2000

ConstraintAgentSpeak: an extension of the Agent Specification Language - AgentSpeak(L)

Boon H. Ooi
University of Wollongong
NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
ConstraintAgentSpeak: An Extension of the Agent Specification Language – AgentSpeak(L)

A thesis submitted in fulfillment of the requirements for the award of the degree

HONOURS MASTERS

from

UNIVERSITY OF WOLLONGONG

by

Boon H. Ooi
B. Com.

Department of Information Systems
2000
Declaration

I, Boon H. Ooi. Declare that this thesis, submitted in fulfillment of the requirements for the award of Honours Masters, in the Department of Information Systems, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

Boon H. Ooi

30-March-2000
ConstraintAgentSpeak : An Extension of the Agent Specification Language – AgentSpeak(L)
Conference Presented

Parts of this thesis have been presented in the following conference:


Preface

I am indebted to Dr. Aditya K. Ghose who was my principal supervisor. Through his never-ending enthusiasm, he has raised my passion for AI and has kept this fire burning ever-since. His devotion to AI research has always been a role model and admiration to me.

To Assoc. Prof Joseph G. Davis, many thanks for being such a kind adviser and supervisor to me all the time, and from whom I have learnt some great thing about life.

I am especially grateful to the financial support that Dr Ghose and Assoc. Prof. Davis have given to me in order for me to complete this course. It has been a great honour to be able to learn under the supervision of the two wise gentlemen.

To Prof. Graham Winley, I am thankful for the time to see me and to provide useful insight into some of my research work which proved to be so valuable. His humour and concern have always been a source of encouragement and inspiration for me throughout the course.

To David Dodds, a cool and very competent gentleman who has provided me with heaps of valuable technical insight. His prompt and kind assistance when I was in trouble with the computer in the DSL has always impressed me without fail.

To Joshua Fan, I am truly grateful for enlighten me with some great tips and insight to the crucial technical details on Java.

To all other ‘comrades’ in DSL, their company has proven to be a life enrichment experience that I will treasure forever as all my other valuable endeavours.
Abstract

Intelligent agent technology has been recognized as one of the prominent research areas. It is viewed as a very promising area in terms of its innovativeness and its potential impacts on the development of information technology. The demand for more flexible and intelligent computing solutions in today's computing environment has strongly supported the above statement and opened up more opportunities for the application of agent-based technology.

The design of complex agent-based systems requires the use of expressive high-level specification languages which eventually translate into efficient implementations. This thesis suggests that constraint-based agent specification is one of the approaches that meet these requirements. To support this assertion, this thesis presents an augmentation of the BDI agent programming language AgentSpeak(L) [Rao, 1996] with constraints and describes an implementation of an interpreter for the proposed agent specification language. The proposed language, called ConstraintAgentSpeak, improves over AgentSpeak(L) in a manner parallel to the gains achieved by integrating constraints in a logic programming framework to obtain constraint logic programming [Jaffar, 1986], in terms of both expressivity and efficiency. A preliminary observation on BDI agent architecture and constrain-based reasoning is given in chapter 2. Other chapters of this thesis present the following discussions on ConstraintAgentSpeak.
1. a detail description of *ConstraintAgentSpeak* in terms of its syntax and operational semantics.

2. an explanation on the implementation of the proposed language interpreter.

3. sample agent-based applications that can be programs using the new agent specification language.
Contents

1. Introduction

1.1 Overview of Agent-Based Research .......................................................... 1
1.2 Problem Identification ............................................................................. 2
1.3 Research Objectives and Proposed Solution .......................................... 6
1.4 Overview of Proposed Solution ............................................................... 7
1.5 Organisation of Thesis ........................................................................... 9

2. Background

2.1 Today's computing dilemma................................................................. 11
2.2 An Abstract of Agent-Based Technology .............................................. 11
2.3 Development of Software Agent and Agent Classification ................. 13
  2.3.1 Deliberative Agent ........................................................................ 14
  2.3.2 Reactive Agent ............................................................................. 14
  2.3.3 Interacting Agent ......................................................................... 16
  2.3.4 Improvised Approach - A Hybrid Agent Architecture ..................... 17
  2.3.5 Summary ..................................................................................... 17
2.4 BDI Architecture .................................................................................. 18
  2.4.1 Belief, Desire and Intentions ......................................................... 18
  2.4.2 IRMA architecture : Bratman et al. ............................................. 19
  2.4.3 Formal BDI Model : Rao and Georgeff ......................................... 20
  2.4.4 Interpreter for A BDI Agent : Rao and Georgeff ......................... 22
2.5 Agent Programming Language ............................................................. 24
  2.5.1 AgentSpeak(L) : A BDI Agent Programming Language ............... 24
    2.5.1.1 Introduction ...................................................................... 24
    2.5.1.2 Agent Programs ............................................................... 25
    2.5.1.3 Agent Operation ............................................................. 27
2.6 Constraint ............................................................................................ 30
  2.6.1 Constraint Satisfaction .................................................................. 32
2.6.1.1 Solving Technique for CSP .........................................35
2.6.1.2 Consistency Technique .................................................37
2.6.1.3 Node Consistency ..........................................................37
2.6.1.4 Arc Consistency ............................................................38

2.6.2 Constraint Logic Programmig ............................................38
  2.6.2.1 Rules (User-Defined Constraints) .......................................40
  2.6.2.2 Basic Terminology and Example of CLP ...........................41
  2.6.2.3 Goal Evaluation ............................................................42
  2.6.2.4 Constraint Solver ..........................................................43
  2.6.2.5 The CLP Scheme and Motivation ....................................44

2.7 Conclusion ..................................................................................45

3. ConstraintAgentSpeak

3.1 Syntax .........................................................................................46
3.2 Operational Semantics ................................................................52
  3.2.1 Operational Semantics for Constraint Solving ....................52
    3.2.1.1 Choice of Reduction Rule ...........................................53
    3.2.1.2 Renaming of Rule .........................................................55
  3.2.2 Operational Semantics for ConstraintAgentSpeak Cycle of Operation ..........................................................55
    3.2.2.1 Inference Procedure .......................................................58
      3.2.2.1.1 Generate Relevant Plan Set (genrelplan) ...............59
      3.2.2.1.2 Generate Applicable Plan (genapplplan) .............60
    3.2.2.2 Agent-Execution (Intention-Execution) .......................62
  3.3 Proof Theory .............................................................................67
  3.4 Conclusion .................................................................................69

4. Implementing AgentSpeak(L) + C

4.1 ConstraintAgentSpeak’s Architecture .....................................71
4.2 Implementation ...........................................................................76
6. Conclusion
6.1 Overview of Research.................................................................123
6.2 Contribution..................................................................................124
  6.2.1 Contribution to Research..........................................................124
  6.2.2 Contribution to Practitioners......................................................125
6.3 Limitation.........................................................................................126
6.4 Future Research...............................................................................127

Appendix A.
A.1 Class diagram for thread managers of
   ConstraintAgentSpeak agent............................................................129
A.2 Class diagram for generators of relevant and
   applicable plan................................................................................130
A.3 Class diagram for base beliefs of
   ConstraintAgentSpeak agent............................................................131
A.4 Class diagram for FD constraint solver of
   ConstraintAgentSpeak agent............................................................132
A.5 Class diagram for constraint belief of ConstraintAgentSpeak agent.....133

Appendix B.
B.1 Automaton for constructing base belief..........................................134
B.2 Automaton for constructing constraint belief
   (FD constraint domain)......................................................................134
B.3 Automaton for constructing constraint belief with linear equations
   or non-linear equations (real arithmetic domain).................................135
B.4 Automaton for constructing agent plan..........................................135

Appendix C.
C.1 Java source codes for thread managers of
   ConstraintAgentSpeak agent............................................................137
C.2 Java source codes for trigger event, plan, the generator of relevant/
   applicable plan and intention of an ConstraintAgentSpeak agent......148
C.3 Java source codes for base belief of

ConstraintAgentSpeak agent..........................................................189

C.4 Java source codes for CustSolutionManager.java

- a customised solution manager that handles solution set generated
  from constraint processing..........................................................196

C.5 Java source codes for lexical analysis, syntax analysis and other
  supporting functions within the BDI agent framework....................201

References......................................................................................236
Chapter 1

Introduction and Motivation

1.1 Overview of Agent-Based Research

Intelligent agent technology is one of the prominent research areas that commands a high level of interest and generates a substantial level of activity. Both the industry as well as research community have viewed it as a very promising area in terms of its innovativeness and its potential to revolutionise the many facets in today’s information technology development. Conceived in the 70’s, under the rubric of Distributed Artificial Intelligence (DAI) and the Actor model [Hewitt, 1977] [Agha, 1986] [Agha, 1988], the notion of agent has since then developed into a major international research area and numerous agent-based applications have been deployed successfully. The latest milestone of such implementation achievement is the Remote Agent Experiment\(^1\) (RAX) carried out by NASA of US which indicates that agent-based technology has even found itself a place in the realm of space research.

As the research activities in this area grow rapidly, the technology involved in this area has also evolved. Research on agent-based technology has become so diversified that any effort trying to group and categorise them is not an easy task. On an overall basis, agent-based research can be classified under three main broad categories: agent theories, agent architectures, and languages. Agent theories are concerned with providing formal account of agent behavior and structure. The logic-based formalism used here will serve as a tool for explanation, analysis and even the specification of an agent.

\(^1\) RAX is available at the URL: http://rax.arc.nasa.gov/
Currently the most popular basic theoretical framework appears to be the belief-desire-intention (BDI) model proposed by [Bratman,1987a][Bratman,1987b][Cohen,1990][Rao,1991]. Agent architectures deal with issues pertaining to the planning and behavioral control of an agent. Plan execution to achieve the desired behavior is the main focus of this research stream is viewed as critical to the effectiveness as well as efficiency of agent execution. Agent languages are referred to agent specification languages, which provide environment for creating agents' programs. As a considerably young field in the area of agent research [Wooldridge, 1995], there is still a vast terrain available for research exploration. It is the intention of this thesis to present the work done on a small section of agent languages. This thesis looks into the specification of agent using an agent specification language and also concentrates on improvisation that can be incorporated into the language so as to improve its flexibility as well as applicability.

1.2 Problem Identification

The focus of the work presented here is on constraint-based BDI agent whereby an independent BDI agent is considered to have its own mental components that directly influence its behavior or determine its course of actions. Thus BDI-based agents are endowed with intelligent reasoning capability which enables them to act or behave autonomously. Under such context of agent behavior, there are two basic inadequacies that can be viewed as the hindrance to more efficient operation as well as a wider scope of agent applicability.

First, the agent's ability to deliberate and behave accordingly does not guarantee that a BDI agent will be able to react with sufficient efficiency so as to tackle a critical situation at hand. This is particularly true when the agent is situated in a time-critical application
environment. What is lacking here is some form of efficient behavior that would make an agent become more ‘watchful’ and ‘attentive’ to some of its surrounding stimuli, recognise these stimuli, and react appropriately.

