrated in Figure 2.2. An agent is defined with an XML file called Agent Definition file (ADF). ADF mainly encapsulates definitions of beliefs, goals, plans and events. Furthermore, Jadex supports to engineer capabilities to group related belief, goal, plan and event structures. These defined capabilities can be extended and used by other capabilities and agents. Notion of capabilities introduces information hiding and encapsulation in agent-oriented modelling.

Jadex uses Java language mainly in plan body definition, data structure definition to be used as beliefs and expressions that can be evaluated inline for conditions. A plan body is a Java class that extends Jadex abstract plan and holds procedural information how a plan should work.

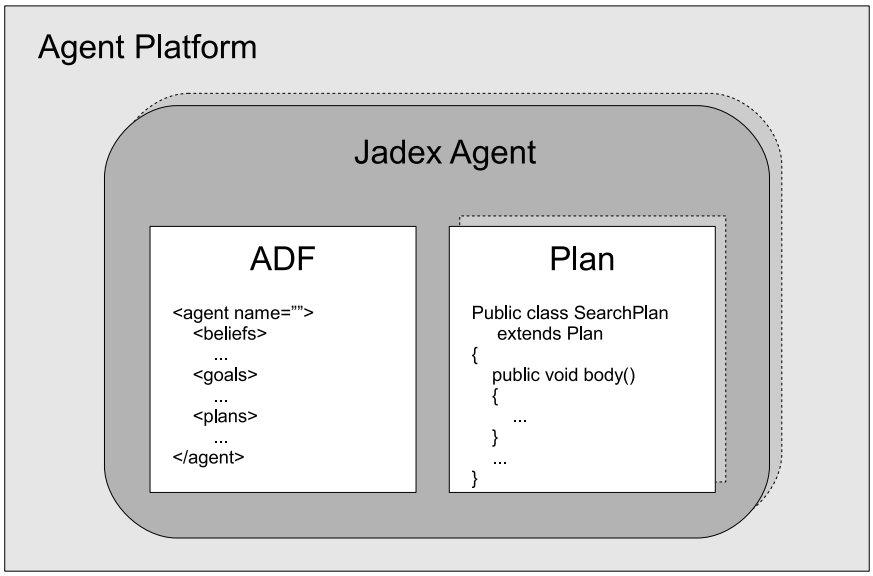

Figure 2.2: Composition of a Jadex Agent

2.4 Motivation

Motivation behind the design of a BDI-based multiagent simulation framework is to deliver a generic framework for team oriented agent programming. The proposed architecture incorporates facilities that eases agent coordination, execution control, synchronization and information exchange. These facilities are domain independent and form foundations for team behavior implementation. Coordination, synchronization and information exchange are fundamental problems that can be encountered in any multiagent implementation involving cooperation between agents.

CHAPTER 3

A BDI-BASED MULTIAGENT SIMULATION FRAMEWORK

Military simulation is a valuable tool used in analysis, human training and asset acquisitions. In this work, a framework to simulate agent behaviours for semi-automated military task force is presented. Main motivation of the framework is to enable analyzing various factors of the simulated environment. This fact presents several requirements such as:

Closed simulation: It should be possible to run the simulation without human interaction. This requirement removes the possible human performance effect from the simulation run. This feature enables to analyze the different factors present between the simulation runs in a controllable way. For example user can change the range of sight of a sensor and rerun the simulation to analyze the effects.

Autonomous simulation run: Simulation will conduct the task flows present in the scenario. Each agent team will try to accomplish the task defined in an autonomous way. Definition of task flows is the users responsibility.

A simulation application using the framework can be implemented with the following ingredients:

- Environmental dynamics and properties that define physical simulation.
- Task implementations that models agent behaviours.
- Scenario definition that command and control task forces.

Analysis of the simulation can be conducted on different factors. Being able to run the simulation closed to human interaction gives the user the opportunity to experiment with the factors present in a systematic and controlled manner.

Physical simulation and environment: Physical environment can affect how the agents observe and act in the simulation. In military domain,

- terrain,
- whether conditions,
- weapon systems,
- ammunition,
- platform mobility,
- sensors,
- damage etc.

can be tested. Environmental dynamics can have different effects on the performance of task units.

Task behaviour: A task defines a clear objective. Achievement of this objective can be interpreted and conducted differently. A task implementation is done programmatically to conduct a chosen behaviour and tries to capture intra-team coordination. In order to test different behaviours, user is free to implement different tasks or change the current ones prior to simulation.

Task flows: Task flows defines how a side behaves in possible conditions at the highest level. Definition of the task flows is a tedious process because of its exponential nature. User has to reflect all possible decisions in task flow definition. Scenario can be seen as a medium to define strategic decisions and inter team coordination. For example, to test different tactics, user can run the simulation with different task flows. Also it is possible to define more comprehensive scenarios by updating other factors to see the current shortcomings of the task flows.

A simulation run is defined by a scenario. With this scenario definition, the framework can be run without human interaction. This enables the analyzing of different parameters of the

simulation, running the same simulation several times to collect statistical information about the outcomes. Although the same simulation can be run exactly the same, environment's physical simulation presents randomness to each run that will end up in different results.

A scenario is defined using three components :

Agent hierarchy definitions: Agent hierarchies define the existing military forces in the simulation environment. Each team is represented by a hierarchy and can follow a task flow.

Task flows: Task flow defines how an agent behave in the simulation. It is a graph like structure where nodes represent the task to be pursued and edges representing the conditions that will lead to task switch in the graph. In order to capture strategic decisions at critical states of the simulation, high level situation awareness and decision making should be encoded in conditions for each team that would be involved in the course. Covering all states is not practically feasible and not an easy process. However task flow definitions can be enriched after the analysis making them to handle broader situations successfully.

Conditions: Conditions can be defined to be referenced from the flow. A condition can be evaluated by a single physical agent in the simplest case. However beliefs of a single agent may not be sufficient to define a complex condition that is meaningful to a group of agent in the course of action. Conditions can be defined to be evaluated distributed over the interested team or by other teams.

A task is the smallest building block of a scenario definition. It can be seen as a command given to a group of agents where agents will autonomously try to perform the given command. Task implementation involves coordination of agents. Agent hierarchies is used to organize coordination by representing different task responsibilities as separate sub hierarchies. Each task implementation can be studied separately and reuse of existing hierarchies and their goals and plans is possible.

A group of agents can only pursue a single task defined in the scenario at a time. In other words, no single agent can have two tasks assigned. In this work, a task is referred to an abstract, scenario level well defined objective.

A task can have one or more subtasks that is defined to run completely separated from the task. The implementation of a subtask is exactly the same as a task. A runlevel represents all running plans and issued goals of a task distributed over the agent hierarchy. When the running task triggers to handle the situation with a subtask, the runlevel of the team changes and subtask is started. In other words, all the goals and plans of the initial task is suspended in the hierarchy and a new runlevel is created to hold new goals and plans of the new subtask. After finishing the subtask, team continues its task execution by discarding the current runlevel and resuming the previous runlevel.

The framework can also be used in training simulations. The simulation needs to run in a constant rate and allow trainees to command group of agents with assigning tasks. A graphical environment is needed for trainees to help assessing the situation. It should also support easy task instantiation to be followed by the agent teams. In this simulation setting, trainees are performing high level situation assessment and strategic decisions and agents autonomously follow the given orders.

The framework can also serve as a testbed for cooperative operations defined with procedural knowledge. An example can be an evacuation simulation to test coverage of the emergency procedures that will be followed by officers.

3.1 Definitions

Every military unit is controlled either by a single physical agent if it is a single actor or by a team agent if it has more than one physical agents. Physical agent's and team agent's beliefs, goals and plans may differ significantly.

Available military units for the scenario are called scenario level actors in the simulation and task flows are defined for their operation.

3.1.1 Physical Agent

A physical agent represents a single physical actor in the environment. Environment holds only physical agents.

A physical agent's goals and plans can be specialized to suit the controlled physical entity in the simulation. For a given simple *moveToLocation* goal, a soldier and a vehicle can generate and use different dash actions in the environment.

3.1.2 Team Agent

Team agents are used to coordinate lower level agents of the hierarchy where these agents can be either team agents of physical agents. Team agents are virtual agents which are not represented in the environment. A team agent governs the decision power of its level in the hierarchy. A hierarchy of a group of agents are driven by the task they run. A simple task may need a hierarchy that top level team agent treats all lower level agents as equal. However complex tasks need to subgroup the agents to fulfill different objectives. For example, agents can be grouped in two so that the group behind has the responsibility to defend the leading group ahead in case of attack.

Team agents are used to represent higher level of decision in a hierarchy in a task. Each team agent can have different goals and plans dictated by the implementation of the task. Since hierarchy of an agent group is determined by the task implementation and this hierarchy should be changeable from task to task, team agents should not need to store and access any belief between tasks.

3.1.3 Agent Hierarchies

There can be situations where a group of agents should be organized precisely. In order to address this need , agents can be defined with a tree like hierarchy structure where the leaf nodes are the actual physical agents present in the environment. Nodes other than the leaves can be seen as virtual team agents. They are virtual because they have no physical existence in the environment. Also, they are team agents whose purpose is to coordinate the necessary actions of the leaf agents.

Figure 3.1 shows two common agent hierarchies. A tree with physical agents as leaves and team agents as nodes is a valid agent hierarchy. Figure contains abbreviations PA for physical agent, STA for sub team agent and TA for team agent. For example, if a tank team will travel

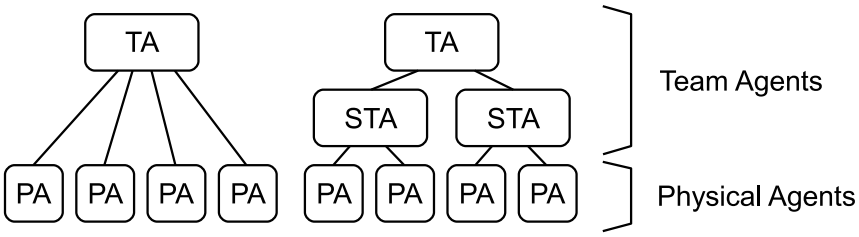

Figure 3.1: Agent Hierarchies

together in column formation, hierarchy on the left is appropriate. However If the team need to travel overwatch, a sub group should be covering the other group leading the route, the hierarchy with 2 STA is the appropriate for this task.

Agent hierarchy abstraction gives the opportunity to state a higher level objective that should be pursued with a group of agents. Furthermore agent hierarchy abstraction enables reuse of other defined objectives for hierarchies, top-down and bottom-up objective modelling.

While physical agents interact with the environment, team agents need not to be able to interact with the environment. Team agents also do not need to hold scenario specific information. All the necessary information is included in the operation parameters or physical agent beliefs. This makes hierarchy handling easier and enables to change the hierarchy totally between operations.

3.2 Simulation Framework Abstract Model

The proposed simulation framework is suitable for event based simulation where time is incremented in discrete time steps and simulation needs agent actions for each time frame. Agents are expected to run classical sense-think-act cycle at each iteration in these simulations. Agent interpretation is based on BDI architecture and enables easy agent implementation and reuse of building blocks. This section will try to introduce the abstract agent framework design and its components.

Figure 3.2 shows a simple overview of the framework. Environment and agent simulation is handled separately. At each time tick, environment provides perception information to the agent simulator, which includes percepts of all individuals that physically exist in the environ-

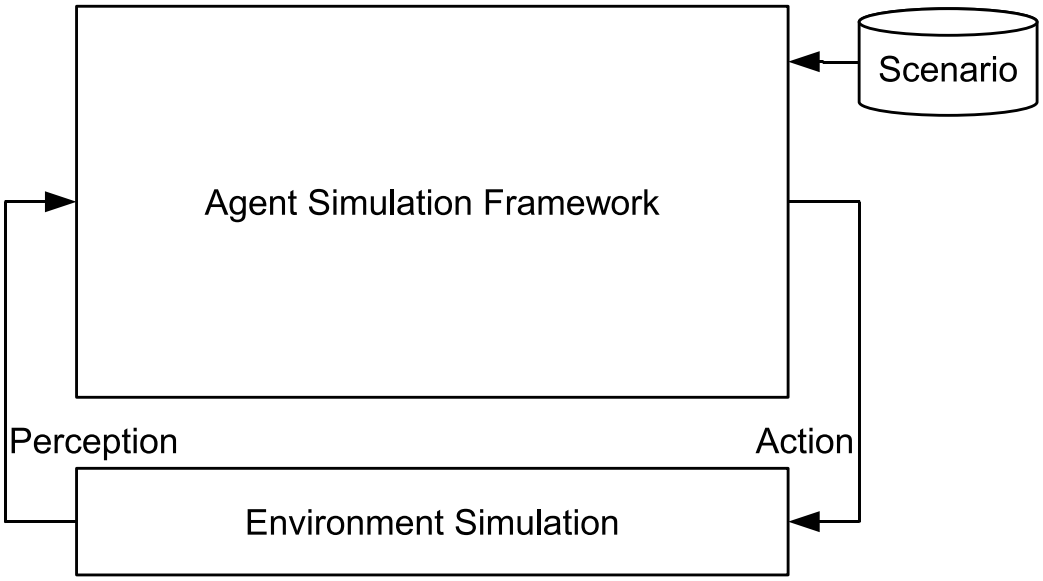

Figure 3.2: Abstract Architecture

ment. After receiving perception, agent simulation framework will perform computations to generate actions and deliver them to the environment. Agent simulation realizes the activities dictated by the simulation scenario. These ingredients represents how the agent simulation is decoupled from the environment simulation.