Besides the issue of time criticalness reactive behavior, scope of applicability is the second consideration that needs extensive attention. As a major part of the research effort is predominantly focused on work pertaining to theoretical framework, agent modeling, agent architecture, agent programming or agent implementation, there is a lack of attention given to the small area on agent applicability. The term agent applicability can be described here as the suitability of a particular type of agent (e.g. BDI agent) to be widely deployed in various application environments with only minor changes in the agent specification. Suitability in this sense depicts how efficient and how flexible agent programming constructs founded on a particular agent framework can be easily applied to various application settings, to encode or to provide more expressive knowledge representation, and subsequently enable the application of more efficient processing techniques. In short, it would be more useful for an agent framework to adopt certain knowledge representation approach and move on to exploit the efficient, specialised solving techniques available based on this approach. A good example would be the use of constraint-based representation to model a variety of scheduling problems and apply the specialised constraint solving technique to generate the solution set. In most agent-based applications, although possibly based on the same agent framework, customisation (via procedural programming languages such as C++, Java) is used to provide the necessary flexibility and features of an agent. Such development strategy lacks the basic, standard agent programming constructs and thus loses the re-useability of the particular
agent framework. For the similar framework to be deployed in different application set­ups, low-level customisation and modification will need to be applied, sometimes extensively to cater for different environmental criteria or elements as there are no standard structures that can be easily adapted to provide for such differing requirements. Thus it appears that there is a need for standard agent programming constructs. Standard programming constructs can provide sufficient situation specific adaptation (such as using situation specific definition for components of agent structure) without the need to make any sacrifice on framework standardisation or to make extensive alteration to the framework itself. The existence of such standardisation will not only enable the creation of an agent program to be simplified but at the same time, it will also indirectly promote the deployment of agent-based technologies and declarative programming to more commercial environments which currently are dominated by standard procedural application software. The lack of standard programming constructs or tools not only impedes the growth of agent programming in the commercial environment but also creates resistance among organisations in the industry from accepting agent-based approach for their computing needs. A main reason for such discouraging sentiment is that most agent-based approaches are primarily logic-based. Most developers are not very conversant with. As a result, agent-based solutions are generally closer to customised software approach in procedural programming. This rules out many of the benefits of reusability despite the significant effort involved. Development effort invested is hardly portable and hence development cost becomes considerably high.

Besides the technical issues involved, there are also other management considerations that cast doubts on existing agent-based technology. More standard or consistent
programming constructs that are open to adaptation do not guarantee that it will address all the problems foreseen in an agent-based application implementation. However it does resolve the problem on major issues such as adaptability, easy reusability or portability. An excellent example that has addressed the above consideration is the IBM \(^2\)Aglet agent-programming environment or Aglets Software Development Kit (ASDK).

In this particular example of IBM’s ASDK, the aglet represents the next leap forward in the evolution of executable content on the Internet, introducing program codes that can be transported along with state information. Aglets are Java objects that can move from one host on the Internet to another. That is, an aglet that executes on one host can suddenly halt execution, dispatch itself to a remote host, and resume execution there. When the aglet moves, it takes along its program codes as well as its data. Thus the Aglets Software Development Kit is a standard environment for programming mobile Internet agents in Java. Though its application is more oriented towards the creation of mobile agent transportable over the Internet, it does highlight the significance of easy adaptation of the same structural components (easily coded into the required functions) to cater for different purposes. The example has also points to the importance of an easily comprehended programming tool with a sound, consistent framework that can be used to code any agent as in the case of a procedural programming language that can be used to code any required functionality. The phrase “\textit{can be used to code any agent}” extracted from the above sentence might sound a bit exaggerated. A programming model that can be flexibly catered for a particular application domain would be a more appropriate description for such a realistic and practical agent-programming tool.

\(^2\)ASDK is available at the URL: \url{http://www.trl.ibm.co.jp/aglets/}
1.3 Research Objectives and Proposed Solution

The work in this thesis adopts an approach which attempts to integrate a constraint-based computing facility into an existing agent framework. This approach is parallel to embedding constraint-based computation into an intelligent system. The integration is anticipated to create a programming environment that provides a standard tool and constructs for specification and programming of agent. Such integration is also hoped to create synergistic value that will not only be beneficial to the overall application environment but also to promote the use of agent-based technology in realistic computing setting. In this research work, constraint-based computation is selected as the enhancement component with the expectation of providing the anticipated improved processing efficiency and application flexibility. The agent model adopted here is the well-known BDI framework, and the agent specification language to be extended is AgentSpeak(L) [Rao, 1996]. The choice of constraint computing paradigm as an integrating component is not an arbitrary one. Constraint-based computation is well recognised for its effectiveness in expressive representation. As the agent model used here is BDI framework in which beliefs of an agent form a major component in the agent architecture, the more expressive this beliefs can be stated and represented, the more effective will be the operations of an agent. This is the case because it is now endowed with information that commands greater precision and is more meaningful as compared to the original explicit belief representation (in predicate description). As mentioned earlier, constraint domain usually comes with specialised constraint solving technique, the integration will indirectly improve processing efficiency and performance effectiveness of the overall agent-based system. This is especially useful in resource-
bounded, time critical applications and will address the difficult, problems suggested by Mackworth et.al. [Zhang, 1994][Sahota, 1994]. Though AgentSpeak(L) is a FOPL (first order predicate logic)-based declarative language and constraint-based components might involve constraint expression such as linear or non-linear equations, these differences are not expected to interfere with one another as ultimately the reasoning algorithm in the agent framework is not expected to play a role in the constraint computation (constraint solving) process. The major modification required for the AgentSpeak(L) is the incorporation of an ‘interface’ that conveys the constraint solving result from the constraint solver to the agent and the agent will make use of the result for further processing or publish the result if required. The interface will play the role of an implicit technical gateway that makes the two differing computing paradigms complement one another to enhance the overall performance during computation.

1.4 Overview of Proposed Solution

In this work, the incorporation of constraint-based computation into AgentSpeak(L) will introduce the following two main elements:

1. The syntax of AgentSpeak(L) is modified slightly to introduce an additional data structure that enhances the original beliefs of the agent. This new data structure is the constraint beliefs.

2. The original operational algorithm is modified to cater for constraint handling as well as the passing of result from constraint solving to the agent framework and vice-versa.

The above modification on AgentSpeak(L) syntax and processing algorithm has reinforced the original operation with constraint solving which enables an agent to extend
its operational capabilities beyond a reasoning machine. Interaction between the constraint solver and the agent framework via the ‘interface’ gateway (constraint beliefs) has significantly extended the scope of deliberation. Previously, limitations on the deliberation existed mainly due to the restrictions on information representation. An example of such limitation is the inability of an agent framework to handle efficiently, more expressive representation such as the constraint expression (e.g. certain complex constraint expressions cannot be processed by the inference engine of the BDI framework). In this particular instance, from first look, the combination of BDI and constraint might seem to be a little strange. However, in actual fact, the blend of concept and principle from two disparate paradigms have created a win-win situation whereby each discipline has complemented one another in a way to generate a “synergistic effect” that will be hard if not impossible to achieve by themselves individually. The “synergistic effect” is asserted to be the extended application capabilities that emerge from the integration of the agent and constraint computing models. The BDI architecture as an intelligent agent framework contributes toward intelligent behavior through the notion of practical reasoning and its ability to adapt as well as to react. On the other hand, constraint-based computation enriches the BDI framework with more expressive knowledge representation and specialised processing capability that opens up more application opportunity for the agent program. Thus the integration of the two approaches has created a value-added feature such as more explicit as well as expressive belief representation and more intelligent environment that enables appropriate constraint solving to be performed.
1.5 Organisation of Thesis

This thesis is structured as follows:

Chapter 2 presents an overview on the needs of today’s software system and explains how agent based technologies can be fitted into the scenario to cater for the need of this expanding requirement. A brief history and classification of software agent is given next. This is then followed by a survey on the following primary areas that are relevant and have substantial influence on the work of this thesis.

- BDI Architecture
- Agent Programming Language
- Constraint based computation

Chapter 3 describes the proposed agent programming language ConstraintAgentSpeak in detail. The operational semantics are discussed and proof theory adapted from Rao [Rao, 1996] for the transformation of an agent from one state to another is formulated based on the execution flow prescribed by the operational semantics.

Chapter 4 outlines the implementation for ConstraintAgentSpeak in term of its interpreter. Procedures and steps occur during the interpreting process is discussed and examples are given to illustrate the working algorithm implemented within the interpreter. The main focus here is on how the tokenizing, parsing and program execution are being handled by the interpreter. A description for the implemented object classes is given at the end of the chapter.
Chapter 5 illustrates the agent model created using *ConstraintAgentSpeak* by providing two illustrative applications. The benefits derived from implementing an agent-based application using *ConstraintAgentSpeak* are discussed and sample programs are used to provide a clearer view on the declarative nature of the language. A discussions on other work that involve agents and constraints is presented to illustrate the similarities and differences in relation to *ConstraintAgentSpeak*.
Chapter 2

Background

2.1 Today's computing dilemma

In the current trend of computing environment, monolithic and centralized computing systems are inadequate to meet the problem-solving needs of modern organizations. Such systems are adequate for addressing a limited scope of computing problems but cannot provide and general solutions. They lack the flexibility in problem solving and the ability to adapt to the problems faced. The dynamic nature of today's computing environment and the increasing demands on computing power have made some software programs appear to be inadequate in providing an intelligent solution. Besides being able to perform routine tasks faithfully, software programs are increasingly required to exhibit reasonable amount of intelligent qualities to enable impromptu behavior that are not part of the routine procedures. In short, it is almost mandatory that software programs should be reactive and take the appropriate actions to take in order for them to be useful.

2.2 An Abstract of Agent-Based Technology

The notion of software agent is evolving leading to the development of computer systems with capability to behave autonomously. Agent-based computer program is deemed to be able to display flexible behavior for coping with dynamic computing setting and to be able to react accordingly to execute the appropriate problem solving procedures. Essentially the two basic characteristics of a software agent are:

- Autonomous and goal-directed behavior
• Reactive to changing environment

Even by just basing on the above two fundamental features, the development of a true software agent is not a straightforward and simple task. A wide variety of technical issues and design considerations have made agent-based technology a rich and challenging area of on-going research. The three major areas [Müller. 1996a] that are keenly explored by agent research communities are:

1. Agent architecture
2. Agent communication
3. Agent cooperation

While agent communication and corporation concern mainly with working agents in a multi-agent system (MAS), agent architecture predominantly focuses on the functional efficiency of an agent and represents the core work for all agent design and development. Incidentally the work in this area has substantially influenced the development of agent communication and cooperation. Besides being a key motivator for the progress in areas 2 and 3 above, agent architecture proposes a clear, explicit structural scheme in which the mental components as well as the decision-making process can be modeled and worked with to produce a cohesive control unit for a running agent. Thus looking at the above consideration, it appears that the efficiency, effectiveness and flexibility of an agent are largely determined by how agent architectures are adopted and deployed.

Formally, autonomous software agents can be identified as computing systems that are able to perform its assigned task autonomously in a complex, dynamically changing environment and behave accordingly by communicating with its environment [Jennings. 1998]. In this case, the environment can be:
1. A working shell with an isolated agent that has been designed for a particular purpose in a single agent scenario.

2. A multi-agent setting whereby individual agent takes up specific sub-function or sub-role that consolidates one another to achieve an overall intended goal. In such setting, the resulting multi-agent system will benefit from the synergistic effects generated from the set of integrated, cooperating agents.

2.3 Development of Software Agent and Agent Classification

As noted in the previous section, the growing complexity of today's organizations together with the ever-increasing dependency of human on computer-controlled processes or systems, have prompted the need of more capable software with greater mobility. The distribution of computing need across vast distributed locations is another factor that drives the demand for more competent ways of domain modeling and robust, improved computation mechanism for problem solving. The greatly diversified computing environment in real life situation has signaled the urgent need of software programs with interoperable capability in order to operate across heterogeneous platform. Collectively, these pragmatic requirements have served as the catalyst that spurs the rapid development of agent-oriented software programs with greater mobility.

The evolution of agent-based technology has been developed under the inspiration of three main AI disciplines: control theory, cognitive psychology and AI planning theory [Müller, 1996a]. Under these influences, control theory has engineered the situation recognition and action planning of an agent. Cognitive psychology has offered the guiding concept along the line of motivation theory that is applied to the formulation of intention in an agent. Lastly, AI planning system contributes towards the decision-
making for appropriate plan to be fired based on relevant situation or state through the use of symbolic representation. Most agent model can be classified into one of the 3 broad categories: deliberative agent, reactive agent, and interacting agent.

2.3.1 Deliberative Agent

Deliberative agent [Müller, 1996a] is by far the most widely researched area of intelligent agent. A deliberative agent is assumed to harness mental components that are maintained in the form of symbolic representation. Such symbolic representation explicitly models the world states or environmental information that is believed by the agent as well as the desires that an agent believes that they can be achieved. The symbolic representation (mental components) within a particular agent can be manipulated via symbolic reasoning in which the representation can be changed and updated. In one of the well-recognised deliberative agent’s architecture, such mental components can be organised into three distinct classes of beliefs, desires and intentions. An agent architecture that adopts this paradigm is known as BDI architecture. BDI architecture will be discussed in detail in section 2.4.

2.3.2 Reactive Agent

Reactive is another ‘species’ of agent which embraces a different approach to handling of an agent’s behavior. Established predominantly based on the work of Brooks [Brooks, 1986][Brooks, 1991] and Kaelbling [Kaelbling, 1990], reactive agent [Müller, 1996a] is a school of thought that disputes the idea of modeling an agent’s mental component via symbolic representation. Architectural design of reactive agent enables a situation-based reactive behavior pattern that acts according to the world or environment perceived by the agent (via some form of sensory input/device). Driven by assessment performed
during run-time, the on-the-fly decision making process is executed based on the limited input fed by the sensory device and is intended to produce a robust behavior pattern rather than a conscious decision making model.

The situation-based and activity-oriented design [Jennings, 1998] adopted in the architecture of reactive agent structurally decompose the agent into a composition of *activity organisers* (produce and response) that react to the agent’s external world. Situation activated rules or modules are deployed in response to perception received and actions are performed accordingly based on the activated rule or module. However, such trigger of activity in turn depends very crucially on the right *activation value* for the rule or module, which is unlikely to be defined comprehensively as the actual environment or world could present unprecedented situation at any time.

In contrast to deliberative agent, the reactive agent adopts a drastically different approach to the control and manoeuvre of an agent’s behavior. However, strategically, both types of agent share the common notion of being able to autonomously determine what is the next appropriate course of action. In doing so, the deliberation process (represented by the processing on the symbolic representation) in deliberative agent is replaced by the direct interaction of an agent (reactive) with the world or the environment. The explicit mental component that exists in deliberative agent is directly reflected in the situation of the world or environment at a particular instance of timing.

### 2.3.3 Interacting Agent

The development of interacting agent [Müller, 1996a] is somewhat different from the direction of the above two types of agent. Instead of concentrating on the behavioral aspects of an agent, the main concern is on the efficiency of the communications that take
place within a multi-agent set-up. The primary focus of the interacting agent is on the coordination and cooperation between agents in the multi-agents environment. Being closely related to distributed artificial intelligence (DAI), the use of interacting agent is essentially concerned with the deployment of a group of cooperating agents (a multi-agents system) to deal with isolated, distributed sub-problems of an overall problem. Thus coordination mechanism is the principal concern rather than the framework or structural components of the agent. The following are a number of aspects considered to be significant in the design of an interacting agent.

Communication. Communication mainly deals with the protocol and methodology used during the relay of messages and information exchange between interacting agents. One such communication facility is the KQML [Finin, 1994] - a standard knowledge sharing language that can be used as the interacting tool among agents.

Distributed Problem Solving (DPS). DPS deals primarily with mechanism for decomposing a main task into appropriate components, protocol for allocating sub-tasks to designate problem-solver agents and coordinating these agents to achieve an overall problem solving strategy.

Conflict Resolution. As in DPS, the main focus is on resolving constraints or conflicts that arise during the cooperating problem solving session. In this case, DPS serves as the set up that demands proper handling of an agent’s cooperation and resolving constraints or conflicts in order to facilitate cooperative problem solving.

2.3.4 Improvised Approach - A Hybrid Agent Architecture

Hybrid architecture has become popular over the years as researchers try to bridge the gap that spans between the above three basic agent architectures. Layered approach (e.g.
[Brooks, 1986][Firby, 1992][Lyons, 1992]) is by far the most favourite standard technique to an integrated architecture that structures agent functionality and most importantly, enables agent design that supports the implementation of desired properties such as reactivity, deliberation and cooperation. The hierarchically organised layers interact with one another to achieve coherent behavior of the agent as a whole and at the same time allow for:

- modularised of different functionality of an agent.
- compact design that facilitates modification and debugging.
- concurrent execution of different layers improves performance and computational capability.
- partition of different knowledge and restrict them into the relevant layers meant for different functionality which enables a more efficient handling and processing of information available to an agent.

2.3.5 Summary

The above sections have outlined the three main, broad classes of agents. The first three classes (deliberative, reactive and interacting) represent the trend of intelligent agent design that has developed over the years. Architectural design of the respective class of agent has been oriented towards the respective behavioral discipline that is clearly indicated by the name of the agent class. Together with the hybrid architecture, the four classes of agent form the area of agent design under the broad umbrella of artificial intelligence. Despite considerable effort dedicated to the exploration on the approach for more effective and efficient integration, the result is still far from satisfactory. So far, layered approach, which is adopted in a majority of hybrid architectures, has remained
the most effective technique that addresses the shortcomings associated with the individual agent class.

2.4 BDI Architecture

2.4.1 Belief, Desire and Intentions

The history of BDI architecture can be traced back to Bratman et al [Bratman, 1987a] in 1987 and since then the notion of Belief, Desire and Intention have developed into a well recognised approach in the design of agent architecture. Based on this paradigm, the essential idea of this approach is to depict the state of an agent at a particular instance through the use of the respective mental components. The processing performed on these mental components change its contents accordingly and the control of these changes is achieved through the decision-making process that takes place within the agent. The notions of goal and plan are introduced to supplement the existing BDI agent architecture and to enable decision-making process to take place. The basic concepts for each of the above notions are as follows:

- Beliefs – These are the explicit representations for the current state of environment in which the agent is situated and what an agent believes to be possible. This will determine the next course of action that an agent will take.

- Desires – High level abstract specification of the preferences of an agent. Inconsistent desires are allowed in an agent even though it is conflicting with its beliefs.

- Intentions – These are the committed goals that an agent has selected to pursue base on its available resources and is a subset of the possible goals available to an agent.
- Goals – This is a subset of consistent desires that an agent believes to be achievable. However they are not committed to be achieved by the agent. Committed goals eventually transform into intention of an agent.

- Plans – Plans are practical implementation of intentions in which they are organised into a stack of intended plans waiting for processing.

The next two sections will provide brief descriptions on the origin and initial research about the BDI framework.

2.4.2 IRMA architecture: Bratman et al.

In the work of Bratman et al. on BDI based architectural design, the IRMA architecture (Intelligent Resource-bounded Machine Architecture) [Bratman, 1987b][Bratman, 1988] modeled the agent’s behavior based on some explicit representation of the beliefs, desires and intentions. Modularised structures of reasoning components (e.g. intention structure, means-end reasoner, opportunity analyzer, filtering process and deliberation procedures) collectively form the control architectures that determine the flow of control based on the agent’s current beliefs. Plans in IRMA take up a twofold role, on the one hand they are viewed as plan library which consists of a repertoire of beliefs and action that can be applied to achieve intended goal. On the other hand, they are also treated as committed means that compose the intention structure of an agent. Despite being the ‘pioneer’ in BDI model, the emphasis on realistic application in IRMA has led to the development of a school of philosophical view points on BDI architecture rather than a formal definition on beliefs, goals and intentions as well as their processing. The work on formal model of BDI was picked up by other researchers (such as Cohen and Levesque [Cohen, 1990], Rao and Georgeff [Rao, 1991]) at a later stage.
2.4.3 Formal BDI Model: Rao and Georgeff

Rao and Georgeff provide formal model of BDI-architecture in term of the possible-world logic. In this formalism, three main issues have been addressed. Firstly, the first-class citizen treatment of intentions enables the definition of different strategies of commitment with respect to intentions (via the use of different plans of action) and thus the possibility of modeling a wider variety of agents. Secondly, the distinction of choice and chance (possibilities), i.e. the distinction between the ability of an agent to deliberatively pick an action from a set of alternatives and the possibilities of outcome determined by the environment. Thirdly, specification of interrelationship between beliefs, goals and intentions to avoid problems such as commitment to unwanted side effects.

In the possible world formalism, the world in which an agent is located in is modeled using a temporal structure with a branching time future (with a single past) called a time tree. In this context, a particular time point in a particular world is called a situation and the transformation from one time point to another is triggered through event. The formal language used to describe these formal structures is a variation of the CTL* (Computation Tree Logic [Emerson, 1989]). Two notable facts within this formalism are:

- the use of state formula and path formula. The distinction between the two formulas is the former which is evaluated at specific time point in a time tree and the latter over a specified path in a time tree.