Figure 3.3 contains the design of the agent simulation framework. Perception distribution and action collection components have only one purpose, interfacing with the environment. Agent interpreter is the component that all the created agents live and run. Scenario manager sitting on the top has the responsibility to read and prepare the agent interpreter for simulation. In simulation, scenario manager is responsible to answer queries of the agents about their next task and conditions to follow. Scenario manager is also capable of manipulating agent hierarchies for fulfilling aggregate deaggregate operators in taskflows.

A group of agents has one root team agent which is responsible to coordinate the group and pursue the given task with its conditions. The coordination and interaction between the agents in the hierarchy is defined by the task implementation. Each task can use different mechanisms to implement the necessary coordination of the task and this can result in different agent hierarchy usages.

Agent interpreter in Figure 3.3, represents the interpretation of a group of agents presented

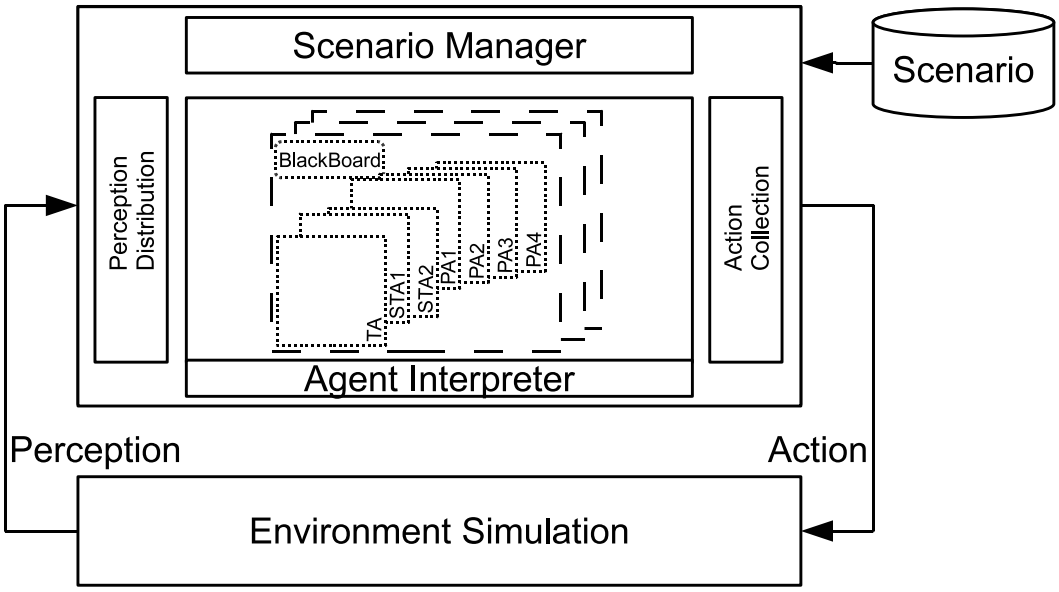

Figure 3.3: Detailed Abstract Architecture

with dotted rectangles namely TA, STA1, STA2, PA1, PA2, PA3 and PA4 with their blackboard on upper left inside a dashed rectangle. TA is the short form for Team Agent, STA for Sub Team Agent and PA for Physical Agent respectively. An interpreter can have arbitrary number of agent groups with arbitrary hierarchy settings.

Since only physical agents have presence in the environment, actions to be interpreted by the environment are issued by only physical agents.

3.3 Scenario Manager

Scenario manager accesses scenario information and creates the agents in the agent interpreter. After agent creation, simulation can start. Agent creation can also be done during simulation run. Agents in the simulation may query the scenario manager which task to follow. Scenario manager is also responsible to change agent hierarchies based on the course of task flows with aggregate and deaggregate operators.

3.3.1 Simulation Flow and Interaction between Environment Simulation and Agent

The overall simulation system is designed to operate at constant time intervals.

When simulation is advanced, sensor information computed by the environment is fed into the physical agents. This feed triggers per simulation time frame decisions coded in terms of plans. Within this decision time, physical agents can coordinate with team agents and send primitive actions to the environment. When computation of all the agents are completed, the simulation advances the time.

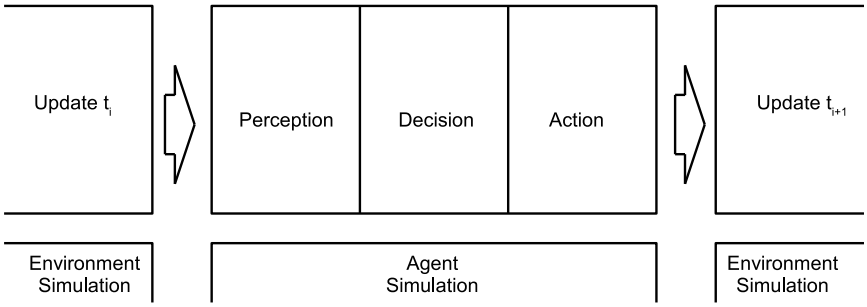

Figure 3.4: Simulation Flow

Figure 3.4 shows the simulation flow of a single cycle for a single physical agent. At each cycle, agent receive sensory information and handles them as perception. After perception related updates, decision phase should take place to determine the necessary actions. Action phase includes action collection and delivering them to the environment.

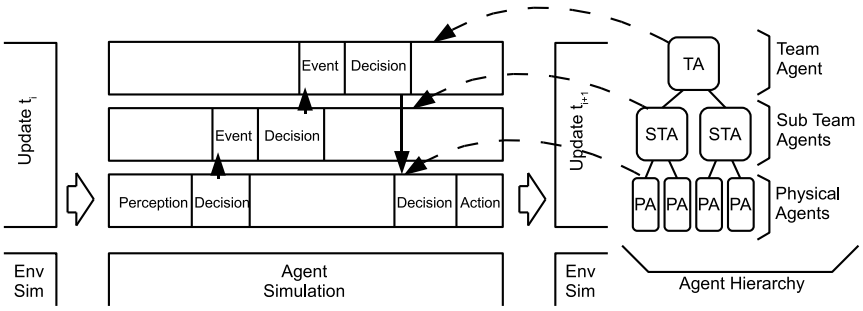

Figure 3.5: Detailed Simulation Flow

Figure 3.5 illustrates a single cycle of the agent simulation for a team of agents. Since team agents are not represented in the environment, only physical agents can receive perception. This perception phase is followed by the decision phase in a physical agent, and it can trigger upper team agents through events. Relevant information for task execution can be propagated upward in plan implementation and necessary high level decision are propagated to physical agents making them to generate correct primitive actions.

In real-time simulations, simulation is incremented with a constant time frame that is parallel with the outside world. The frequency of time frame incrementation and yet its duration is determined by the requirements of its simulated model. For example, incrementing a simulation of a human recognition simulation with 1 minute time frame is too rough.

Some simulations can not be made real-time because of its high computation needs. In these cases, simulation can be ticked on constant intervals but actual incrementation of time is slower than real. In such settings, real world interaction opportunity is lost.

Time requirements of simulation are not constant in every time frame. In order to utilize computation power, a mechanism to detect when to increment time frame should be available. Plans can be written in a way that will tell the simulation that it has completed its computation for that time frame. By this way, the simulation is incremented as soon as all the plans are completed computation relevant to the current time frame. A plan is able to work arbitrarily long. A plan can force the simulation to wait its computation to finish by not telling it is ready for time increment.

In this framework, a mechanism is introduced that can be used in plans to signal that the plan is ready for next tick. With this mechanism, simulation can be run in two different modes without affecting the task implementations. In the first mode, simulation advances with constant time frames at constant rate. In the second mode, simulation advancement is not at constant rate although time frame is constant. It allows computationally light time frames to be skipped fast. Computationally heavy time frames can consume all necessary time for their processes. The advancement rate is solely dependent on computation load of the time frames.

3.3.2 Task Queries and Scenario Manager

Scenario manager is the first point to handle the scenario information. With this information, it creates the agents in the agent interpreter. Also, it is responsible to manage hierarchy properties of the agents. The generated agents have to follow *scenario goal* as default goal. When the agents start to execute, these goals will make each agent to request its first task from the scenario manager.

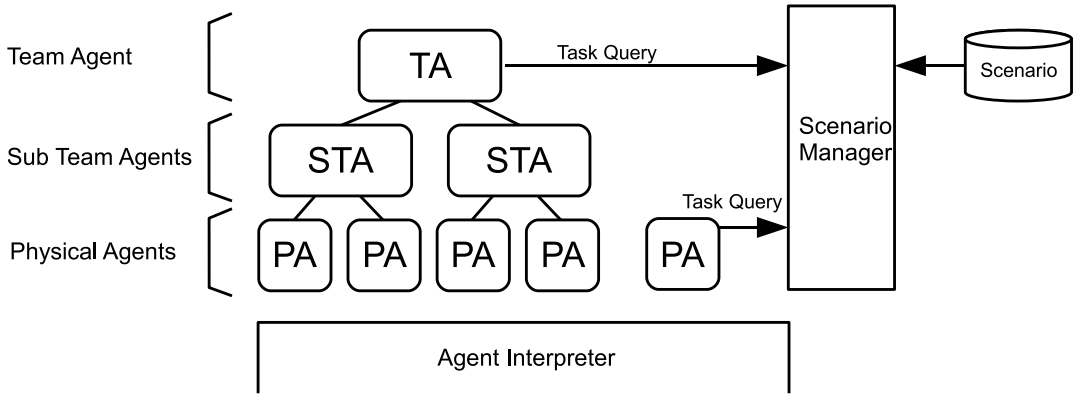

Figure 3.6: Scenario Manager and Task Queries

Figure 3.6 shows how top level agents communicate with the scenario manager. From the perspective of scenario manager, each query should come from either a single physical agent or from a team agent that is also at the top of the hierarchy to determine next task to follow and conditions to trigger task change. Scenario manager answers the query with a task and a set of conditions, which can trigger task change, as defined in the taskflow.

Agent then starts executing the given task and checks the conditions at each time frame. If no condition evaluates to true, task execution is done normally and agent pursues the task until it completes either with success or failure. The next task query will include the result of the last pursued task. If one of the conditions evaluates to true in any time frame, agent stops the task execution and makes a query stating which condition triggered task change.

An agent can encounter aggregate and deaggregate operators in its taskflow. Scenario manager is responsible to create new team agent or dispose unnecessary ones for the agent hierarchy. These operators are not treated as a task in agents.

A taskflow of an actor can be defined in a tree structure. Bookkeeping of task positions of all agents are done by the scenario manager. If an agent is in a state where no task is defined, scenario manager will reply the task query with empty task.

As the reply of task query contains a task and a set of conditions, a condition can trigger while pursuing the task. The taskflow of the agent contains the next tasks for each given condition. In this circumstance, agent will query its next task by stating which condition triggered this query.

3.3.3 Follow Scenario Goal

Each created agent initiates *follow scenario* goal as default. With the help of this goal, agents pull their tasks from the scenario manager and start pursuing them.

Only the agents that represent scenario level actor is designed to make task queries to the scenario manager. Physical agent can only make queries when it is representing a single scenario level actor. In simulation run, physical agents can become a member of an agent hierarchy. In this settings, physical agent's goal will not make queries as it has an upper level agent pursuing scenario tasks. Only the team agents that are at the top of the agent hierarchies can make scenario queries. In simulation run, physical agents can leave the agent hierarchy either by deaggregate operator or by death. The team agent will not make any queries if its hierarchy does not have any physical agents. In other words, only the agent at the top of an agent hierarchy is responsible to make scenario queries.

```
public class FollowScenarioPlan extends Plan {
  public void body() {

    AgentIdentifier scenarioManager = searchScenarioManagerAgent();
    Agent myself;

    TaskChange cause = TaskChange.INITIAL;
    QueryAnswer qresult;

    registerForTickWait();
    while(true) {
      //wait if not root
      while(true) {
        myself = (Agent)getBeliefbase().getBelief("my_self").
            getFact();
        if(myself.getHierarchy().isRoot(myself))
```

```
            break;
          waitForNextTick();
        }
        // i am root of the hierarchy
        while(true) {
          qresult = queryNextTask(scenarioManager, cause);

          for(ScenarioCondition c : qresult.getConditions()) {
            prepareConditon(c);
          }
          Goal goal = startTask(qresult.getTask());

          // execute task
          while(qresult.getTask != null) {
            if(myself.getHierarchy().isRoot(myself))
              break;
            if(goal.finished()) {
              cause = goal.getFinishState();
              break;
            }
            //check for conditions
            for(ScenarioCondition c : qresult.getConditions()) {
              if(c.isTrue()) {
                cause = c.getTaskChange();
                qresult.getTask = null;
              }
            }
            waitForNextTick();
          }
        }
      }
    }
}
```

3.3.4 Distributed Condition Evaluation

A taskflow incorporates conditions to ensure synchronization between relevant taskflows. Each task has its default success and failure conditions defined in the task implementation. In a taskflow, these default conditions can be used as a decision point for choosing next task. These triggers can be enriched by assigning user defined conditions, enabling task change whenever the user defined conditions hold. For example, an air defense team will rendezvous with a tank team at a position. The air defense team can wait for a predefined time so that if no tank team reaches the rendezvous point in that time, joint operation will fail. However a condition representing tank team arrival to the predefined point can be given. If this condition triggers, joint operation encoded in the taskflow after the condition will be followed.

A task finishes whenever a default condition (success or failure) or user condition triggers. Then, agent queries the scenario manager to fetch its next task. Scenario manager will reply the query with a task and condition set as defined in the taskflow.