- two modal operators, optional and inevitable are introduced for operating only in path formula. A path formula $\Phi$ is optional if, at a particular time point in a time tree, $\Phi$ is true of at least one emanating path, is inevitable if $\Phi$ is true of
all emanating paths. (Other standard operators next, eventually, always and until operate over the state and path formulas)

Semantics of the formalism is defined in three parts: semantics of state and path formula, semantics of event and semantics (possible-world) of beliefs, goals and intentions. The semantics of state and path formula is given by an interpretation \((M)\) that maps a standard first-order formula from one situation (individual world in the form of time tree) to another situation. The semantics of event provides a mechanism for defining the success or failure of an event during the transformation from one time point to another. Semantics of beliefs, goals and intentions are in the form of possible world semantics. In each situation in a time tree, there exists a set of belief-accessible, goal-accessible and intention-accessible world which characterise the worlds that an agent believes is possible, desires to achieve and has committed to achieve.

Axiomatization or semantics conditions are used to capture and enforce the required interrelationships among an agent’s beliefs, goals and intentions. Under such relationship restriction, beliefs, goals and intentions become closed (compatible) under implication and need to be consistent. Two examples of such important requirements are belief-goal compatibility and goal-intention compatibility. The former axiom essentially states that if an agent has goal \(\Phi\) that \(\text{optional}(\Phi)\) is true, the agent also believes that there is at least one path in the belief-accessible worlds in which \(\Phi\) is true, i.e. \(\text{GOAL}(\Phi) \supset \text{BEL}(\Phi)\). The latter emphasizes that if an agent adopts \(\theta\) as intention, then the agent should also have adopted \(\theta\) as goal to be achieved, i.e. \(\text{INTEND}(\theta) \supset \text{GOAL}(\theta)\).
2.4.4 Interpreter for A BDI Agent: Rao and Georgeff

Review on the BDI theory (logics) [Rao, 1991] has produced the criticism that there is no efficient implementation available for BDI architecture based on its multi-modal logic specification. The influence of BDI logics on actual implementation is negligible. Rao and Georgeff address this issue by providing an abstract interpreter for a BDI agent [Rao, 1995]. The abstract interpreter represents a pragmatic abstraction of the theoretical framework by making some simplified assumptions. It comprises of data structures corresponding to beliefs, desires and intentions as well as a queue of events. The explicit identification of the data structures has not only improved the efficiency and precision of communications with humans and other agents, but also simplified the building, maintenance and verification of the agent-based system. Updating operations on these data structures are subjected to compatibility requirements and formalised constraints on the agent's mental components. Despite being the basis for practical reasoning systems, the interpreter is still subject to the following problems:

1. Due to the existence of logically closed set of mental components (belief, goals and intentions formulae), procedures specified in the interpreting cycle will involve provability procedures that are basically not computable.

2. Reactive ability is by the time taken to perform an interpreting cycle. The option generator and deliberation processor (intended plan selector) are not open to modification for coping with real-time requirement demand in some agent-based applications.
Addressing the above problems has resulted in a new practical BDI architecture with the following assumptions. The Procedural Reasoning System [Georgeff, 1989][Georgeff, 1986] is the first actual implementation based on the above new design.

1. Formulas for beliefs and goals are restricted to ground sets of literals without implications or disjunctions.

2. Only current states of the world are explicitly represented. Intuitively, these currently held representations could be changed over time as the agent’s beliefs change.

3. Means of achieving certain world states are represented as the plan of an agent. Plans are abstract specifications of both the means for achieving certain desire or goal and the options available to the agent.

4. Intention of an agent is a set of adopted plans of action and is represented as a run-time stack of hierarchically related plans. Multiple intention stacks can coexist and run in parallel.

In summary, the translation of BDI multi-modal logic to practical programs that are able to solve real-life complex problem is only a partial success. Despite the fact that an agent’s available options or strategies can be implemented using plans, it does not contain any predictable decision-making process on how an agent makes decision to select a particular option or plan from a set of available alternatives. In addition, there is no feature available to support reactive behavior; all situations are uniformly treated by a generic plan selecting process.
2.5 Agent Programming Language

This section provides a brief outline on the notion of agent programming and specification. A detailed description of AgentSpeak(L) is given in this section which serves to unveil the motivation for the work presented in this thesis. In the simplest and direct terms, the task of programming a BDI agent is essentially creating a software entity whose state at any particular instance of time consists of mental components such as beliefs, capabilities (plans library), choices and commitments. At a more abstract level, agent programming can be viewed as the specification of conditions for making commitments that result in executable actions. A number of agent-oriented programming languages for BDI framework such as AOP [Shoham, 1993], PLACA [Thomas, 1993][Thomas, 1995] have emerged and in one way or another have provided data structures for the representation of an agent’s mental categories. As a BDI agent programming language, PLACA is more complete compared to AOP in the sense that it provides better expressive power through the use of plans, a PLACA agent’s intention is a subset of plans the agent has committed to. Each of the language has also included an interpreter for execution of the agent specification in the respective languages.

2.5.1 AgentSpeak(L) : A BDI Agent Programming Language

2.5.1.1 Introduction

AgentSpeak(L) [d'Inverno, 1998][Rao, 1996] is a logical computable language for agents based on the BDI framework. The terms logical computable reflects the emphasis and the intended objective of AgentSpeak(L) – to narrow the gap between the expressive logical specification and an efficient implementation. The inherited complexity of theoretical specification which makes use of logics and modal operators has always been the greatest
hindrance encountered by agent researchers when it comes to validating their work through practical design and implementation. This has resulted in the lack of one-to-one correspondence between the theoretical properties and the eventual system. Thus implemented systems for the most part lack of strong theoretical underpinning. In AgentSpeak(L), Rao took a radical approach by providing an alternative formalisation for BDI agents. His attempt is a 'reverse engineering' strategy. Instead of the usual approach of translating formal logic specifications into implemented system, an implemented system (PRS [Ingrand, 1992][Georgeff, 1987]) is selected and its operational semantics are formalised into model theory and proof theory to facilitate the construction of agent specification logics. This alternative attempt opts for the proof-theoretic approach as compared to the CTL* style formal language specification (in which the use of standard modal operators is deemed to be an implementation obstacle). The essential operational aspect is the interpretation process, which is similar to the processing of horn-clause logic programs whereby unification is the driving force behind the operation of the entire BDI framework.

2.5.1.2 Agent Programs

Technically, AgentSpeak(L) is a programming language based on a restricted first-order language. It provides an alternative formalisation of BDI agents with first-order characterisation. Thus the alphabet of this formal language is partially made up of the usual first-order predicate logic representation. The language constructs in AgentSpeak(L) are built based on construction rules that apply to the building of terms, first-order formulas or closed formulas in FOPL. Besides these basic FOPL symbols,
additional symbols are used to support other functionality of AgentSpeak(L) such as the application of goal (! or ?), addition or deletion of belief or goal (+ or -).

As AgentSpeak(L) is a language design for programming BDI agent, it comes with provisions that can be applied to model the basic data structures of BDI framework. In doing so, it provides not only the data structures for the explicit representation of the agent’s mental attitudes, but also facilitates the modeling of agent’s state similar to the use of state formula and path formula in the CTL* fashion language. For example the beliefs (or current belief state) of an agent at an instance is in fact a model of itself, its environment and its surrounding agents at that particular instance. Its adopted desires (goals) will be the states that an agent would like to bring about based on the invocation stimuli and its intentions are the subset of adopted desires that satisfy the stimuli.

Essentially an AgentSpeak(L) agent consists of a set of base beliefs and a set of plans. Base belief is ground belief atom [ground atomic formula eg. location(car, lane2)] and plan is context-sensitive, event-triggered recipe(means) that can be hierarchically decomposed into sub-goals or perform action execution. In this context, the goals or sub-goals will be the agent’s state that an agent is intended to bring about (achieve !) or wished to query (test ?) whether it is its own belief or not. While working to achieve its goal, action may also be performed to change the base belief(s). Plan is a means-end specification with a triggering event (head) to invoke the plan, a context sensitive section to test whether a particular subset of base beliefs hold and finally the body which consists of a sequence of sub-goals or actions. Thus in summary, an agent (in BDI framework) can be specified or programmed by writing a set of base beliefs and a set of plans.
2.5.1.3 Agent Operation

Before going into the operation of an AgentSpeak(L) agent, a number of concepts need to be clarified and explained. This section will begin with the notion of event. Event is the basis for the operation flow in that they are the cue or the trigger for the reasoning and action. Event can be either an external event, event result from external source (such as other agent, user or the environment) and thus is unrelated to any intentions hold by the agent, or an internal event, in which case it is resulted from the execution of a current intention. In the case of internal event, the event is actually a sub-goal of an intention that belongs to the agent. Plan does not exist in ground form as base belief. A plan instance is a partially instantiated plan whereby some of the variables have been bound to terms. Intentions are plans that an agent has committed to execute and this set of committed plans are arranged in the form of a plan stack in which the first original intended means or plan instance is located at the bottom of the stack. Other components which are required to make a complete AgentSpeak(L) agent are:

- Plan library: a repository that contains all the available plans at the disposal of the agent.

- Event-selection function: selects an event for processing from the event queue.

- Applicable-plan-selection function: pick an appropriate plan from the plan library based on the invocation condition and context of the plan.

- Intention-selection function: select an intention to be executed from the intention set.
The execution of an AgentSpeak(L) agent is performed in an operation cycle that make up of two mode of operations. Before an agent begins its operation, it is initially given a set of beliefs to start with. With this initial knowledge base, the agent will begin its operation when the first triggering event (an external trigger) enters the event queue. The first mode of operation is the response of an agent to a triggering event. In this mode of operation, an applicable plan is generated and is instantiated to become an intended means. At the end of the operation, a new intention with the new intended means will be created if the triggering event is from an external source. Otherwise the new intended means is pushed on top of the plan stack which form the intention that initiate the internal trigger event. The second mode of operation serves as the respond of the agent to the intention(s) resulted from the first mode of operation. Thus essentially intention execution is the main focus in this sub-operation. A single intention (selected intention or executing intention) is selected one at a time from the intention set. The standard naming convention is that the first intended means on the top of this plan stack is the executing plan and the next formula in the body of the executing plan is the executing formula. Based on the executing formula, one of the three alternative courses of action is taken.

1. If the executing formula is an achievement goal, a new event (internal event) is generated and posted to the event queue. The execution of the selected intention will be suspended pending the achievement of the sub-goal. The suspended intention will be reinstated when a new intended means is generated and push on top of the intention's plan stack.
2. If the executing formula is a query goal, the formula can be unified with the base beliefs. The most general unifier (mgu) is applied to the rest of the executing plan.

3. If the executing formula is a primitive action, it is then executed to perform the action desired.

In the last two courses of action, the executing formula is removed after the execution. In this case, the next formula in the body of the executing plan is selected for execution. If the body of the executing plan is empty, then the next plan in the plan stack will be the new executing plan. If there is no next formula or next plan, then the intention has been successfully achieved and will be removed from the intention set.

AgentSpeak(L) is one of the more impressive development based on BDI agent framework in terms of bridging the gap between the theoretical foundation and the practical feasibility of implementation. The language indicates the fact that adopting a unique 'reverse' approach of design based on an implemented feasible system has indirectly ensured that the resulting agent programming language is practical, acceptable and most importantly, the practical (implementation) aspects is correspond to the theoretical (formal) aspects - a formal computational model. With such a well-defined specification language based on an established agent framework, the obstacles involved in creating an autonomous agent program with all the necessary basic mental components is greatly reduced. This contribution to increase the possibility of creating an agent program endowed with reasonable amount of intelligence. However the use of symbolic representation in the data structures of the mental components has also introduced restrictions that limit the exploitation of the full capacity of agent programs. Symbolic
representation that can only be interpreted literally (via symbolic processing and unification) will tremendously reduce the expressiveness of the agent language and hence greatly limit the scope of application for the language. This limitation is the primary motivation that drives the work of this thesis to research the possibility for improving the explicitness as well as expressiveness of the representation used in AgentSpeak(L) and hence extending the application scope of this agent programming language.

2.6 Constraints

Constraints-based computation has recently emerged as a research area that has attracted the interest of researchers from artificial intelligence, programming language, symbolic computing and computational logic. Currently the two main areas of research that involve constraint theme are: constraint satisfaction problem and constraint solving. Both of these are unified or integrated under the new computing paradigm – constraint programming.

Before go on to look into more detail on these two areas in detail, it would be helpful to provide a clearer picture of what constraints are. This section will provide a technical definition of constraints and give a basic introduction that will be helpful in understanding how constraints can be fitted into the bigger picture of constraint-based computation.

Technically a constraint can be defined simply as a logical relation that exists among a number of unknown or variables [Marriot, 1998a] [Kumar, 1992]. A number of domains exist as well and each domain contains a number of values pertaining to the problem domain involved. Each of the variables can then be associated with a particular domain that contains all the possible values pertaining to that variable. Based on the two basic ideas of variables and domains, a constraint can be said to impose a restriction on the
possible values that a variable can take. It represents some partial informations of the whole picture and involves only the relevant subset of all the variables. A number of basic properties [Marriot, 1998a] that apply to all constraints are listed as follows:

1. Constraints can represent partial information and need not specify exactly the values of variables involved.

2. Constraints are non-directional. A constraint on two variables X and Y can be used to infer a constraint on X given a constraint on Y and vice versa.

3. Constraints are declarative. It is used to specify the relationship that must hold without specifying the computational procedure to enforce the relationship.

4. Constraint are additive. Given a set of constraints, the order of imposition of constraints does not matter, all that matters is the conjunction of constraints is in effect at the end.

5. Constraints rarely exist independently. Typically constraints share variables among themselves.

As constraints arise naturally in most of the human endeavour in the real world, constraints can be viewed as a natural medium that can be used to model and express objects, problems or processes that humans need to deal with. Concomitantly, a new computing paradigm named constraint programming has evolved and is built upon the basic properties of constraints mentioned above. Technically constraint programming can be defined as the study of computational systems based on constraints. The primary objective of constraint programming is to reap the advantage of constraint representation. This is done by simplifying a real world situation into a system of constraints about the real world object or process, and uses the resulting systems of constraint to better
understand the problem and behavior of the real world phenomenon. Although being able to define entities that exist within a problem domain is great, the additional ability to make use of constraints to specify relationships or constraints among the different entities has provided an even more powerful and complete approach to modeling problems and solving them. This is especially the case when it is required to simply state the relationships or constraints, the underlying implementation is then responsible for maintaining these relationships or constraints by involving only those entities that are able to satisfy the given constraints. Thus the constraint programming is a computing paradigm in which the core operations consist of the specification of requirements (constraints) and the generation of solutions to these requirements through the use of specialised constraint solvers.

2.6.1 Constraint Satisfaction

Constraint satisfaction arises from the research in artificial intelligence specifically in the area of combinatorial problems. In the artificial intelligence computing, satisfaction of constraint problems over finite domain has been studied and researched under the name of constraint satisfaction problem or CSP for short. Intuitively CSP can be characterised by the following features:

- a set of finite variables,
- a set of finite domain,
- a function which maps every variable to a finite domain, and
- a set of constraints.

The discussion of CSP in the following section is restricted binary CSP (in which each constraint is either unary or binary). It is possible to convert n-ary constraints to another
equivalent binary CSP [Rossi, 1989]. Incidentally a CSP can also be represented using a constraint graph or constraint network. In such graphical representation, the variables are represented as nodes, the arc or edge represents a constraint between variables at the end point of the arc. The labels of the arcs define the constraint and the labels of the nodes represent the domain of the variables. Each of the constraint will restrict the combination of values that a set of variables may take simultaneously. A solution to a CSP is to locate the assignment to each variable a value from its domain that will satisfy all the constraints. Under this class of constraint programming, the typical standard approach is to begin by defining all the entities and domain values that will be involved. This is then followed by modeling the computing problem at hand as a constraint satisfaction problem. In doing so constraints are formulated to represent the relationships that exist and finally, solving procedure is then applied to search for solutions that satisfy the constraints defined. A constraint solver will be involved during the process of solving for solution. Classical examples that are always mentioned when it comes to discussion in AI concern with CSP are the map colouring problem and the N-queen problem. In the map colouring problem [Marriot, 1998b], the problem domain is concerned with attempting to colour different regions in a particular map with a limited number of colours, subject to the conditions that no two adjacent regions should have the same colour. For instance, in the following figure 2.1 on the map of Australia, the entities that are of interest are the respective states on the map. Each of these states can be assigned with a variable to represent the region: WA, NT, Q, SA, NSW, V' and T. The domain values will consist of colours to be used for filling the respective regions: \{red, yellow, green\}. Each of the above variables will be associated with the same set of domain values. The set of
constraints that can be formulated based on requirement that no adjacent region should be filled with the same colour are: \(WA \neq NT \land WA \neq SA \land NT \neq SA \land NT \neq Q \land SA \neq Q \land SA \neq NSW \land SA \neq V \land Q \neq NSW \land NSW \neq V\). Thus the process of solving for solution would be to determine the sets of colours that can be assigned to the respective states and the colour assigned should be consistent with the set of constraints listed above.

In the N-queen problem [Marriot, 1998b], the challenge is to place N-queens on a chess board of size \(N \times N\) so that no two queens are allowed to be placed on the same line horizontally, vertically or diagonally, i.e. each queen should not be able to captures one another. The entities of interest here will be the position of the queen on the chessboard and this can be indicated via coordinates whereby two variables are used to represent the column (C) and row (R) of a particular position. In an \(N \times N\) chess board, the variables will be \(\{C_1, ..., C_N\}\) and \(\{R_1, ..., R_N\}\). Similar to the set of variables, the domain values will also be restricted by the size \(N\) of the chess board. For instance, a 6 by 6 chess board will have a set of domain values \(\{1, 2, 3, 4, 5, 6\}\) and both the column as well as the row variable will be associated with this set of domain values. Example of constraints formulated will be such as:
R₁ ≠ R₂, R₁ ≠ R₃, R₁ ≠ R₄, ……

which ensures that no two queens can be in the same row.

C₁ ≠ C₂, C₁ ≠ C₃, C₁ ≠ C₄, ……

which ensures that no two queens can be in the same column.

C₁−C₂ ≠ C₂−C₂, C₁−R₁ ≠ C₃−R₃, C₁−R₁ ≠ C₄−R₄, ……

and

C₁+R₁ ≠ C₂+R₂, C₁−R₁ ≠ C₃−R₃, C₁−R₁ ≠ C₄−R₄, ……

which ensures that no two queens can be on the same diagonal.

2.6.1.1 Solving Technique for CSP

From the above examples, an obvious standard feature of CSP is that the domain value involved is finite. It is also obvious that CSP is a combinatorial problem in which solving by search is the principal action in the process of deriving the solution. The constraint solving process is basically a variant of the searching process with the algorithm that navigates through the search tree in order to arrive at a possible solution. Various patterns of search are available and each comes with an efficient algorithm to implement them. [Paul, 1988] provides a good outline of feasible algorithms that can be adapted to do the above task.

The simplest and straightforward solving method is using the generate-and-test method (GT). In this method, each possible combination of values for the variables is systematically generated and tested to see whether they are consistent with the specified constraints. The total number of combinations considered by this method is equivalent to the size of the Cartesian product of all the variable’s domain value. Thus this algorithm is
guaranteed to find a solution if one does exist. Obviously, this method is not efficient (takes a long time to run) and hence is not a common or popular method in solving CSP. Another more efficient and common method used in performing search in solving CSP is the \textit{backtracking} method (BT). Backtracking is essentially a depth-first search which sequentially instantiate variables with values from their respective domain. Along the instantiation process, as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is verified. If verification reveals that the partial solution violates any of the constraints, backtracking is performed to the position of the most recently instantiated variable and still has alternatives available. By backtracking, any occurrence of partial solution that violates a constraint will effectively eliminate a sub-space from the entire Cartesian product of all variables' domains. Despite performing better compared to GT, BT run-time complexity is still exponential. The two major drawbacks of backtracking are:

- The repeated failure caused by the same reason, i.e. thrashing. This occurs because the failure of the algorithm to identify the conflicting variable. Subsequent searches in different sub-trees following the conflicting variable will keep failing due to the same cause (i.e. the same conflicting value assigned to the conflicting variable). Intelligent backtracking will help to address this problem directly.

- Performance of redundant work. This is due to conflicting values identified even during intelligent backtracking is not ‘remembered’. Hence the same failure will be repeated for the same set of conflicting values.
2.6.1.2 Consistency Technique

In CSP, consistency techniques are introduced mainly to prune the search space and thus indirectly improve the efficiency of the search process by cutting short the search for the solution. The handling of constraints using consistency technique is basically a domain 'shrinking' process. In this approach, the technique of propagating information about variables via the constraints between the propagation continues until no further new domain reduction is possible. During the constraint reduction process, each of the constraints is considered in turn and using the information about the domain of each variable in the constraint is used to eliminate values from the domains of other variables. Solvers that are equipped with such consistency-enforcing algorithm are said to be consistency based because it propagates information about allowable values from one variable to another variable until all the domains are consistent with the constraint. Consistency techniques alone rarely can be used to solve a problem totally. Consistency based solver is frequently combined with backtracking to produce a more effective and efficient search strategy. In this case, the consistency test is used to prune the domain of the variables as the search process transverse over the search tree to look for a solution and has proved to be effective in a wide variety of hard search problems.

2.6.1.3 Node Consistency

This is the simplest consistency technique, which is concerned with the unary predicate of variables in a set of constraints. In a constraint graph, the node represents a variable V is node consistent if for every value 'a' in the current domain of V, each unary constraint on V is satisfied. If the domain of a variable V contains a value that does not satisfy the unary constraint on V, then the instantiation of V to that value will always result in
immediate failure. Thus node inconsistency can be eliminated by simply removing those values from the domain of each variable V that do not satisfy the unary constraint on V. The algorithm for node consistency is available in [Kumar, 1992].