Agents can watch a condition to trigger various activities. In a distributed multiagent environment, complex conditions can emerge with the necessity to fusion different information among other agents. The presented distributed condition evaluation mechanism is intended to address this gap.

The conditions can be constructed from well formed formulas and can be defined to be evaluated with different agents. When an agent needs the value of the condition, the condition is decomposed for each relevant agent and requested for evaluation. This request establishes a value link between the requester and the evaluator. This value link is only utilized when the condition value changes. The link is also disposed when the condition evaluation is no longer required.

3.3.5 Aggregate and Deaggregate Operators

It is possible to change agent hierarchies in taskflow definitions. Aggregate and deaggregate operators are defined for this purpose. A tank team can be deaggregated to 4 single tanks. With this deaggregation, user can define more precise and low level taskflows. Another example would be to aggregate 4 single tanks to form again a tank team and make this tank team pursue team taskflows as before.

Although aggregate and deaggregate are used in taskflow, their interpretation is different. Tasks are interpreted by agents but aggregate and deaggregate operators are interpreted by scenario manager.

Deaggregate operator can only be issued to scenario level actors which are also made up of smaller scenario level actors. For example, if a tank team is made up of 4 single tanks and a single tank is also a scenario level actor, user can deaggregate it to 4 single tanks and use them in taskflows. Deaggregate operator makes a group of agent to divide into sub groups. Scenario manager sets the hierarchies of the agents in this process by adding or removing team agents. Each subgroup will pursue its own taskflow with *follow scenario* goal.

Aggregate operator can only be issued to actors which will result again in a scenario level actors. For example, if there is no actor defined made up of tank teams, user can not aggregate tank teams to form bigger agent groups. Execution of the aggregate operator is harder because the two agent groups need to be synchronized. Mostly one of agent groups will want to participate in aggregate operator while other agent group pursuing another task. In this state, first group will make a task query to scenario manager where scenario manager will not answer. Unanswered queries will be re-sent next time frame again. Meanwhile, scenario manager knowing an actor ready for aggregate operator, will make that agent wait for the other agent. When both participants are ready for aggregate operator, scenario manager will manage their hierarchies. This will result in a new task query from the top most team agent as other actors will no longer be the top most agents in the hierarchy.

3.4 Agent Interpreter System

Agent interpreter system is the core of agent execution. This framework uses Jadex BDI interpreter as a base for agent interpretation. BDI agents allow scalability and distribution.

3.4.1 Blackboard

A team of agents should be well synchronized to fulfill the goals they are pursuing. Agents need to communicate for synchronization in their plan executions. The need to communicate can be just to synchronize at a time point or to exchange information necessary for plans.

Implementing the behaviour of a group of agents centrally in a single algorithm is not suitable in the nature of the framework. The central algorithm indeed can be converted to distributed goals and plans utilizing messaging between them. However, implementing plans with explicitly messaging not only complicates the plan implementation, it can introduce overhead compared to using a blackboard.

Blackboard addresses the need of synchronization facilities and joint beliefbase for coordination between the running plan instances. In a fully distributed setting, blackboard can utilize messaging. For performance improvements, it can serve as an interface to bypass and simplify the need for messaging. For example, it is possible to use shared memory and semaphores

in a single computer while being able to fallback to a message based approach whenever the participant agent is not locally available.

Jadex is a fully asynchronous event based interpreter. In order to ensure synchronization, one has to define several messages, handlers in terms of events and plans. Using blackboard in plan implementations eliminates these need in an efficient way.

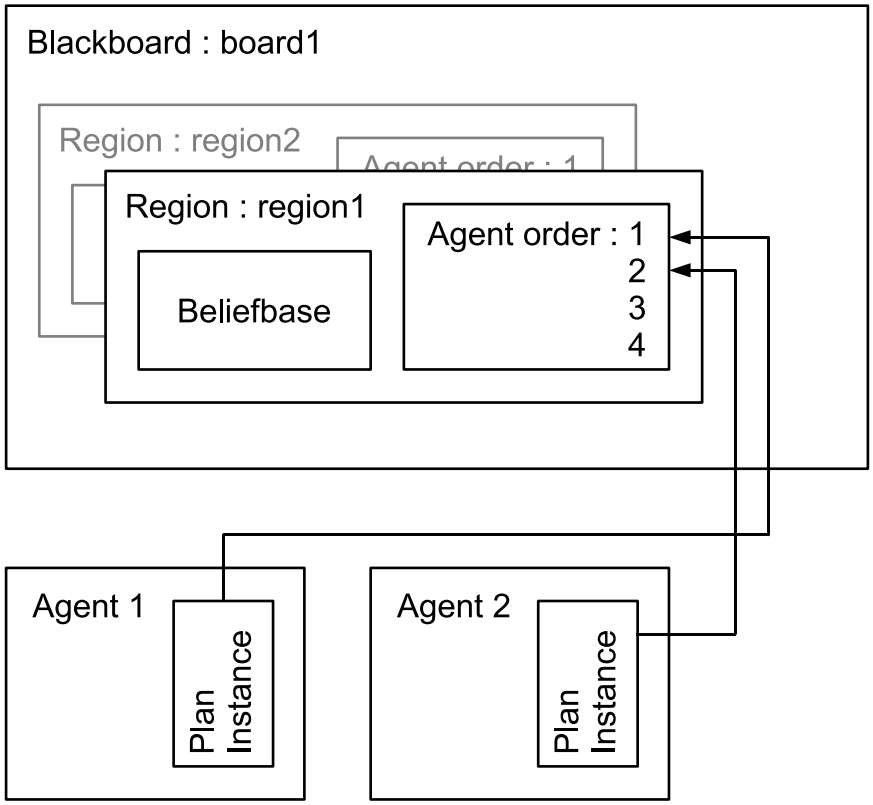

Figure 3.7: Blackboard

Blackboard has two functionality that can ease coordinated plan writing. First functionality is to address synchronization of different plans of team agents. Second functionality is to address the need of common beliefbase shared between agents of team. Figure 3.7 illustrates a blackboard.

Blackboard properties are defined below:

Blackboard regions : A blackboard is used in a task implementation as a tool. A blackboard

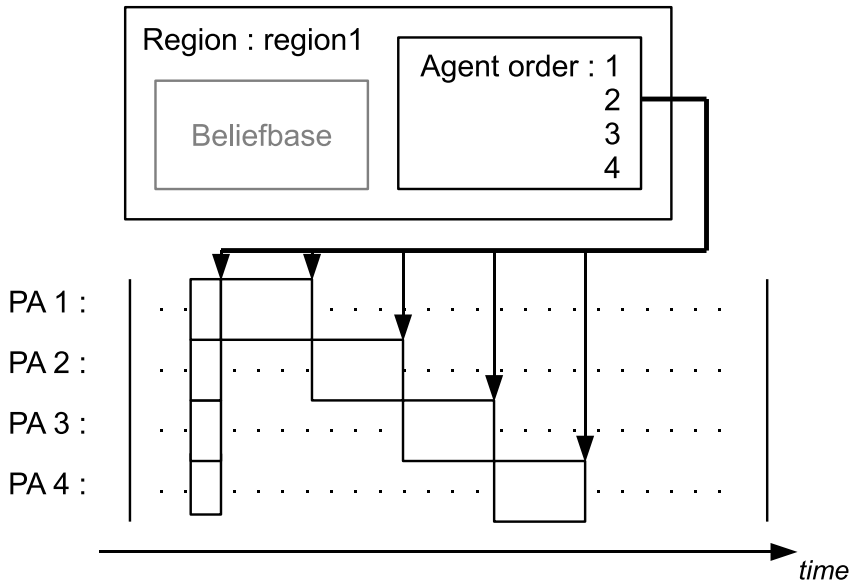

Figure 3.8: Blackboard Agent Order for Plan Execution

can have independent regions that can be used isolated from other sub teams.

Agent orders : Some tasks require decision of actions in a single time frame. The agents in the hierarchy runs their plan in parallel. If there is a need to ensure execution order of some plan steps, agent orders in the blackboard can be used. A plan can wait for its execution order before committing plan steps. For example in the simple setting, the programmer wants 4 agents to run orderly. He can define a blackboard region with agents in order and he can use synchronization in the plans of physical agents. Each agent will decide where to go and write it to blackboard beliefbase, and the next agent will decide based on previous agent. This facility enables agents to synchronize efficiently in order and generate primitive actions at each time frame. Figure 3.8 illustrates the usage of agent order of a region for a single time frame. In the example figure, physical agents execute common steps of plan but then uses agent order for synchronously executing rest of the plan steps.

Blackboard beliefs : Utilizing blackboard as a common beliefbase between agents is possible. However the lifetime of a blackboard is intended to be for a single task, like beliefs of team agents, which are disposable between tasks.

A blackboard is intended to be used by agents of a single group only, in other words no two

team is supposed to use the same blackboard for the sake of distribution. In distributed agent execution setting, this property can help agent relocation between the machines, enabling efficiency gain by using local machine blackboard interaction facilities. If task implementation needs blackboard usage, the team agent is responsible for its creation and setting. Necessary information to access the created blackboard can be passed in goal parameters afterwards while delegating goals to sub agents.

3.4.2 Runlevel

A task is executed by a group of agents organized in a hierarchy. In order to start a new task, previous task and its relevant goals and plans should be stopped. Runlevel enables to control all running plans of a task distributed over the agents in the hierarchy whenever necessary.

Function calling another function is a way to decompose and implement tasks. The analogy in BDI architecture is to issue sub goals in a plan. These approaches are for single control flow and called function stack and intention stack respectively.

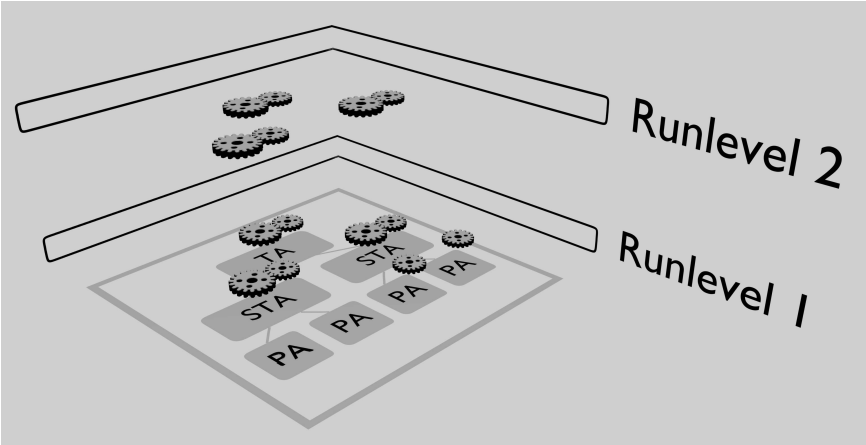

Figure 3.9: Runlevel

Runlevel is an interpreter control mechanism that can not only control a single agent goals and intentions but also control the whole agent hierarchy.

Runlevels can be seen as a stack where each level is on top of other and only the top most

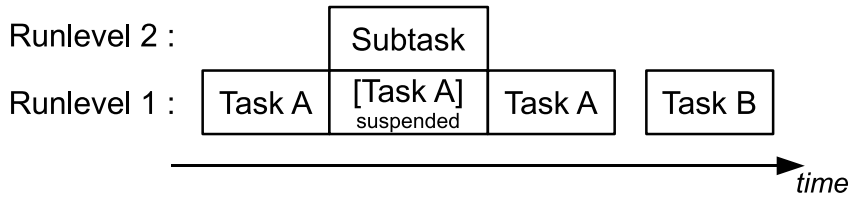

Figure 3.10: Task execution with Runlevels

level runs. Figure 3.9 tries to illustrate runlevel stack for a group of agents. In the figure, agents in the hierarchy has gears, representing the running plans, aligned in runlevels. With this abstraction, a task is running with all its relevant goals and plans on a level. With runlevel functionality, one can call subtasks on a higher runlevel, making the current runlevel suspended. These subtasks can be executed to handle different conditions and help maintain the task to be pursued. After handling the situation, the runlevel can be disposed and all agents can continue from the states of the previous runlevel. Figure 3.10 illustrates the explained task execution.

Runlevels are named with integers starting from 0. Runlevel 0 is a special runlevel that helps to easily define service plans that should not be suspended. When a subtask or team plan should be run, runlevel is suspended and incremented.

Below is a list of possible runlevel usage:

Runlevel 0 : *Follow scenario* goal runs in this runlevel and never discarded through out the simulation. This runlevel can be seen as a special container for goals that should not be affected by task execution. *Follow scenario* goal starts the given task in runlevel 1 and controls its execution.

Runlevel 1 : This runlevel is suitable for task execution. All goals and plans of the task are interpreted in runlevel 1.

Runlevel 2 : This runlevel and higher runlevels are available for subtask execution. A subtask is triggered when its condition evaluates to true. This runlevel is discarded when the subtask execution is finished and suspended lower runlevel is resumed.

Runlevel is controlled with a message passing mechanism that is treated specially by the agent interpreter. The message can create a new runlevel or dispose the current one. Creation of

a runlevel suspends the current runlevel and it is not resumed until the created runlevel is disposed. The message also includes the agents in the hierarchy so that this control message can be broadcasted from the top agent to all runlevel participants.

3.4.3 Conditions and Subtasks

A plan can have various conditions watched in parallel with its normal plan execution. When one of these conditions holds, the condition triggers various measures.

1. It can finish the plan execution with either success or failure.
2. It can call a subtask with a higher runlevel, making itself and all subtask below the hierarchy suspended.
3. It can replace the current triggering goal with another goal.