2.6.1.4 Arc Consistency

In a constraint graph, binary constraint corresponds to arc or the edge of the graph. An arc \((V_i, V_j)\) is arc consistent if for every value of ‘a’ in the current domain of \(V_i\), there is some value ‘b’ in the domain of \(V_j\) such that \(V_i = a\) and \(V_j = b\) is permitted by the binary constraint between \(V_i\) and \(V_j\). However the notion of arc consistency is directional as a consistent arc \((V_i, V_j)\) does not necessary implied that \((V_j, V_i)\) is also consistent. An arc can be made consistent by simply deleting those values from the domains of the variables (nodes) where these values are creating conflicting assignments (of values to variables) against the constraint that exists between the variables. The removal of these values will not eliminate any solution of the original CSP. Similar to node consistency, the algorithm for arc consistency is available at [Kumar, 1992]. However there are a number of versions whereby each revised algorithm rectifies the shortcoming of the earlier version and results in enhancement with greater efficiency.

2.6.2 Constraint Logic Programming

Constraint logic programming (CLP) paradigm [Jaffar, 1994a] is derived from the merger of two declarative computing paradigms: constraint solving and logic programming. Despite that it is a relatively new field, it has progressed in several different directions and is beginning to gain commercial significance. The main motivation is that CLP has the power to tackle those difficult combinatorial problems encountered for instance in job scheduling, timetabling, and routing which stretch conventional programming techniques
beyond their breaking point. Though CLP is still the subject of intensive research, it is already being used by large corporations such as manufacturers Michelin and Dassault, the French railway authority SNCF, airlines Swissair, SAS and Cathay Pacific, and Hong Kong International Terminal, the world's largest privately-owned container terminal [Pountain, 1995].

As stated earlier, CLP has part of its roots in logic programming. This ancestral link has made CLP a computing paradigm developed with a strong influence of logical framework. However in CLP, this logical framework has been supplemented by objects that have meaning in a specific application domain - for example in the CLP scheme introduced by Jaffar et. al., CLP(R) [Jaffar, 1992] is a scheme that deals with problem domain involved with real numbers along with their associated algebraic operations (eg. addition and multiplication) and predicates (eg. =, <, and >). Hence there is not a single CLP language, but a whole family of them defined for different application domains, i.e. constraint logic programming language is parametric in its choice of underlying constraint domain and the solver for that domain. For instance, a CLP programmer introduces 'constraints' (eg. \( X > 0 \) or \( Y+Z < 15 \)) into programs, which have to be satisfied for successful execution of the programs. In terms of operations, CLP is built with enhancements whereby the simple unification algorithm that lies at the heart of logic programming is augmented by a dedicated 'solver' for the particular domain of application. The solver will receive as input a set of constraints and determine whether the set of constraints is solvable (whether all the constraints are consistent with one another). The solvers for CLP systems normally require to be incremental, so that adding a new constraint to an already solved set does not force them all to be re-solved. This
improves the overall efficiency of the run-time operation. A number of generic features are present in constraint logic programming such as:

- In the absence of complete information the answer might be a symbolic expression like 10 - X, or even a constraint like X > 23 (in real arithmetic domain).

- Provide richer data structures that allow for more expressive and explicit representation of relationships that exist between objects or processes to be modeled.

The core idea of introducing more expressive data structure is to replace unification (the main computation heart of logic programming) with constraint handling procedures for the constraint domain in a particular CLP.

2.6.2.1 Rules (User-Defined Constraints)

Constraint logic programming languages provide only one programming construct — 'rules'. Rules (or less formally known as user-defined constraint) allow programmers to define their own constraints in term of the underlying constraint domain of the CLP language. The scenario of modelling an object or process with a constraint can be viewed as comprising two discrete phases: first, a general description of the object or process is modeled, then this is followed by modeling the specific information detailing the situation of the problem at hand. In most situations the general constraint description will be reused and it will be useful to have a mechanism that enables some constraint descriptions to use repeatedly in different problems. In such situation, rules will come in handy by allowing the programmer to define problem specific constraints in term of the underlying constraints. Rules formulated in this way can be used to evaluate the 'goal' to
derive solution. In doing so, rules are used to replace goals with definition of user-defined constraints. The process is repeated until only primitive constraints are left and the set of resulting constraints is the ‘answer’ to the goal. Such process of using rules to obtain an ‘answer’ is called ‘derivation’.

2.6.2.2 Basic Terminology and Example of CLP

User-defined constraint is of the form \( p(t_1, \ldots, t_n) \) where \( p \) is a n-ary predicate and \( t_1, \ldots, t_n \) are expressions (variable or term) from the constraint domain. A literal is either a primitive constraint or a user-defined constraint. A goal is a sequence of literals of the form \( L_1, L_2, \ldots, L_n \) where \( n \geq 0 \) and \( L_i \) is a literal. A rule is of the form \( A :- B \) where \( A \) is a user-defined constraint (head) and \( B \) is a goal (body). A fact is a rule with an empty body and is simply written as \( A \). A constraint logic program is then made up of a sequence of rules [Marriot, 1998]. The following is a very trivial sample program use to illustrate the basic structure of a CLP program. This sample is a CLP(R) program [Heintze, 1992] with real constraint domain in which the main constraint handling operation is real arithmetic.

```
mortgage(P, Time, IntRate, Bal, MP) :-
    Time > 0, Time \leq 1, Bal = P^*(1+Time*IntRate/1200) - Time* MP.
mortgage(P, Time, IntRate, Bal, MP) :-
    Time > 1, mortgage(P^*(1+IntRate/1200)-MP, Time-1, IntRate, Bal, MP).
```

The above mortgage computation program models the relationship that exists between the parameters [principal, mortgage life (month), annual interest rate (%), monthly payment and outstanding balance] involved in a property mortgage contract. The definition for the predicate \( \text{mortgage}(P, Time, IntRate, Bal, MP) \) appears as a sequence of rules in the CLP program. Query can be performed against the program by submitting (input) a goal plus the other information available. For instance the goal

\[ ? \text{mortgage}(100000, 180, 12.0, MP). \]
will present a query asking how much is the monthly repayment to finance a S$100,000 mortgage at interest rate of 12% for 15 years. The constraint handling, which is real arithmetic for this real constraint domain will return an answer MP=1200.17. The goal can also be presented in different way by providing different information such as

? mortgage(P,180,12,Bal,MP).

will return the answer

\[ P = 0.166783 \times \text{Bal} + 83.3217 \times \text{MP} \]

The emphasis of the above example is that the constraint solving (handling) not only able to return specific answer if sufficient information is available, but also able to provide partial solution in the form of answer constraints with respect to whatever information that is available. Thus the example of answer constraint P above serves as solution that indicates the relationship exists between P, MP and Bal. A detail description of CLP(R) implementation together with operational design of the constraint solver used to perform the type of constraints handling given in the above example is available in [Jaffar, 1992].

2.6.2.3 Goal Evaluation

Operations of a CLP program during run-time (as illustrated by the example in the previous section) display the behavior in which rules are used repeatedly in the evaluation of a goal. A renaming process will be applied during the evaluation process to map variables to variables whereby no two variables will be mapped to the same variable. The intention of renaming is to ensure each time when a rule is used for evaluation, the process will involve only a distinct set of variables. As a result, quite a number of intermediate variables will be generated from the renaming process but they are not of interest. The final solution is obtained by simplifying the constraint store’s constraints
with respect to the variables in the initial goals. Throughout the evaluation process, the resulting set of primitive constraints is checked regularly to see whether they are satisfiable. If they are not, no further evaluation will be done as a consistent set of constraints will never be obtained.

Formally, the process of evaluation in CLP can be treated as a sequence of derivations. At each step of the derivation, a literal is selected for processing. If the selected literal is a user-defined constraint, it will be re-written (rename) into the goal. If it is a primitive constraint, it is added to the constraint store. The new constraint store is then tested for satisfiability. Thus at each step of derivation, the constraint store will contain a conjunction of primitive constraints. The derivation will continue until the constraint store contains only primitive constraints and there are no more user-defined constraints available in the goal for rewriting or renaming or an empty goal is resulted. A constraint logic programming language interpreter normally evaluates a goal by first constructing a derivation tree (search space) and then traverse the tree to search for successful derivations. For the sake of operation efficiency, the entire derivation tree is not constructed before the traversing begin. Instead the tree is built dynamically as the tree is searched for answer.

2.6.2.4 Constraint Solver

Technically, a constraint solver is an algorithm for determining the satisfaction of a constraint or set of constraints. In most cases, this algorithm for determining constraint satisfaction also results in solution as a by-product and thus can be used to generate answer for the constraint to be solved. Based on the above description, it is obvious that the type of constraint solver involved in constraint logic programming language is
domain dependent. The constraint domain underlying a CLP will decide the type of algorithm to be applied for constraint solving and thus the class of constraint solver. For instance, constraint domain with finite domain value, i.e. variable with a finite number of values, enumerate through all the valuations available for the constraint and test whether there is a solution. The main drawback of the enumeration method is that its run-time complexity will increase exponentially even when there is a small increase in the number of variables. On the other hand, constraint domain with infinite number of valuations to check will normally require specific constraint solving technique specifically relevant to the constraint domain involved. A good example to illustrate this is the CLP(R) by Jaffar et.al. [Jaffar, 1992]. In his example, the constraint domain involved is real arithmetic and thus the solver applied here will adopt Gauss-Jordan elimination as the primary solving algorithm.

All constraint solvers exhibit a number of common features that are essential as well as desirable. Formally, a constraint solver can be complete or incomplete. Given a set of constraints as input to the constraint solver, a complete server is one that will return either true or false to indicate that the constraints are consistent with one another or otherwise. An incomplete solver is one which in addition to true or false, it will also return an answer unknown to indicate that the solver is unsure of whether the set of constraints are consistent (satisfiable) or not.

2.6.2.5 The CLP Scheme and Motivation

The constraint logic programming scheme [Jaffar, 1986] has defined a family of languages, i.e. constraint logic programming language is parametric in term of the choice of the underlying constraint domain. As a programming scheme, it serves as a framework
that provides standard definitions for rules, derivations and evaluation. All the goal evaluation mechanism will remain the same for all constraint logic programming language and is independent of the constraint domain involved. Thus each CLP scheme will come with a particular constraint domain, a standard goal evaluation mechanism (a depth-first left-right search) and is supported by a specific constraint solver. Constraint programming language created from CLP scheme differs significantly from the traditional programming language mainly due to the evaluation mechanism incorporated into the CLP programs. This impressive and convincing result produced from attempt to combine "constraint + logic programming" has become the inspiration of this research work and subsequently motivated the attempt to marry constraint with an agent programming language.

2.7 Conclusion

This chapter provides the relevant background knowledge that has motivated the research work to be discussed in the subsequent chapters. Two broad categories of computing technology (agent-based technology and constraint computation) were explored in detail. Agent-based approach has been recognized as one of the possible solution to the demand for flexible and intelligent computing power. Agent with BDI architecture is selected as the model to illustrate the intelligent attributes that can be leveraged on to offer an improved application framework. Constraint processing as an arbitrary choice of computing technique to be integrated into the agent framework is hoped to create an effective, efficient agent and constraint based computing environment.
Chapter 3

ConstraintAgentSpeak

The syntax and semantics of ConstraintAgentSpeak extend those of AgentSpeak(L) in a manner similar to the constraint logic programming (CLP) framework extending classical logic programming. In this chapter, section 3.1 discusses the syntax of ConstraintAgentSpeak, section 3.2 describes the operational semantics and section 3.3 provides a brief discussion on the proof theory adapted from Rao [Rao, 1996] that translates an agent from one state to another.

3.1 Syntax

The formal language of ConstraintAgentSpeak consists of variable, terms, constants, function symbols, predicate symbols, connectives, quantifiers and other symbols such as goal (!, ?), implication (←) and addition or deletion operator (+, -). The set of predicate symbols is partitioned into sets of constraint predicates and non-constraint predicates. The constraint domain $C$ characterises the set of predicate, function and constant symbols from which the primitive constraints are constructed as well as determines the solver $solv_c$ that is integrated into the framework of BDI agent architecture. Different choices of constraint domain neither result in changes to the BDI framework nor give rise to any syntactical changes to the ConstraintAgentSpeak. Instead the possibility of having different constraint domains incorporated into an agent framework provides the flexibility that enables the application of agent-based computing to different computing problems. As with the CLP scheme [Jaffar, 1994a], it is possible to define a generic agent
programming framework without making reference to a specific constraint domain. Our discussion in the remainder of this chapter will thus be parameterised by a choice of some constraint domain C. The following definitions provide the generic description on the components that collectively form the specification language of ConstraintAgentSpeak. A number of definitions for basic components such as base belief, goal, plan etc have been adapted from [Rao 1996] in order to maintain consistency on definition for these element.

**DEFINITION 3.1**

The set \( P \) of predicate symbols is partitioned into the sets of constraint and non-constraint predicate symbols.

If \( p \in P \) and \( (t_1, \ldots, t_n) \) is a vector of terms, then \( p(t_1, \ldots, t_n) \) is an atomic formula.

**DEFINITION 3.2**

A term is any constant, variable or n-ary function symbol applied to a sequence of terms.

**DEFINITION 3.3**

A belief is an assertion of the form \( b(t) \) where \( b \) is a belief predicate symbol, \( t \) is a vector of terms. A ground belief (base belief) will be of the form of \( b(t) \) where \( t \) contains no variables.

A constraint belief is an assertion of the form \( b(t) \leftarrow c_1(t_1), \ldots, c_n(t_n) \) in which \( b(t) \) is a belief. In general, \( c_i \) can either be constraint or non-constraint predicate symbol. In the case where there is no non-constraint predicate exist in the body of the constraint belief, \( c_i \) will be constraint predicate symbol (constraint symbol) in constraint domain and so is \( t_i \) term occur in the constraint domain. \( c_1(t_1), \ldots, c_n(t_n) \) are primitive constraints and are to be input for solving if they are considered pertinent as determine by the \( b(t) \), i.e. if \( b(t) \) is logical consequence to the agent's base beliefs.

The example on the multi-agent stock broking system (MABS) [Ooi. 1999] discussed in chapter 5 illustrates the integration of real constraint domain in ConstraintAgentSpeak. In the MABS example, a transaction agent has a base belief \( \text{client(john, valid)} \) which
indicates a valid client and a constraint belief fund(john, Sale, Bank, Pur, Qty, Rprice)

\[ \leftarrow \text{Sale}+\text{Bank}-\text{Pur} \geq (\text{Qty} \times \text{Rprice}) \] that ascertains the fund available for a client.

**Definition 3.4**

A goal is an assertion which can take one of the two possible forms: \( \!g(t) \) or \( \?g(t) \) where \( g \) is a goal symbol and \( t \) is a vector of terms.

**Definition 3.5**

If \( b(t) \) is a belief literal and \( \!g(t) \) or \( \?g(t) \) are goals, then \( \pm b(t), \pm \!g(t), \pm \?g(t) \) are triggering events or invocation condition of a plan.

Continue from the above example, the transaction agent may acquire from the user a goal to purchase certain quantity of shares at a particular price or to arbitrage a trade based on specific rate of return, written as \( \!^+\text{buy}(X, Stk, Qty, Rprice, Agentid) \) or \( \!^+\text{arbitrage}(X, Stk, Dqty, Dprice, Ret) \) respectively.

A transaction agent may also response to a goal (external trigger) or sub-goal (internal trigger) by executing an action to accomplish a task such as informing the client about a completed trade. In this instance, the action can be written as \( \text{notify}(X, \text{buy}, Stk, Dqty, Dprice) \) where \( X \) is an variable that can be unified with a particular client Id and the remaining variables represent information about the trade.

**Definition 3.6**

Action is a primitive activity of the form \( a(t) \), where \( a \) is an action symbol and \( t \) is a vector of terms.

An agent normally will be equipped with a ‘recipe’ which directs the behaviour of the agent based on a trigger and the existing context where an agent is located. The recipe will consist of a list of plans whereby from this same list, a most appropriate applicable plan will be invoked in response to the trigger as well as the agent context. The set of plans can exist in its most generic form not only with constraint(s) imply in the context.
but also with constraint(s) exist in the invocation condition (trigger) and action(s) in the body of the plan. An example of plan for a transaction agent that gets triggered when a user delivers a trigger to initiate a purchase transaction in its generic format as explained above is written as follow.

\[ +!\text{buy}(X, \text{Stk}, \text{Qty}, \text{Rprice}), \text{Sale}+\text{Bank}-\text{Pur} \geq 0 \] : \text{client}(X, \text{valid}) @_

\leftarrow \text{[submit(order)}, \text{Sale}+\text{Bank}-\text{Pur} \geq (\text{Qty} \times \text{Rprice})\text{]}.

However, to simplify implementation procedures and to provide easy-to-read program statement, the constraints associated with the invocation and action of a plan can be amalgamated into a form of constraint belief that places in the context of the plan. During the process of generating an acceptable applicable plan, the interpreter will perform the required constraint consistency validation if constraint beliefs are found in the context section of the plan (together with other context beliefs). Thus the above plan can be rewritten in its more efficient format as below:

\[ +!\text{buy}(X, \text{Stk}, \text{Qty}, \text{Rprice}) : \text{client}(X, \text{valid}) \& @_

\text{fund}(X, \text{Sale}, \text{Bank}, \text{Pur}, \text{Qty}, \text{Rprice}) @_

\leftarrow \text{submit(order)}].

With the following constraint belief exists as part of the belief for the agent.

\text{fund(john, Sale, Bank, Pur, Qty, Rprice)} \leftarrow \text{Sale}+\text{Bank}-\text{Pur} \geq (\text{Qty} \times \text{Rprice})

Based on the above format, formally an agent's plan can be defined as follow.

**DEFINITION 3.7**

If \( e_t \) is an invocation condition (trigger) in terms of goal or belief, \( \{b_1(t)\}, \ldots, \{b_n(s)\} \) are context beliefs, and \( h_1, \ldots, h_m \) are goals (sub-goals) or actions, then a plan will be in the following form:

\[ e_t : \{b_1(t)\}, \ldots, \{b_n(s)\} \leftarrow h_1; \ldots; h_m \]

A plan with some of the variables bound to terms is a plan instance or an intended mean. Again, the term here can be referred to term from constraint domain, which indicates the
explicit value pertaining to the problem domain. An intended mean that appears in the stack of an intention reflects the current mental attitude that determines the behaviour of the agent.

**DE**

**IN**

**ITION 3.8**

Additional connectives are used to indicate the special semantics associated with atomic formula occurs in different parts of the program clause. ‘!’ is to indicate an achievement goal, ‘?’ to indicate a query goal, ‘+’ or ‘−’ to represent trigger event operator and ‘&’ to indicate the sequencing of the formulae in the body of a program clause.

Based on the above definitions, essentially an **ConstraintAgentSpeak** program is specified using a finite set of **unique** Horn sentences (a conjunction of unique Horn clauses). Each Horn sentence is an improvised definite program clause in one of the following two formats.

**Format 1:**

\[ A \leftarrow L_1, L_2, \ldots, L_n \quad 3.1.1 \]

or

\[ A \vee L_1 \vee L_2 \vee \ldots \vee L_n \quad 3.1.2 \]

**Format 2:**

\[ [A : C_1 \& C_2 \& \ldots \& C_m] \leftarrow B_1, B_2, \ldots, B_n \quad 3.1.3 \]

or

\[ [A \wedge C_1 \wedge C_2 \wedge \ldots \wedge C_m] \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n \quad 3.1.4 \]

Syntactically, \( A \) and \( [A : C_1 \& C_2 \& \ldots \& C_m] \) are the heads whereas \( L_1, L_2, \ldots, L_n \) and \( B_1, B_2, \ldots, B_n \) are the bodies of the clause. \( A, C_i, \) and \( B_i \) are literals of atomic formulae whereas \( L_i \) will be restricted to the form of constraint formula (primitive constraint literal) from the constraint domain involved. The use of this additional type of literal \( (L_i) \) in the improvised Horn sentence serves to bestow more expressive power to traditional Horn clause (thus partially offset the loss of expressiveness resulted from the restriction to only Horn clauses). The semantic of the implication connective \( (\leftarrow) \) in form 1 (3.1.1)
has been slightly modified to cater for the non-negation nature of the primitive constraint literals. Essentially, clauses appear in format 1 are in fact the user-defined constraint rules for the constraint domain and will be used in the derivation process of constraint solving as well as in the specification of an agent’s plan context.

The symbol ‘:’ (colon) in 3.1.3 represents an improvised form of representation for the head of the definite Horn sentence. The inclusion of ‘:’ symbol significantly extends the expressiveness of the sentence by allowing to consider for more positive literals (hence more expressive specification of agent’s plan and more precise decision making process) during the resolution process but does not degrade the simple efficiency of processing using Horn clause. The symbol ‘:’ syntactically replaces the first connective ‘\(\land\)’ in the head of the definite clause. However, semantically the symbol is implying a ‘\(\land\)’ connective as such usage of ‘:’ instead of ‘\(\land\)’ will enable the notions of relevant and applicable plan to be displayed more vividly during implementation. The resulting syntax will become more oriented to the traditional style of imperative programming (whereby the context section in the head and the body will actually work algorithmly equivalent to an ‘If –Then’ control construct). Such grammatical framework will provide better comprehension to most programmers as it can be viewed and treated at least very close to (if not synonymous to) the conventional ‘If-Then’ construct in conventional imperative programming.

There are three types of program specifications as categorized below:

1. **Base belief**: specified in format 2 with \(m = 0\) and \(n = 0\);

2. **Plan**: specified in format 2 with \(m \geq 1\) and \(n \geq 1\);

3. **Constraint belief**: specified in format 1.
The goal literal which occurs in $A$ and $B_i$ in the form of atomic formula is enriched with additional connective symbols (‘!’ and ‘?’) which each of them contributes specific semantic to the various component of the sentence.