3.4.4 Inter Agent Goal Request, Observation and Control

In a hierarchic team programming framework, ability to request an activity from another agent is a key control facility. Observing the requested activity and having opportunity to control the request is also crucial. It enables to abort activities that are no longer necessary.

This extension is designed like the distributed condition evaluation in terms of value links. When a requester asks another agent to pursue a goal, a link about the goal status is established. With this link, observer is informed about the state of the goal. This link also enables the observer to cancel the goal request as soon as the fulfillment of the goal is not necessary.

3.4.5 Definition of a Scenario Task

A task in scenario is defined with parameters. The task has a corresponding goal that is pursued by the top level team agent in the hierarchy.

The plan that will fulfill the goal also can have conditions that can trigger different runlevel operations.

3.5 Perception Distribution

Perception distribution component illustrated in Figure 3.3 is an interface used to feed the agent simulation with the generated sensory information in the environment simulation. Agent health states are fed to scenario manager too. If an agent is killed in the simulation, scenario manager updates the hierarchy that the agent is member of and kills the interpreter of the agent.

3.6 Action Collection

Action collection component is illustrated in Figure 3.3. It is also an interface point like perception distribution component. It simply collects all primitive actions from physical agents and delivers them to environment simulation.

3.7 Realization of Scenario Tasks in Agent Interpreter

Scenario tasks can be assigned to scenario level actors. A task can be given either to a single physical agent or to a team agent. Agents that should act as a team will organize themselves in an agent hierarchy. For example, if a tank team consisting of 4 tanks should operate in two sections, the team hierarchy will need 3 team agents as can be seen in Figure 3.11. If the task and its implementation does not need subdivision and corresponding subteam agents, The team will contain only 1 team agent as in Figure 3.12.

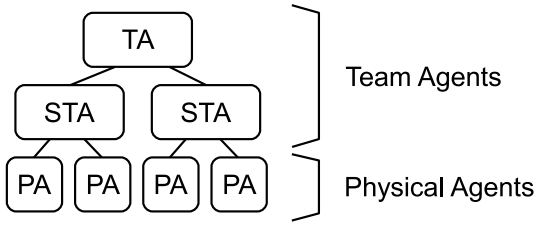

Figure 3.11: Agent Hierarchy Containing Subdivision

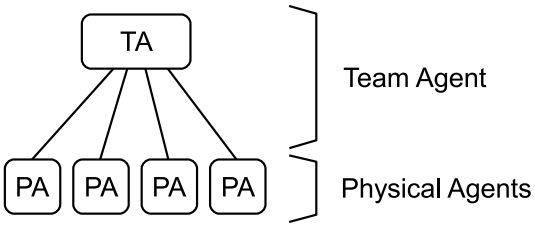

Figure 3.12: Simple Agent Hierarchy

A task has conditions that can trigger task change when evaluated to true. The agent that is at the top of the hierarchy is responsible for evaluation of these conditions. In order to fulfill task objective, agent at top will utilize lower agents by sending goals. A task implementation includes all necessary goals and plan definitions that handles these goals and events. Utilization of the blackboard and subtasks will be explicitly stated in modelling section of tasks.

Team agents are virtual in the simulated environment. In other words, they can not send primitive actions to manipulate the environment or receive any sensory information directly from the environment. Team agents are used to organize physical agents centrally and accommodate coordination between them. Team agents decide which underlying agent should do by sending goal. Physical agents that receive goals will instantiate a plan as an intention. From this point on, a physical agent can generate primitive actions according to their plans and send these actions to the environment.

Task definitions has failure and subtask conditions. This conditions will be watched with the plan of *follow scenario* goal in runlevel 0. During execution of a task, any of the conditions can trigger. If the failure condition triggers, the goal of the task will return with failure. In the case subtask condition triggers, the current task is suspended and a new runlevel is created to run the defined subtask. When the execution of subtask ends, the runlevel of the subtask is disposed and the suspended runlevel is resumed to continue its task execution.

In task implementation, a subtask that can be triggered by a subtask condition can be specified. Implementation of a subtask is exactly the same as a task. The task will be suspended temporarily when a subtask starts to execute. This property allows a subtask to run without any interference from other tasks.

CHAPTER 4

IMPLEMENTATION ON JADEX

During thesis work, a proof of concept implementation of the proposed architecture is made and an example case study is carried out. Since abstract architecture is based on BDI paradigm, a BDI implementation is needed that is suitable for extension and open source. Jadex, which is a well known BDI Agent Interpreter, is chosen because it is open source and extensible.

Jadex source distribution also comes with various multiagent examples but none of them contains difficulties attacked by the approach introduced in this thesis work. In order to demonstrate the capabilities, a small scale military scenario containing 2 sides, one ally and one enemy, is prepared.

The detailed case study is presented top down in three steps. The scenario and its ingredients are stated first. Modelling of the tasks that are used in the scenario is given later. Finally, an example run is narrated at the last section. In this chapter, implementation details carried out on Jadex interpreter is presented.

4.1 Implementation of Environment, Scenario Manager and Closed Simulation

Jadex platform provides its all functionality through various built-in agents. Jadex Control Center agent is a GUI front end to control everything running on the platform and it has several debug and introspection tools that can ease agent development besides overall platform control. Agent Management System (AMS) agent enables agent life cycle control by giving the ability to create, suspend, resume and kill. Jadex implementation also provides a Direc-

tory Facilitator (DF) agent whose purpose is to provide directory services to help agents find information easily. In Jadex platform, all agents communicate with sending and receiving messages.

For the implementation of the case study, an agent named environment is created that is responsible for environment simulation. The environment agent then registers itself on director facilitator agent. All other created agents fetch the environment information through directory facilitator.

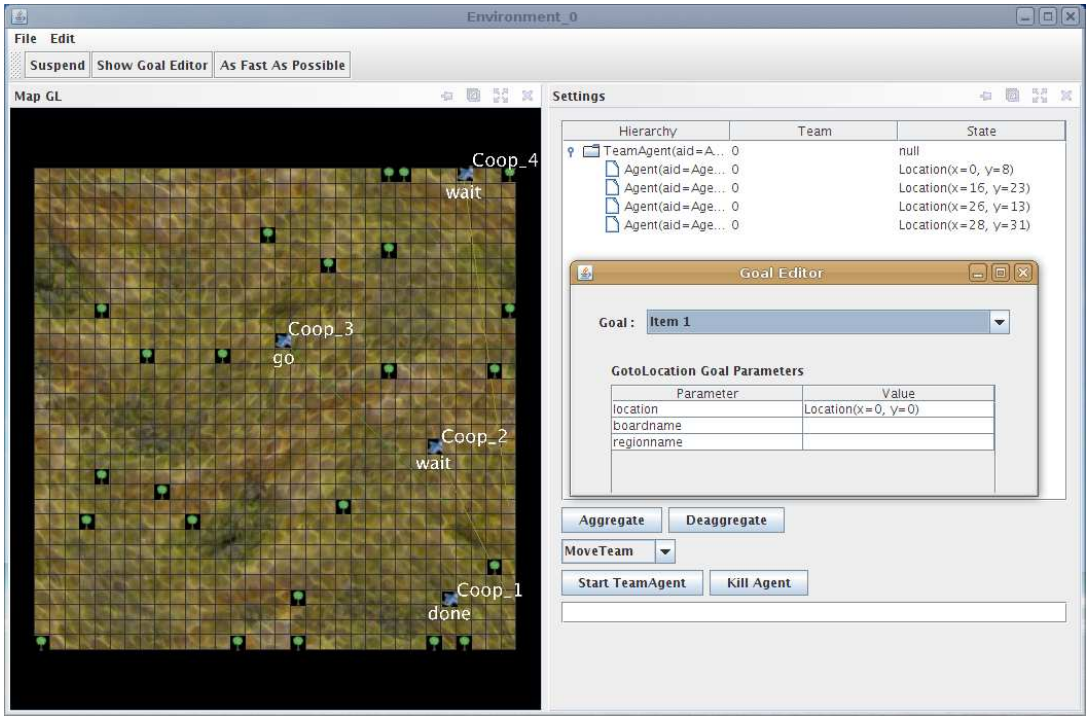

Figure 4.1: Environment Gui

Environment agent has a GUI (Figure 4.1) to show the simulation state and states of the connected agents. The environment GUI also allows control of the simulation execution. The suspend toggle button enables to pause the simulation. Pause can be used to study the logs of the ongoing simulation. Button named as fast as possible can be used to toggle between normal realtime incrementation of the simulation and fast incrementation as soon as the computation of the plans finishes. For easy and basic testing purposes, GUI has a simple goal editor that lets chosing a goal and editing of the goal parameters. After setting the goal pa-

rameters, the goal can be dragged and dropped onto any agent in the agents view. If the agent has the goal defined, it will start to execute the goal immediately.

In order to simplify coding, scenario manager component is also built into the environment agent. However, creating a different agent for scenario manager is also possible. Scenario manager has two main purposes: initialization of agent simulation framework and task query answering.

In initialization phase, scenario is read by the manager and necessary setting of the environment is also done. Afterwards, all agents defined in the scenario is created and their hierarchy is set up accordingly.

In simulation run phase, scenario manager has no active tasks, its main responsibility is answering queries about which agent should follow which task.

In order to answer queries, scenario manager keeps track of task flow positions of each agent. This book keeping process is carried out through out the simulation. Each query involves an advancement of the agent in its taskflow: asking for next task to pursue. The latest given answer to the query is the current position of the agent in its taskflow.

Scenario manager is informed when an agent is destroyed in the environment. Scenario manager than kills the agent and removes from the interpreter.

Task following from the agent perspective is also needed to be implemented. This is accomplished by a simple initial goal that queries a task to follow in a loop. The details of this goal is given in previous chapter as follow scenario goal.

Closed simulation property makes scheduling of the agents harder. In order to utilize computing resources efficiently, advancement of the simulation time frame should be made as soon as possible. From the nature of distributed systems and a central environment, completion of decision relevant to the current time frame should be detected and simulation should be advanced. In order to address this need, wait manager is implemented in the environment agent and team plans coordinate with this manager by messaging. A time consuming plan can register itself on wait manager to ensure it has enough time at each time frame. The plan signals when it has completed its computation for that time frame. Wait manager keeps track of these plans and advances the simulation time whenever all plans are ready for the next time

frame. Using this facility from plans is accomplished by two function calls.

registerForTickWait() : This function immediately returns after sending a register message to the wait manager.

waitForNextTick() : This function signals that the plan has completed necessary computation for the current time frame and ready to advance. This function returns when the simulation time is incremented.

If a plan is registered for tick wait, upon plan termination it automatically unregisters itself from the wait manager.

Wait manager keeps track of all requests in a list identified by <*agentId, plan instance*> pair. If a plan is unresponsive, which means it does not call *waitForNextTick*, for a defined number of time frames, it is automatiaclly discarded. Wait manager can operate in two modes.

Normal : In this mode, wait manager advances time based on simulation defaults which is 1 tick per second.

As fast as possible : Wait manager is utilized in this mode. It keeps track of all registered plans and advances as soon as all of them completes their computations. Wait manager increments the time frame at worst case as in normal mode. This can be caused by plans having heavy computation loads or unresponsive plans. The plan instances that slows the execution are logged for further investigation.

4.2 Implementation of Runlevel

Runlevel mechanism is incorporated into the Jadex agent interpreter. Jadex agent interpreter is event based and event processing is done with the help of an agenda that organizes the pending events in tree structures. Running plans may not have any pending event on agenda. However, when a plan goes to sleep waiting for a condition, the condition can generate an event to wake up the plan. The triggered event will first take its place in the agenda. Interpreter processes events in the agenda one by one.

Runlevel information is incorporated on running plans and the interpreter holds the current runlevel. When a new plan starts to run, it is created with the current runlevel. Two field and a couple of management function are added to JadexInterpreter class. These additions are shown below.

```
private int currentRunLevel = 1;
private Stack<IAgenda> suspendedRunLevels = new Stack<IAgenda>();
public synchronized void newRunLevel(RunLevelRequest rlreq);
public synchronized void removeRunLevel();
public int getCurrentRunLevel();
public IAgenda getAgendaOfRunLevel(int runLevel);
```

Runlevel change is triggered by a message event. The message event contains three type of information: command, agents and goal. Command specifies whether the request is new runlevel or remove runlevel. The request is initially sent to the root agent on top of the hierarchy. The request also specifies all the other agents of the hierarchy. A simple request consisting of only the command is then broadcasted to all agents specified in the request by the root agent. Goal information is used to run the initial goal of the new runlevel.

Jadex interpreter can be extended by tool adapters. A tool adapter is a special class that can handle message events when the agent first receives. With this capability, Jadex control center has implemented various debugger and introspector tools. In order to add runlevel tool adapter, *config/runtime.properties.xml* file is edited adding the line below.

```
<property name="tooladapter.runlevel">new y_thesis.runlevel.
    RunlevelAdapter($agent)</property>
```

This runlevel adapter handles *RunLevelRequest* messages and handles them by calling *newRunLevel* or *removeRunLevel* fuction in the *JadexInterpreter* class. Broadcasting of messages are done within the runlevel adapter.

When a new runlevel is triggered, agent actually starts a new plan that will start and monitor the given goal. The runlevel goal takes the *RunLevelRequest* message as parameter and it is dispatched as a top level goal. *RunLevelPlan* handles the runlevel goal and start the given goal as a subgoal. The runlevel is automatically removed by *RunLevelPlan* when the given goal is finished. *RunLevelPlan* sends itself the same runlevel request with remove runlevel command. Below is the source code of *RunLevelPlan*.

```
public class RunLevelPlan extends Plan {
 public void body() {
  RunLevelRequest rlreq = (RunLevelRequest)getParameter("request").
      getValue();

  // start the goal
  if(rlreq != null && rlreq.getGoalname() != null) {
   String goalname = rlreq.getGoalname();
   Map<String,Object> params = rlreq.getGoalparams();

   IGoal goal=null;
   IRGoal rgoal=null;
   try {
    rgoal = getRCapability().getAgent().getGoalbase().createGoal(
        goalname);
    goal = new GoalWrapper(rgoal);
   } catch (RuntimeException re) {
    logger.severe("runlevel can not create goal : " + goalname);
   }

   // set parameters
   if(params != null) {
    for(String param:params.keySet()) {
     goal.getParameter(param).setValue(params.get(param));
    }
   }

   // run the goal and wait
   dispatchSubgoalAndWait(goal);
  }

  // end the run level
  RunLevelRequest req = new RunLevelRequest(rlreq);
  req.setCommand("remove");

  //From ToolRequestPlan
  IMessageEvent request_msg = createMessageEvent("tool_request");
  request_msg.getParameterSet(SFipa.RECEIVERS).addValue(req.getAid
      (0));
  request_msg.getParameter(SFipa.REPLY_WITH).setValue(SFipa.
      createUniqueId(null));
  request_msg.getParameter(SFipa.CONVERSATION_ID).setValue(null);
  request_msg.setContent(req);

  sendMessage(request_msg);
 }
}
```

Runlevel change request is specially treated by the agent interpreter. Internal data structures are need to be handled prior to any runlevel related goal processing begins.

When a new runlevel is created, a new agenda for the runlevel is created and the current agenda is stored in a stack. Since interpreter will work on the events generated, previous events will not be considered. However, wake up events for the suspended plans related to the previous can emerge. In this case, event is pushed to the relevant runlevel agenda. That way, when the runlevel is resumed, previous events will be processed. This functionality is implemented in addAgendaEntry function of JadexInterpreter class.

4.3 Implementation of Blackboard

Blackboard facility can be seen as an interface, as mentioned in previous chapter, and there can be different implementations providing this interface. In case study, blackboard facility is implemented to work on a local Java runtime instance where all agents of the group assumed to live. Although this assumption seems limiting, its application has performance benefits. In order to gain performance, all agents of the group can be executed on the same Java runtime instance.

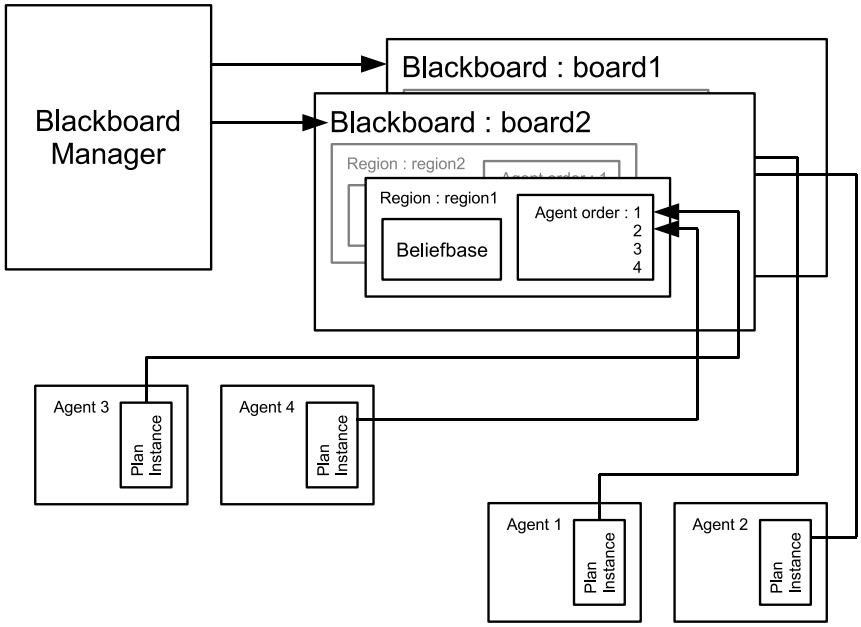

Figure 4.2: Blackboard Manager

The blackboard can also be implemented fully distributed and message based. This imple-

mentation will probably have overheads but it will enable blackboard utilization wherever possible.

In case study implementation, each Java runtime instance has a blackboard manager(Figure 4.2) as a singleton. Blackboard manager is used to create and register blackboards. After blackboard usage is over, the creator plan should remove it from the blackboard manager. Blackboard manager holds a list of available blackboard that can be accessed with names. Blackboard names are passed to other agents in goal parameters. All agents of the group can access blackboard after receiving the name.

A blackboard is a container of regions. All functionality is implemented in blackboard region. Below is the interface of the blackboard region with comments shortly explaining its usage.

```
//used to set the order of agents in creation
public void addAgent(AgentIdentifier aid);

//used by an agent to query its place in agent order
public int getAgentIndex(AgentIdentifier self);

//used to query how many agents are in agent order
public int getNumberOfAgents();

//agent can query if it needs to wait for order
public boolean isWaitRequired(AgentIdentifier self);

//called to block current agent to wait order, timeout in
    milliseconds
public boolean blockAgent(AgentIdentifier self, long ms);

//signal activity for the agent is done
public void agentDone(AgentIdentifier self);

//get common beliefbase as a hashmap
public Map<String,Object> getBeliefbase();
```

4.4 Implementation of Scenario Conditions

A scenario condition is an expression where its evaluation to true triggers task change. Jadex has support for expressions that can use functions and variables present in beliefbase.

Jadex also has support for conditions. In Jadex manual it is described that a condition is

a monitored boolean expression. Usage of conditions is possible within plans but it does not allow to explicitly wait for multiple of condition instances. A task can have multiple conditions that can trigger task change. Because of these short comings condition facility of Jadex could not be used for demonstration.

In this case study, distributed condition evaluation is needed. A scenario condition is defined with a list of <agentId, boolean Jadex expression >pairs. AgentId defines which agent should monitor the expression. If the agentId refers to another agent, a value link is established between agents to monitor the expression. After value link establishments, all of them is combined by AND binary operator and monitored.

A value link is implemented message based and uses beliefbase. Value link usage has three phases as initiation, update and termination.

Link initiation : Initiator reserves a new belief to hold value of the expression. Then initiator sends a message to the remote agent containing the expression and the belief name. Receiver registers this information for later evaluation.

Value update : Receiver monitors the expressions once in every time frame. This is triggered in follow scenario plan. If the value of the expression changes, a value update message is sent to the agent containing the belief name and its value. With this information, initiator agent directly updates its belief based on the incoming update message.

Link termination : Link termination is done with belief removal and sending a link termination message to receiver. Receiver removes any setting for expression monitor.

Evaluation of scenario conditions is handled in *follow scenario* plan. When a new task is received with scenario conditions, all scenario conditions are prepared for monitoring. A task change is triggered whenever one of the scenario conditions evaluate to true or the task goal finishes with success or failure. Task change handled by runlevel cleaning and a task query to the scenario manager.

Although Jadex condition facility could not be used in the case study, it is possible use after extending the Jadex internals. Condition facility of Jadex currently more efficient than the case study implementation.

CHAPTER 5

EXAMPLE CASE STUDY

5.1 Case Study : Air Defense of Tanks

Scenario can be briefly stated as an air defense escort operation. A group of tanks accompanied by air defense vehicles will go thorough a valley. The mission has a high risk of failure because the valley is suitable for an enemy helicopter attack.

This scenario outline describes air defense units their objective: escort the tank group and engage enemy helicopters. The success of this mission is highly depended on the capability of air defense units. With this scenario, utilization of the air defense units and how they should be operated can be studied. For this purpose, case study focused on the implementation of the tasks of air defense units and their task flow. Implementation of the enemy helicopters and tank group is not in detail and mostly hardcoded.

Map is a simple grid with obstacles that blocks the available moves. Figure 5.1 shows the beginning of the simulation. Map size is a 32 by 32 cells. Every vehicle in the simulation moves at most 1 grid in a time frame. Tanks and air defense vehicles can not move through obstacles. Helicopters can move over the obstacles. No one can see anything behind an obstacle. In this implementation, only a generic blockage is defined which is called a mine field. This mine field can be detected when the agents come closer. It is assumed that relevant actions can be taken by infantry to clean the mines. Based on this assumption, implemented environment simulation vanishes the mine field after a predefined time. If the vehicle moves in the mine field, it can be damaged with a probability.

Enemy consists of a group of helicopters. Enemy tries to secure an area and avoid penetration

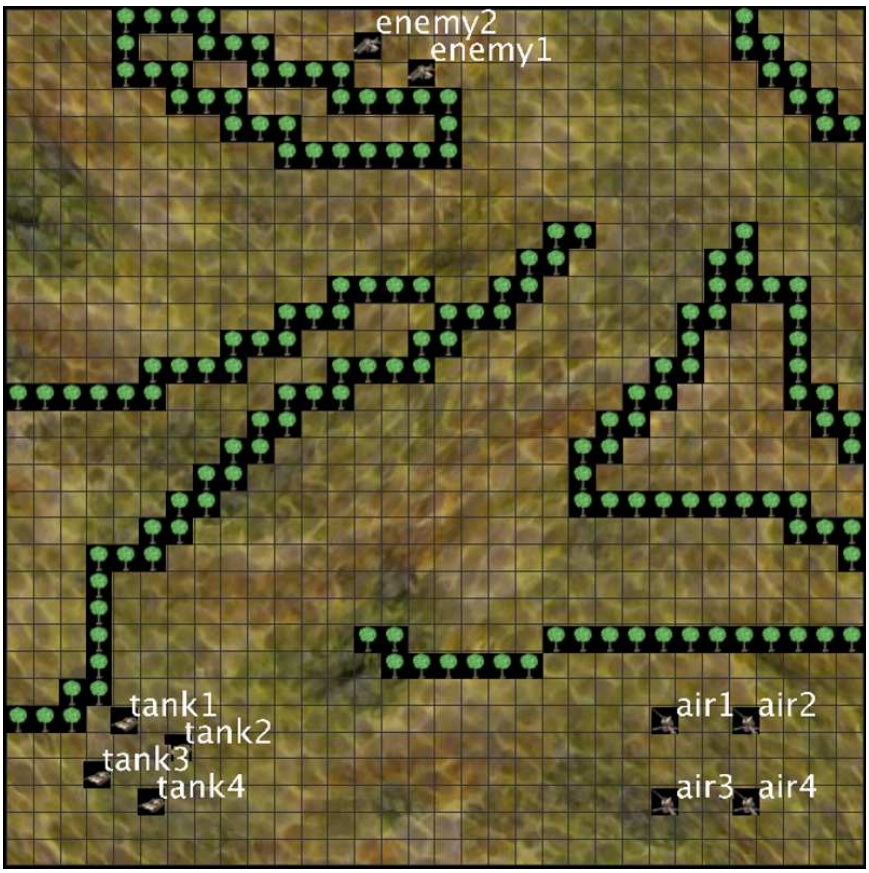

Figure 5.1: Initialization of Simulation

of other forces from the valley.

Friendly units consists of a tank group and a group of air defense vehicles. The tank group has the mission to reach and secure an area that is of utmost importance. The area is located on the upper right corner of the map. Tank group is aware of the helicopter strike risk and will be accompanied by air defense vehicles. The air defense will move with the tanks for protection. The key capabilities of the framework will be demonstrated in task implementations of this unit.

Enemy scenario and friendly scenario are run completely separated. In other words, it is possible to run enemy agents on a different computer.

5.1.1 Example Scenario

There are two friendly teams on the map, namely air defense team and tank team. Initially they are located far away from each other on the map as can be seen on Figure 5.2. Main activity of the scenario is to carry out air defenses of the tank team. In order to start this mission, air defense team should travel near the tank team. After air defense team covers the tank team, tank team can start to follow the route in the valley to north.

Friendly intelligence also states enemy presence guarding the north of the valley. It is reported that helicopter teams may be patrolling the north of the map.

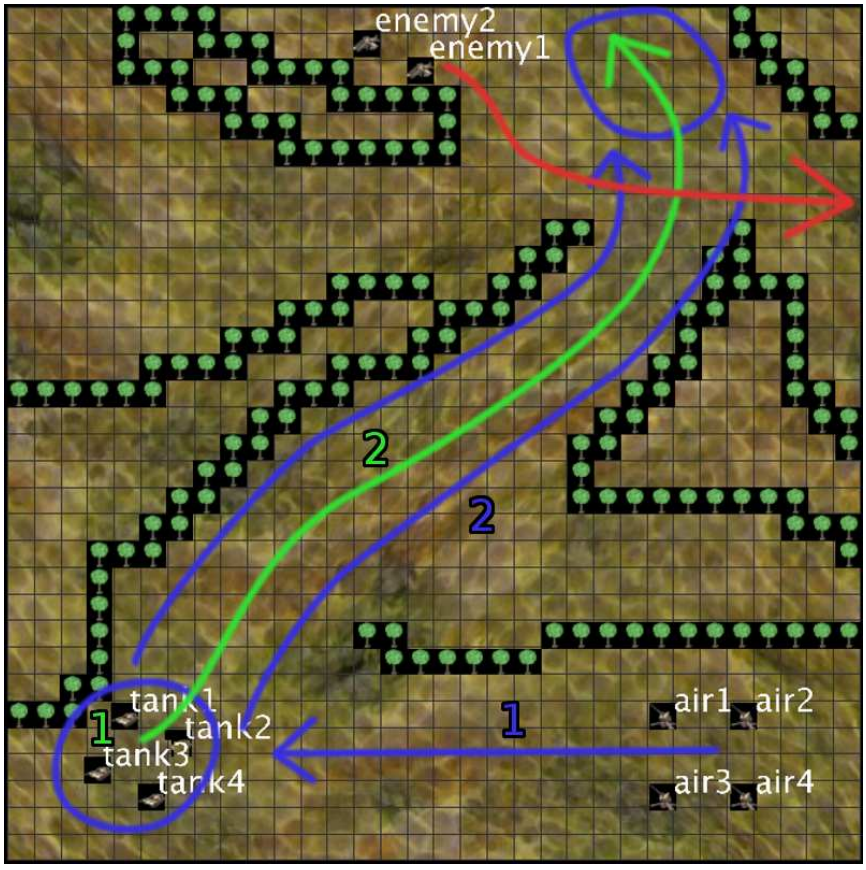

Figure 5.2: Scenario Overlay

Scenario has two phases for friendly units, first preparation and escorted travel to north.

Phase 1 : In the first phase, tank team takes position on their initial coordinates. This position

is highlighted with green 1 in Figure 5.2. Tank team will wait until accompanied by air defense team. Air defense coverage of positioned tank team is designated with blue circle on the map. Meanwhile, air defense team travels to west to reach tank team. The route is highlighted with a blue 1 on the figure. When position is reached, air defense team will cover tank team to protect from air attacks.

Phase 2 : Tank team follows the green route highlighted with number 2 in the valley. Tank team travels at a suitable speed to let air defense vehicles to change position appropriately. Air defense team pursues its defensive move task along the blue path in the valley highlighted with blue number 2. Based on the intelligence, possible enemy encounter is awaited in this phase. At the end of this phase, tank team and air defense team take positions on the north where the green route ends.

Scenario for enemy helicopters is very simple and involves no coordination. It can be summarized as:

- Patrol the red line on the north continuously.
- Attack any threat in vision range.

5.2 Scenario Level Actors

Scenario level actor refers to a single physical agent or a group of agents that is suitable for task assignment in task flow definitions. A group of agents will have a hierarchy and the topmost team agent is responsible for pursuing the task.

Air Defense Team: This actor is composed of 4 air defence vehicles.

Single Air Defence Vehicle: This is the only physical agent that is defined as a scenario level actor. User can define a single air defence vehicle to command from the beginning or deaggregate an air defense team and command all vehicles in it one by one.

Tank Team: The tank team consists of 4 tanks. The team agent has 2 sub team agents each corresponding to a tank buddy. A tank buddy is made up of 2 tanks.

Helicopter Team: The helicopter team is composed of 2 attack helicopters.

A single tank or a single helicopter is not defined as a scenario level actor. Thus user is not given the opportunity to define detailed taskflows for these units. However implementation of each physical agent is exactly the same whether it is a scenario level actor or not. Defining scenario level actors is a design decision and can be directed with the desired detail level of the simulation.

5.3 Modeling of Tasks

Below sections presents the modelling details of the tasks used in the case study. Some of the tasks reuse existing goals and plans from other tasks.

5.3.1 Goto Location Task

This task is the simplest task that can be given to a physical agent. The agent will find a route to the desired location and immediately start moving. If no valid route to the location found, the task fails.

Applicable Actors:

Single Air Defense Vehicle.

Task Parameters:

Table 5.1 defines the task parameters.

Preconditions and Assumptions:

- Actor should have mobility.
- A route to the location should exist

Table 5.1: Goto Location Task Parameters

Task Parameters	Description	Data Type
Location	The target point that should be reached by the agent	2D Map Coordinate

Task Implementation

The physical agent tries to find a path to the target location first. If the route exists, it follows the route and finishes the task successfully. Task will fail when there is no route. However temporary route blockages will not fail the task. The task can call subtasks to handle the route blockages.

Hierarchy

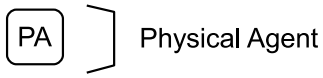

Figure 5.3: Goto Location Task Hierarchy

Single PA physical agent has no hierarchy (Figure 5.3).

Success Condition

The task is successful when the physical agent reaches to given target location. The *Task-Goal_GotoLocation* goal will finish successfully.

Failure Condition

If there is no applicable route is present to reach the target destination, The *TaskGoal_Goto-Location* goal will finish with failure.

Subtask Conditions

The route can be blocked temporarily by a mine field. In this circumstance, the subtask *Wait Mine Cleaning* can be activated in a new runlevel to handle the situation.

Blackboard Usage

There is no blackboard usage for this solo task.

Belief Usage

Environment obstacles and agent positions are accessed inside the plan to calculate possible moves. Subtask checks environment to see whether the mine field is vanished.

Task Goals and Plans

PA Physical Agent

TaskGoal_GotoLocation: This goal represents the task. Goal parameters are same as the task parameters.

Plan_GotoLocation: This plan handles the goal *TaskGoal_GotoLocation*.

```
public class GotoLocationPlan extends Plan {
  public void body() {
    Location loc = (Location)getParameter("location").getValue
        ();
    String boardname = (String)getParameter("boardname").
        getValue();
    String regionname = (String)getParameter("regionname").
        getValue();

    if(boardname == null || boardname.equals("")) {
      logger.severe("boardname is empty");
      fail();
    }

    BlackBoardRegion region = BlackBoardManager.getInstance().
        getBlackBoard(boardname).regions.get(regionname);

    //termination
    Boolean done = (Boolean)region.content.get("done");

    //loop
    while(!done) {
      Agent myself = (Agent)getBeliefbase().getBelief("my_self"
          ).getFact();
```

```
int agentno = region.getAgentIndex(myself.getAID());
String action = "";
done = (Boolean)region.content.get("done");

//wait
for(int t=0; t<8;t++) {
  if(region.isWaitRequired(myself.getAID())) { // need to
      wait
    boolean ret = region.blockAgent(myself.getAID(), 100)
        ;
    if(ret == false) {
      continue;
    }
  }
}

//action
EnvironmentInfo einfo = (EnvironmentInfo) getBeliefbase()
    .getBelief("env_info").getFact();
Location currLoc = myself.getLocation();
Location nextLoc;
//TODO done condition?
if(einfo.getManhattanDistance(currLoc, loc) == 0) {
  region.content.put("done", new Boolean(true));
  action = RequestMove.DIRECTION_NONE;
}

// wait the next
if(agentno+1 != region.getNumberOfAgents()) {
  Location nextAgentLoc = (Location)region.content.get("
      loc"+(agentno+1));
  if(nextAgentLoc == null || einfo.getManhattanDistance(
      currLoc, nextAgentLoc)>3) {
    action = RequestMove.DIRECTION_NONE;
  }
}

if(action == "") {
  //normal move
  Location leaderLoc;
  if(agentno == 0) {
    leaderLoc = loc;
  } else {
    leaderLoc = (Location)region.content.get("loc"+(
        agentno-1));
  }

  action = RequestMove.DIRECTION_NONE;
  if(leaderLoc !=  null) {
    if(agentno==0 || einfo.getManhattanDistance(leaderLoc
        , currLoc) > 1) {
      String dirs[] = einfo.getDirections(currLoc,
          leaderLoc);
      action = dirs[0];
    }
```

```
            }
          }

          //write our desired next loc
          nextLoc = einfo.calculateLocation(currLoc, action);
          region.content.put("loc"+agentno, nextLoc);

          //tell
          region.agentDone(myself.getAID());

          //make move
          move(action); // this function ends when environment
              gives reply
        }
      }
    }
```

Subtask : Wait Mine Cleaning

There is only one subtask defined for this task. The subtask *Wait Mine Cleaning* is started when the route is blocked by a mine field. In order to continue, agent has to wait until the mine field is cleaned. This subtask could call for mine cleaners. The example simulation is not interested in mine cleaning and there is no actors or facility for mine cleaning. Mine cleaning is modelled simply by expiration in the environment. The simulation assumes a mine field is cleaned after a predefined time of its exposition.

PA Physical Agent

Goal_WaitMineCleaning: This is the goal for *Wait Mine Cleaning* subtask. Table 5.2 lists the goal parameters.

Table 5.2: Wait Mine Cleaning Goal Parameters

Goal Parameters	Description	Data Type
Mine Field Region	The mine field region the agent waits for cleaning	Mine Field Id

Plan_WaitMineCleaning: This plan handles the goal *Goal_WaitMineCleaning*.

```
public class WaitMineCleaningPlan extends Plan {
  public void body() {
    Agent myself;
    registerForTickWait();
    while(true) {
      myself = (Agent)getBeliefbase().getBelief("my_self").
          getFact();
      if(myself.getVision().getMineFields() == null)
        break;
      waitForNextTick();
    }
  }
}
```

5.3.2 Team Goto Location Task

This task is one of the simplest team level tasks that is given to a group of agent. The group will go to the location given as a parameter in line formation. The leader waits other agents in order to maintain the formation. The task succeeds when the leader reaches the given location.

Applicable Actors:

Air Defense Team, Tank Team, Helicopter Team.

Task Parameters:

Table 5.3 defines the task parameters.

Table 5.3: Team Goto Location Task Parameters

Task Parameters	Description	Data Type
Location	The target point that should be reached by the team	2D Map Coordinate

Preconditions and Assumptions:

- Physical agents should have mobility.
- A route to the location should exist

Task Implementation

The team hierarchy defines the order of physical agents. The PA1, PA2, PA3 and PA4 are defined by the hierarchy and TA prepares the blackboard according to this information and delegates coordinated movement by giving them *Goal_CoordinatedGotoLocation*. The TA team agent observes all agent positions and finishes the task accordingly. Also it is TA's responsibility to check for subtask conditions.

The leader agent (PA1) tries to find an applicable route to the destination. All other agents try to follow the agent prior to itself. For example, PA2 follows PA1. If any of the agents looses mobility or killed, the gap will be filled in order.

Hierarchy

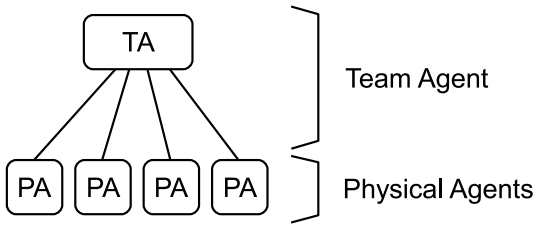

Figure 5.4: Team Goto Location Task Hierarchy

Team agent TA is responsible for delivering necessary goals to agents PA1, PA2, PA3 and PA4, and coordinating them (Figure 5.4).

Success Condition

The task is successful when the leader agent reaches to given target location. The *TaskGoal_TeamGotoLocation* goal will finish successfully.

Failure Condition

If there is no applicable route is present to reach the target destination, The *TaskGoal_TeamGotoLocation* goal will finish with failure.

Subtask Conditions

The route can be blocked temporarily by a mine field. In this circumstance, the subtask *Wait Mine Cleaning* can be activated in a new runlevel to handle the situation.

Task Goals and Plans

TA Team Agent

TaskGoal_TeamGotoLocation: Parameters of this goal is the same as the task parameters. In order to reuse from other tasks, additional parameters for blackboard definition is added and shown in Table 5.4.

Table 5.4: Team Goto Location Goal Parameters

Goal Parameters	Description	Data Type
Mine Field Region	The mine field region the agent waits for cleaning	Mine Field Id
Blackboard Name (Optional)	The name of the blackboard to be used	Blackboard Id
Blackboard Region (Optional)	The region of the blackboard to be used	Blackboard Region Id

Plan_TeamGotoLocation: This plan handles the goal *TaskGoal_TeamGotoLocation*.

```
public class TeamGotoLocationPlan extends TeamPlan {
  public void body() {
    TeamAgent myself = (TeamAgent)getBeliefbase().getBelief("
        my_self").getFact();
    BlackBoardManager bbm = (BlackBoardManager)getBeliefbase().
        getBelief("blackboardmanager").getFact();

    Location loc = (Location)getParameter("location").getValue
        ();
    String boardname = (String)getParameter("boardname").
        getValue();
    String regionname = (String)getParameter("regionname").
        getValue();
```

```
        //blackboard management
        BlackBoard board;
        if(boardname==null || boardname.equals("")) {
          boardname = bbm.generateBoardName();
          board = new BlackBoard();
          regionname = "";
          bbm.addBlackBoard(boardname, board);
        } else {
          board = bbm.getBlackBoard(boardname);
        }

        BlackBoardRegion region;
        if(regionname==null || regionname.equals("")) {
          regionname = bbm.generateRegionName();
          region = new BlackBoardRegion();
          board.regions.put(regionname, region);
        } else {
          region = board.regions.get(boardname);
        }

        //for termination
        region.content.put("done", new Boolean(false));

        registerForTickWait();
        for(AgentIdentifier taid:myself.getChildAIDs()) {
          region.addAgent(taid); //setup agent orders
        }

        //send goals to sublevel agents
        RequestGoal rg = new RequestGoal("gotolocation");
        rg.addParameter("location", loc);
        rg.addParameter("boardname", boardname);
        rg.addParameter("regionname", regionname);

        // send goal request to all children
        for(AgentIdentifier r_aid: myself.getChildAIDs()) {
          IGoal sendgoal = createGoal("sendgoal");
          sendgoal.getParameter("goal").setValue(rg);
          sendgoal.getParameter("receiver").setValue(r_aid);
          dispatchSubgoal(sendgoal);
        }

        //watch termination
        while(true) {
          Boolean b = (Boolean) region.content.get("done");
          if(b != null && b == true) {
            break;
          }
          waitForNextTick();
        }
      }
    }
```

PA1, PA2, PA3 and PA4 Physical Agents

Goal_CoordinatedGotoLocation: This goal is sent from *Plan_TeamGotoLocation* to all physical agents with the same blackboard and region information. All agents runs their plan in coordinated with the blackboard.

Plan_CoordinatedGotoLocation: This plan handles the goal *Goal_CoordinatedGoto-Location.*

```
public class CoordinatedGotoLocationPlan extends Plan {
  public void body() {
    Location loc = (Location)getParameter("location").getValue
        ();
    String boardname = (String)getParameter("boardname").
        getValue();
    String regionname = (String)getParameter("regionname").
        getValue();

    if(boardname == null || boardname.equals("")) {
      logger.severe("boardname is empty");
      fail();
    }

    BlackBoardRegion region = BlackBoardManager.getInstance().
        getBlackBoard(boardname).regions.get(regionname);

    //termination
    Boolean done = (Boolean)region.content.get("done");

    //loop
    while(!done) {
      Agent myself = (Agent)getBeliefbase().getBelief("my_self"
          ).getFact();
      int agentno = region.getAgentIndex(myself.getAID());
      String action = "";
      done = (Boolean)region.content.get("done");

      //wait
      for(int t=0; t<8;t++) {
        if(region.isWaitRequired(myself.getAID())) { // need to
            wait
          boolean ret = region.blockAgent(myself.getAID(), 100)
              ;
          if(ret == false) {
            continue;
          }
        }
      }

      //action
      EnvironmentInfo einfo = (EnvironmentInfo) getBeliefbase()
          .getBelief("env_info").getFact();
      Location currLoc = myself.getLocation();
      Location nextLoc;
```

```
        //TODO done condition?
        if(einfo.getManhattanDistance(currLoc, loc) == 0) {
          region.content.put("done", new Boolean(true));
          action = RequestMove.DIRECTION_NONE;
        }

        // wait the next
        if(agentno+1 != region.getNumberOfAgents()) {
          Location nextAgentLoc = (Location)region.content.get("
              loc"+(agentno+1));
          if(nextAgentLoc == null || einfo.getManhattanDistance(
              currLoc, nextAgentLoc)>3) {
            action = RequestMove.DIRECTION_NONE;
          }
        }

        if(action == "") {
          //normal move
          Location leaderLoc;
          if(agentno == 0) {
            leaderLoc = loc;
          } else {
            leaderLoc = (Location)region.content.get("loc"+(
                agentno-1));
          }

          action = RequestMove.DIRECTION_NONE;
          if(leaderLoc !=  null) {
            if(agentno==0 || einfo.getManhattanDistance(leaderLoc
                , currLoc) > 1) {
              String dirs[] = einfo.getDirections(currLoc,
                  leaderLoc);
              action = dirs[0];
            }
          }
        }

        //write our desired next loc
        nextLoc = einfo.calculateLocation(currLoc, action);
        region.content.put("loc"+agentno, nextLoc);

        //tell
        region.agentDone(myself.getAID());

        //make move
        move(action); // this function ends when environment
            gives reply
      }
    }
  }
```

Subtask : Wait Mine Cleaning

There is only one subtask defined for this task. The subtask *Wait Mine Cleaning* is started when the route is blocked by a mine field. In order to continue, team have to until the mine field is cleaned. This subtask can call for mine cleaners. The example simulation is not interested in mine cleaning and there is no actors or facility for mine cleaning. Mine cleaning is modelled simply by expiration. The simulation assumes a mine field is cleaned after a predefined time of its exposition.

TA agent reuses *Goal_WaitMineCleaning* from *Goto Location Task* and sends to all physical agents. When one of the agents succeeds, TA finishes the subtask with success.

TA Team Agent

Goal_TeamWaitMineCleaning: This goal represents the subtask and parameters are shown in Table 5.5.

Table 5.5: Team Wait Mine Cleaning Goal Parameters

Goal Parameters	Description	Data Type
Mine Field Region	The mine field region the agent waits for cleaning	Mine Field Id

Plan_TeamWaitMineCleaning: This plan handles *Goal_TeamWaitMineCleaning*.

```
public class TeamWaitMineCleaningPlan extends TeamPlan {
  public void body() {
    Agent myself;
    registerForTickWait();
    Vector<Goal> goals = new Vector<Goal>();

    RequestGoal rg = new RequestGoal("waitminecleaning");
    // send goal request to all children
    for(AgentIdentifier r_aid: myself.getChildAIDs()) {
      Goal goal = sendGoal(r_aid, rg);
      goals.add(goal);
    }
    boolean bwait = true;
    while(bwait) {
      waitForNextTick();
```

```
            for(Goal g : goals.toArray()) {
              if(g.isfinished())
                bwait = false;
            }
          }
        }
      }
```

5.3.3 Team Air Defense Task

This task is given to a team of air defense vehicles to cover and defend an area cooperatively. Position of the team members and weapon range of the vehicles define the area.

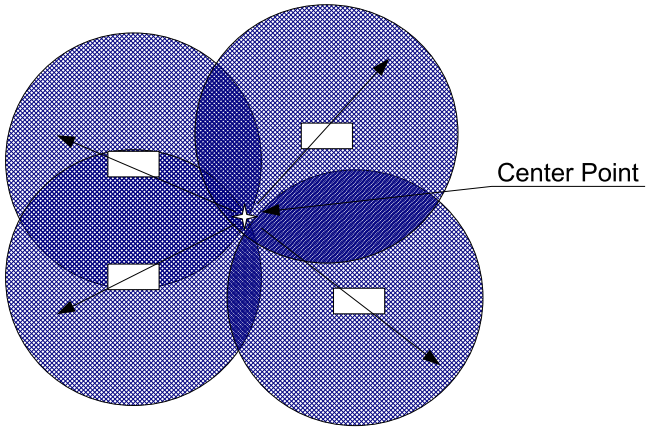

Figure 5.5: Team Air Defense

As can be seen on Figure 5.5, covered area is the sum of all circles where a circle is defined by the agent position and weapon range. Center point of the agents are the mean value of their positions. Each responsibility direction of agents is defined as the opposite vector to the center point. The responsibility direction is used in target assignment.

The task will end when the maximum execution time is reached. If the maximum execution is zero, the task will work forever unless a scenario condition holds true.

Applicable Actors:

Air Defense Team.

Task Parameters:

Table 5.6 defines the task parameters.

Table 5.6: Team Air Defense Task Parameters

Task Parameters	Description	Data Type
Weapon Mode	States whether agents can engage enemy	Boolean
Maximum Time	States the maximum execution time of the task	Seconds

Preconditions and Assumptions:

- Physical agents should have the ability to fire their weapons.

Task Implementation

The team hierarchy defines the order of physical agents. The PA1, PA2, PA3 and PA4 are defined by the hierarchy. The team agent prepares the blackboard and delegates the coordinated air defense goals to physical agents. TA observes the time to end the task successfully.

Hierarchy

Team agent TA in Figure 5.6 is responsible for ensuring agent hierarchy is as above and

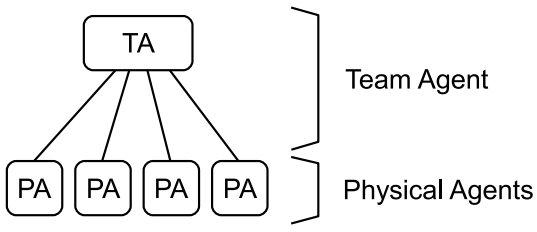

Figure 5.6: Team Air Defense Hierarchy

delivering necessary goal to agents PA1, PA2, PA3 and PA4.

Success Condition

The task will end when the maximum execution time is reached. The *TaskGoal_TeamAir-Defense* goal will finish successfully.

Failure Condition

There is no explicit failure condition defined. The *TaskGoal_TeamAirDefense* goal will not finish with failure. It will try to continue until all agents are destroyed.

Subtask Conditions

There is no need for subtask condition.

Task Goals and Plans

TA Team Agent

TaskGoal_TeamAirDefense: This goal represents the task. Goal parameters are same as the task parameters.

Plan_TeamAirDefense: This plan handles the goal *TaskGoal_TeamAirDefense*.

Subtask Goals and Plans

There is no subtask defined for this task.

5.3.4 Air Defense Task

Air Defense task is given to a single physical agent to watch for enemy and engage if it is in weapon range. The agent is stationary in this task.

Applicable Actors:

Single Air Defense Vehicle.

Task Parameters:

Table 5.7 defines the task parameters.

Table 5.7: Air Defense Task Parameters

Task Parameters	Description	Data Type
Weapon Mode	States whether agents can engage enemy	Boolean
Maximum Time	States the maximum execution time of the task	Seconds

Preconditions and Assumptions:

- Physical agents should have the ability to fire their weapons.

5.3.5 Team Change Positions Task

This task is given to a team of air defense vehicles to change positions cooperatively. How team changes positions are defined with task parameters. Active moving agents can be limited during the task. The change list also specifies the order of change movements.

Applicable Actors:

Air Defense Team.

Task Parameters:

Table 5.8 defines the task parameters.

Preconditions and Assumptions:

- Physical agents should have the ability to fire their weapons.

5.3.6 Team Defensive Move Task

Team Defensive Move task is a high level task that involves coordinated moving and collective defense of the area covered by the air defense physical agents. The air defense team accompanies the given escorted agent as parameters with following them.

Table 5.8: Team Change Positions Task Parameters

Task Parameters	Description	Data Type
Change List	This list defines which agent should change its current position to the new position. AgentNo is the order of the physical agent in the hierarchy.	<AgentNo, Location >list
Weapon Mode	States whether agents can engage enemy	Boolean
Maximum Moveable Agent Limit	Defines how many agents are allowed to move at a time. Agents can not use their weapon while moving.	Integer greater than 0

Applicable Actors:

Air Defense Team

Task Parameters:

Table 5.9 defines the task parameters.

Preconditions and Assumptions:

- Physical agents should have mobility.
- Physical agents should have the ability to fire their weapons.
- When none of the escorted agents is seen, team holds last position.

Table 5.9: Team Defensive Move Task Parameters

Task Parameters	Description	Data Type
Route	The route to be followed automatically based on position of escorted agents. The route is used in selecting appropriate defensive position.	Location list
Defensive Position Pairs	This array contains location pairs. Normally air defense is carried out by 4 agents and 2 item from this list appropriate for coverage. The list contains position pairs that will be chosen when searching for good defensive positions	Location pair list
Advance Type	Defines how agents will advance to their new defensive positions	”bounds” or ”follows”.
Maximum Moveable Agent Limit	Defines how many agents are allowed to move at a time. Agents can not use their weapon while moving.	Integer greater than 0
Escorted Agents	This list defines the escorted agents. Based on their positions, air defense team tries to cover them as they move.	AgentId list

5.4 Scenario File

Below listing is the scenario file in XML file format that is used to define the scenario discussed in section 5.1. Scenario file starts by defining the environment and agents. It also contains task flows utilizing the tasks modelled in 5.3.

```
   <?xml version="1.0" encoding="UTF-8"?>
<tns:scenario xmlns:tns="http://www.yukselen.web.tr/y_thesis/
    scenario" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.yukselen.web.tr/y_thesis/scenario
    _scenario.xsd_">
  <tns:environment size-x="32" size-y="32">
    <tns:obstacle-map>map1.png</tns:obstacle-map>
```

```
</tns:environment>
<tns:agents>
  <tns:team-agent name="team1" side-id="1">
    <tns:configuration-file>y_thesis.agents.teamagents.
        DefenseTeam</tns:configuration-file>
    <tns:agent name="air1" side-id="1">
      <tns:configuration-file>y_thesis.agents.teamagents.
          Cooperative</tns:configuration-file>
      <tns:location x="24" y="5"/>
      <tns:icon-file>AirDefense.png</tns:icon-file>
    </tns:agent>
    <tns:agent name="air2" side-id="1">
      <tns:configuration-file>y_thesis.agents.teamagents.
          Cooperative</tns:configuration-file>
      <tns:location x="27" y="5" />
      <tns:icon-file>AirDefense.png</tns:icon-file>
    </tns:agent>
    <tns:agent name="air3" side-id="1">
      <tns:configuration-file>y_thesis.agents.teamagents.
          Cooperative</tns:configuration-file>
      <tns:location x="24" y="2" />
      <tns:icon-file>AirDefense.png</tns:icon-file>
    </tns:agent>
    <tns:agent name="air4" side-id="1">
      <tns:configuration-file>y_thesis.agents.teamagents.
          Cooperative</tns:configuration-file>
      <tns:location x="27" y="2" />
      <tns:icon-file>AirDefense.png</tns:icon-file>
    </tns:agent>
  </tns:team-agent>

  <tns:agent name="tank1" side-id="1">
    <tns:configuration-file>y_thesis.agents.teamagents.
        Cooperative</tns:configuration-file>
    <tns:location x="4" y="5" />
    <tns:icon-file>Tank.png</tns:icon-file>
  </tns:agent>
  <tns:agent name="tank2" side-id="1">
    <tns:configuration-file>y_thesis.agents.teamagents.
        Cooperative</tns:configuration-file>
    <tns:location x="6" y="4" />
    <tns:icon-file>Tank.png</tns:icon-file>
  </tns:agent>
  <tns:agent name="tank3" side-id="1">
    <tns:configuration-file>y_thesis.agents.teamagents.
        Cooperative</tns:configuration-file>
    <tns:location x="3" y="3" />
    <tns:icon-file>Tank.png</tns:icon-file>
  </tns:agent>
  <tns:agent name="tank4" side-id="1">
    <tns:configuration-file>y_thesis.agents.teamagents.
        Cooperative</tns:configuration-file>
    <tns:location x="5" y="2" />
    <tns:icon-file>Tank.png</tns:icon-file>
  </tns:agent>
```

```
</tns:agents>

<tns:flows>
  <tns:flow flow-name="flow1" assigned-agent="team1">
    <tns:flow-element>
      <tns:goal goalname="teamgotolocation">
        <tns:goalparam paramname="location">
          <tns:value>new Location(5,5)</tns:value>
        </tns:goalparam>
      </tns:goal>
    </tns:flow-element>
    <tns:flow-element>
      <tns:operator operation="deaggregate" operator-id="op1" />
      <tns:flow flow-name="flow11" assigned-agent="air1">
        <tns:flow-element>
          <tns:goal goalname="gotolocation">
            <tns:goalparam paramname="location">
              <tns:value>new Location(3,3)</tns:value>
            </tns:goalparam>
          </tns:goal>
        </tns:flow-element>
        <tns:flow-element>
          <tns:operator operation="aggregate" operator-id="aggop1
              " />
        </tns:flow-element>
      </tns:flow>
      <!-- 3 more flow for each agent to cover tank team -->
    </tns:flow-element>
  </tns:flow>

  <tns:flow flow-name="flow2" assigned-agent="team1" after-
      operator-id="aggop1">
    <tns:flow-element>
      <tns:goal goalname="teamdefensivemove">
        <tns:goalparam paramname="route">
          <tns:value>new Location(5,5)</tns:value>
          <tns:value>new Location(12,13)</tns:value>
          <tns:value>new Location(25,16)</tns:value>
          <tns:value>new Location(22,30)</tns:value>
        </tns:goalparam>
        <tns:goalparam paramname="defensivePositionPairs">
          <tns:value>new LocationPair(new Location(3,3), new
              Location(5,3))</tns:value>
          <tns:value>new LocationPair(new Location(6,4), new
              Location(9,5))</tns:value>
    <!-- DELETED : to conserve space -->
        </tns:goalparam>
        <tns:goalparam paramname="advanceType">
          <tns:value>bounds</tns:value>
        </tns:goalparam>
        <tns:goalparam paramname="maximumMoveableAgentLimit">
          <tns:value>2</tns:value>
        </tns:goalparam>
        <tns:goalparam paramname="escortedAgents">
          <tns:value>tank1</tns:value>
```

```
            <tns:value>tank2</tns:value>
            <tns:value>tank3</tns:value>
            <tns:value>tank4</tns:value>
          </tns:goalparam>
        </tns:goal>
      </tns:flow-element>
      <tns:flow-element>
        <tns:goal goalname="teamairdefense">
          <tns:goalparam paramname="weaponMode">
            <tns:value>true</tns:value>
          </tns:goalparam>
          <tns:goalparam paramname="maksimumTime">
            <tns:value>1000</tns:value>
          </tns:goalparam>
        </tns:goal>
      </tns:flow-element>
    </tns:flow>
  </tns:flows>
</tns:scenario>
```

5.5 Simulation Run

This section tries to narrate a sample simulation run. In order to visualize the sample run, Figure 5.7 is given with colored overlays. Blue overlays are for air defense team and green overlays are for tank team. Red is used for enemy units on the north of the map. Orange box on the south is used to visualize a mine field.

Throughout the simulation run, enemy is constantly patrolling the are designated with red arrow back and forth. The enemy was programmed to attack only to threats that are in fire range.

Simulation of friendly units are controlled with a scenario in two phases.

In the first phase, tank team is positioned to a location (green 1) to wait for air defense team to arrive. Air defense team (near blue 1) wants to travel to the position to rendezvous with the tank team. Air defense starts to move west in column formation with a *team goto location* task. In the middle of their task, they encounter a mine field (near blue 2). How to handle this situation is encoded in the task implementation as a subtask. Exposition of the mine field triggers the subtask condition. Air defense team starts to execute the *wait mine cleaning* subtask to handle the situation. Implementation of the subtask is simply waiting until the mine field is removed from the environment. As stated previously, environment vanishes the

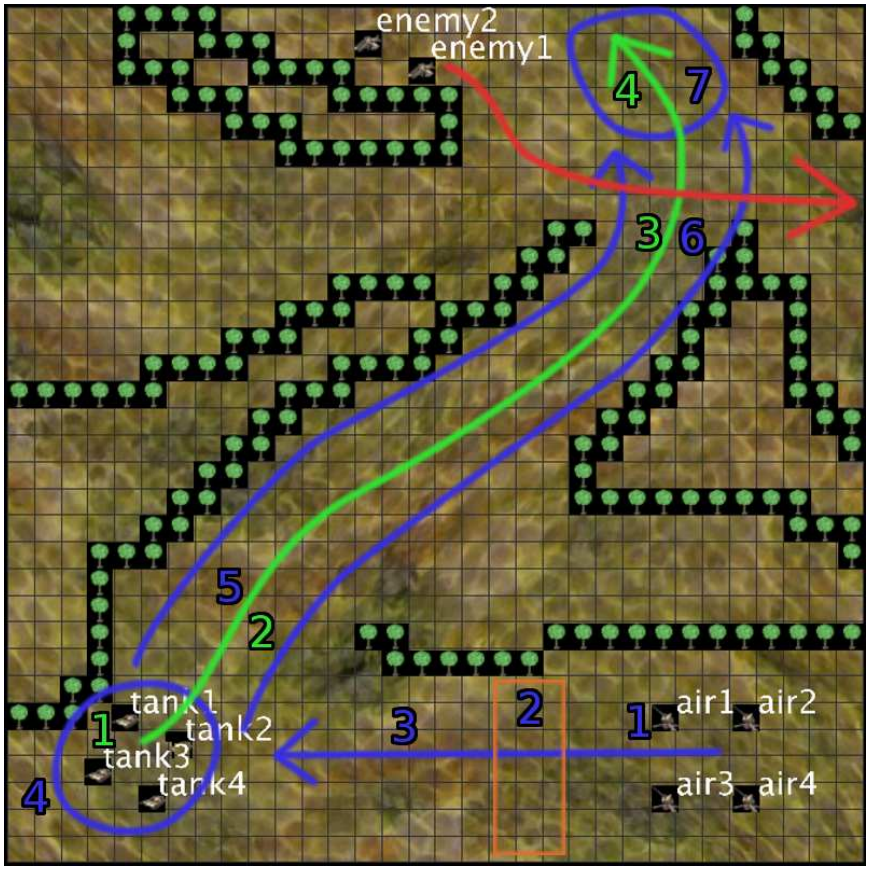

Figure 5.7: Sample Simulation Run

mine field after its predefined time passes. Removal of the minefield from the environment is enough to finish the running subtask. Air defense team continues its task execution and follows the route again (near blue 3). After reaching rendezvous point, air defense vehicles scatter around (blue 4) to take cover of the tank team. Afterwards, air defense team starts to execute *team defensive move* task monitoring the movement of 4 tanks. Start of the *team defensive move* task ends the phase 1 of the scenario.

Phase 2 of the scenario begins by movement of tank team. Tank team tries to follow the green line in the valley that reaches to north. Region near green number 4 is the target point to reach by the tank team. Tank team does not move at full speed because air defense team need to follow and take cover of the tanks easily. Movement of tank team is monitored by *team defensive move* task. This task tries to keep tanks in the coverage for air attacks. In order to accomplish this, air defense vehicles needs to change their location. The rules of the behaviour is encoded in the task definition. Along the way, air defense team changes location

near the tank team whenever tanks crosses the safe region covered by the air defense team. Tank team moves at a slow constant speed in this region.

Opening of the valley to the north has clear visibility both for enemy helicopters and friendly vehicles. At this region near green number 3, tank team encounters enemy helicopters. Enemy helicopters do not stop their patrol when they encounter any vehicles. They open fire to each vehicle in their attack range while they are continuing their patrol task. Air defense team is still executing *team defensive move* task around blue number 4 on figure. Each vehicle will choose a suitable target based on their attack range in coordination and respond to enemy presence.

After enemy contact, one of the sides eventually destroyed. Air defense team has more fire power than enemy helicopters and can survive with less casualty. However the tanks are totally vulnerable to air attack and success of this scenario is subjectively assessed with vehicle casualty numbers. Air defense vehicles reorganizes their position based on their casualties. They continue *team defensive move* task until reaching region highlighted by blue number 7 on the figure. Alive tank team actors reaches to north and take up position around green number 4. This task was the last task defined in the scenario.

CHAPTER 6

CONCLUSION

Modelling and implementing simulation applications involves software engineering difficulties that should be solved. Team-oriented modelling of multiagent systems increases the complexity of the architecture. With the framework presented in this thesis, we try to introduce facilities that will ease application of team-oriented simulations in multiagent systems.

Proposed solution is generic enough to be applied to any team-oriented problem with well defined task coordination that is definable by team agent plans. Several types of global synchronization issues are handled with facilities such as blackboard and runlevel management in a team. Inter team coordination is addressed effectively by introducing distributed condition evaluation.

Scenario definition and management with conditions presents a high level control mechanism. This control mechanism enables to glue the missing gap between top down problem solution and bottom up agent implementation in the framework.

As the implementation is based on extending Jadex, which is an open source mature platform for BDI agents, application of the framework will ease the development process and yet leave room for extensibility.

Application of the framework to cases with higher social interactivity can be studied as a future work. Application of the framework to domains with loose agent cooperation can bring out the need of additional facilities to be served by the framework.

The process of scenario definition is the most labor intensive task to perform. Additional graphical tools and visualization techniques can ease this process.

Scenario definition has also an open problem for verification. Although giving semantic meaning to the scenario definition is a hard problem, it can enable high level scenario planners to be developed. Syntactic verification of the scenario definition can help detecting definition errors prior to simulation run. This verification

REFERENCES

[1] D. Ancona and V. Mascardi. Coo-bdi: Extending the bdi model with cooperativity, 2003.

[2] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade: a fipa2000 compliant agent development environment. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 216–217, New York, NY, USA, 2001. ACM.

[3] R. H. Bordini, J. F. Hbner, and et al. *Jason: A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net*, release version 0.7 edition edition, August 2005.

[4] Michael E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge MA, 1987.

[5] Mark D'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dmars architecture: A specification of the distributed multi-agent reasoning system. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):5–53, 2004.

[6] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *AAAI*, pages 677–682, 1987.

[7] Michael P. Georgeff and Anand S. Rao. A profile of the australian artificial intelligence institute. *IEEE Expert*, 11(6):89–92, 1996.

[8] Robert Gray, David Kotz, George Cybenko, and Daniela Rus. Agent Tcl. In William Cockayne and Michael Zyda, editors, *Mobile Agents: Explanations and Examples*, chapter 4, pages 58–95. Manning Publishing, 1997. Imprints by Manning Publishing and Prentice Hall.

[9] Koen V. Hindriks and Frank S. De Boer. Agent programming in 3apl. *AAMAS Journal*, 2:357–401, 1999.

[10] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. In *ATAL '00: Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages*, pages 228–243, London, UK, 2001. Springer-Verlag.

[11] N. Howden, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - summary of an agent infrastructure. In *In Proceedings of the 5th ACM International Conference on Autonomous Agents*, 2001.

[12] D B Lange and Mitsuru Oshima. Programming and deploying java mobile agents with aglets, 1998.

[13] Jaeho Lee and Edmund H. Durfee. Structured circuit semantics for reactive plan execution systems. In *AAAI'94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, pages 1232–1237, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[14] Agent Oriented Software Pty. Ltd. Jack Intelligent Agents. http://agent-software.com.au/jack.html.

[15] P. Marrow, E. Bonsma, F. Wang, and C. Hoile. Diet — a scalable, robust and adaptable multi-agent platform for information management. *BT Technology Journal*, 21(4):130–137, 2003.

[16] Viviana Mascardi, Daniela Demergasso, and Davide Ancona. Languages for programming bdi-style agents: an overview. In Flavio Corradini, Flavio De Paoli, Emanuela Merelli, and Andrea Omicini, editors, *WOA*, pages 9–15. Pitagora Editrice Bologna, 2005.

[17] David Mcilroy and Clinton Heinze. Air combat tactics implementation in the smart whole air mission model (swarmm). In *In Proceedings of the First International SimTecT Conference*, 1996.

[18] Karen L. Myers and David E. Wilkins. *The Act Formalism, version 2.2*. Artificial Intelligence Center, SRI International, Menlo Park, CA, Sep 1997.

[19] Holger Peine and Torsten Stolpmann. The architecture of the ara platform for mobile agents. In *MA '97: Proceedings of the First International Workshop on Mobile Agents*, pages 50–61. Springer Verlag, 1997.

[20] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A bdi reasoning engine. In J. Dix R. Bordini, M. Dastani and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, pages 149–174. Springer Science+Business Media Inc., USA, 9 2005. Book chapter.

[21] A. S. Rao and M. P. Georgeff. BDI-agents: From theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[22] Anand S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[23] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a bdi-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[24] Birna van Riemsdijk, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in dribble: from beliefs to goals using plans. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 393–400, New York, NY, USA, 2003. ACM.

[25] lvaro F. Moreira, Renata Vieira, Rafael H. Bordini, and Centro De Ciłncias Exatas. Extending the operational semantics of a bdi agent-oriented programming language for introducing speech-act based communication. In *In Proc. of DALT*, pages 45–59. Springer-Verlag, 2002.