3.2 Operational Semantics

This section presents the model of operation of $ConstraintAgentSpeak$. A canonical operational semantic is derived for the cycle of operation of an agent. Each cycle of operation is made up of the inference procedure and the agent-execution. Inference procedure is an operation that consists mainly of determining the relevant and applicable plan of an agent. Embedded within this operation is the constraint solving that serves to augment the existing unification process with the intention to provide: wider scope of applicability, better precision in term of belief specifications and deployment of specialised efficient problem solving or solution generation technique. Agent-execution (or more precisely, intention-execution) is the execution of an agent’s intention, which made up of a combination of two possible courses of action: a sub-goal or a primitive action. This section is divided into 2 subsections in which the first subsection describes the operational semantics for constraint solving while the second outlines the same for cycle of operation of an agent.

3.2.1 Operational Semantics for Constraint Solving

The operational semantics of constraint solving within the execution context of $ConstraintAgentSpeak$ is strictly restricted to determining the satisfiability for a collection of primitive constraints and work synergistically to enhance the practical reasoning efficiency of the BDI framework. In short, the primary role of constraint processing is for problem solving rather than logical reasoning. This is achieved through its role in the
augmentation and representation of the base beliefs of an agent. It complements the inference mechanism employed to control the operation model with additional data structures that enable more expressive knowledge representation. A notable difference when compared to constraint solving in constraint logic programming are the extent of non-determinism. Constraint solving in the context of an agent encounters less non-determinism due to the following two rationales which also form the basis of operation within the solving process.

3.2.1.1 Choice of Reduction Rule

Within the context of practical reasoning, the choice of which constraint belief (rule) to be used for reduction of user-defined formula in a plan's context (a main source of non-determinism and thus the possibility of infinite derivation in constraint logic programming) does not arise. The heuristic factor that is instrumental to rule ordering in constraint logic programming also does not apply. The only determinant that is significant is the right specification of base belief (constraint belief). Such advantage is due to the fact that conceptually the user-defined constraint used for reduction is a constraint belief, which is also a base belief. Corollary no two constraint beliefs will have the same head. This is an acceptable proclamation because base belief is knowledge representation for information known to the agent and thus no agent will logically own two exactly identical base beliefs in its belief set.

**Definition 3.9**

*Given a base belief $A$ encounters in the context of a plan. If $A \leftarrow L$ in which $L = \{L_1, L_2, \ldots, L_n\}$, is the constraint belief to be used for the reduction of $A$, then it is the one and only one constraint belief that will be applicable for this process.*
In constraint solving, assuming $c$ is the current constraint (in constraint store). At each reduction step, a literal $L_i$ (primitive constraint) is selected according to a left-to-right selection rule and append to the constraint store for consistency checking.

**DEFINITION 3.10**

Given $L_1 \land L_2 \land ... \land L_n$ is a conjunction of literals reduced from a constraint belief. A left-to-right literal selection rule is a function that returns one literal $L_i$ at a time from the conjunction begins from the first literal on the left until a terminating signal (fail reduction) is received or there are no more literal is available (an $\emptyset$ is encountered, a successful reduction). If the tuple $(L|l)$ is the literals configuration during each step of the reduction process, then the sequence of configurations for the entire reduction process is

$$\langle o_1 | l_1 \rangle \Rightarrow \langle o_2 | l_2 \rangle \Rightarrow \ldots \Rightarrow \langle o_n | l_n \rangle$$

in which $n = I$ and it is a successful reduction whereby

$$\langle o_1 \text{ is } L_1 \land L_2 \land \ldots \land L_n \text{ | } l_1 \text{ is } \emptyset \rangle \Rightarrow \langle o_2 \text{ is } L_2 \land \ldots \land L_n \text{ | } l_2 \text{ is } L_1 \rangle \Rightarrow \ldots \Rightarrow \langle o_n \text{ is } \emptyset \text{ | } l_2 \text{ is } L_1 \land L_2 \land \ldots \land L_n \rangle$$

Thus, if $c$ is the current constraint store, the two optional outcomes from each such reduction steps are:

1. If $\text{solv}( c \land L_i ) = true$, then the new constraint store $c'$ will be $c \land L_i$. The next reduction step follows if $L$ is not empty. Otherwise solving will be successfully terminated and $A$ is returned as a sound belief (true belief) of the agent for continuation of execution.

2. If $\text{solv}( c \land L_i ) = false$, $c$ is reduced to $\emptyset$ and $A$ is returned as a false belief.

The entire solving process is a sequence of reduction-consistency test steps. When the solving process is completed and successful, answer in the form of answer constraints will be retained in the constraint store given by $\exists_{\text{vars}(c)} c$, with respect to the set of variables in interest returned by $\text{vars}(c)$.
3.2.1.2 Renaming of Rule

Renaming of rule refers to the renaming of variables occur during the reduction of a user-defined constraint in constraint logic programming. Pertaining to this issue in ConstraintAgentSpeak is the fact that every single constraint solving process [occur within a session (during evaluation of a plan context) initiates by a new external event and ends when the intention's plan stack is empty] is treated as a new operation in the sense that a fresh set of constraints will be applied. Thus no renaming of variables is necessary here as compared to the case in CLP. For the same reason, the constraint store of an agent will be started fresh (from $\emptyset$) in each constraint solving session. The resulting solutions or answer constraints will only be applicable up to the last executing formula, which is a sub-goal or primitive action in the body of the current executing (current) plan that initiates the constraint solving process. Such feature is an emphasis of constraints being deployed as part of the beliefs of an agent in its belief set to provide more precise belief specification. This is especially true since the belief is most likely to be changed frequently as the state of an agent changes when plans are executed, thus the constraints that form part of the entire base beliefs will be inclined to change as well. Accordingly a fresh set of constraints will need to be accessed each time a constraint solving operation is invoked.

3.2.2 Operational Semantics for ConstraintAgentSpeak Cycle of Operation

The entire operation model of ConstraintAgentSpeak essentially consists of two modes of operations. One mode follows after the other within one single cycle of operation. First mode of the operation cycle will begin in response to trigger event returned by an event selection function and end up with a selected (acceptable) applicable plan or an empty set
of plan (no applicable plan). An acceptable application plan eventually transform into an intended means (instantiated plan) that become an agent’s intention. The next mode follows with the execution of an intention returned by the intention selection function. In real life run-time environment, it is possible that the two modes of operation are running concurrently but each will bear little logical relevance with one another. The following presents the operational semantics associated with entire cycle of operation. In doing so, relevant significance for component entities of a BDI agent will be explained and illustrated pictorially if applicable.

**DEFINITION 3.11**

An agent is specified by a tuple \((E, B, C, P, I, A, SE, SÎ, Sf, DC)\). \(E\) is a set of events, \(B\) is a set of base beliefs, \(C\) is a set of constraint beliefs, \(P\) is a set of plans, \(I\) is a set of intentions and \(A\) is a set of actions.

\(SE, SÎ,\) and \(Sf\) are the selection functions for events set \((E)\), applicable plans set \((Î)\) and intentions set \((I)\) respectively. Each of these selection functions is responsible to pick an element from their respective set and submit the element for processing. \(DC\) is the constraint domain integrated into the BDI framework.

Figure 3.1 adapted from [d’Inverno. 1998] provides a pictorial representation of a ConstraintAgentSpeak agent at a particular instance of time. The state diagram depicts the data structures that are used to store the knowledge base, the working plans and actions associated with an agent. Specific functions \((SE, Sf)\) are assigned to manage and manipulate the contents of data structures (events and intentions) that are not handled directly by the BDI engine and constraint engine of the ConstraintAgentSpeak interpreter.
Constraint beliefs (dark-grey oval) is the new data structures in addition to the original set-up of AgentSpeak(L) without the constraint processing capabilities. The beliefs and constraint beliefs are both base (ground) belief atoms of the agent whereby the constraint belief is extended with constraint expression (literals).

The definitions in section 3.1 and section 3.2.2 have defined the mental components that are illustrated in the above figure. Collectively, they describe a ConstraintAgentSpeak agent and the state of an agent at any instance during run-time. Note that in ConstraintAgentSpeak, the terms occur in the constraint domain are allowed to occur in the specification of belief, goal and action literal. Such grammar rule will enable the result generated during constraint solving to be accessible by the BDI reasoning framework and thus can be applied to construction of belief that contributes to improvement on the sensitivity of an agent.

**DEFINITION 3.12**

*E is a set consists of events. Each event is a tuple of \((e, i_e)\), where \(e\) is a triggering event and \(i_e\) is an associated intention that generates the triggering event. If \(i_e\) is a null intention (null value), then \(e\) is an externally triggered event. Otherwise, \(e\) is an internal event that is a sub-goal that occurs in the body of a plan.*
DEFINITION 3.13

I is a set of intentions whereby each intention is a stack of partially instantiated intended means or plan instances. Every intention is assigned with an intention status indicator to denote whether the intention is currently in active status or suspend status. Intention can be represented as a tuple of non-empty sequence of plan instances and status, \( I = (\{ p_1 || p_2 || ... || p_n \}, \text{Sts}_i) \) where \( p_i \) is a plan instance, \( p_n \) is the top of the stack and \( p_1 \) is the bottom of the stack. \( \text{Sts}_i \) is the status of the intention. A null intention is where the head or top of the plan stack is null.

By referring to the plans (as shown below) use to process incoming appointment request in the sample program for IschAgents presented in chapter 5,

\[
\begin{align*}
+!\text{inrequest}(\text{Inreqid, Fromagent, Person, Month, Year, Date, Slot}) & : \emptyset_\_ \\
& \text{valid(Person)} \& \emptyset_\_ \\
& \text{registered(Fromagent)} \emptyset_\_ \\
& \leftarrow \text{newbelief(\"apptrequest(\",\text{Inreqid, Fromagent,\")\")\")} \& \emptyset_\_. \\
& +!\text{appointreq}(\text{Inreqid, Fromagent, Month, Year, Person, Date, Slot}). \\

+!\text{appointreq}(\text{Inreqid, Fromagent, Person, Month, Year, Date, Slot}) & : \emptyset_\_ \\
& \text{currentmth(Month)} \& \emptyset_\_ \\
& \text{currentyear(Year)} \& \emptyset_\_ \\
& \text{appointperson(Person, Date, Slot)} \emptyset_\_ \\
& \leftarrow \text{display(\text{Inreqid, Fromagent, Listsoln})} \& \emptyset_\_. \\
& \text{genproposeappt(\text{Inreqid, Fromagent, Listsoln})}. \\
\end{align*}
\]

the following trigger

\[
+!\text{inrequest(req1, agent2, jane, jun, 1999, Date, Slot)}
\]

will generate an intention (INTENTION 1) with stacks of intended means as follow (Figure 3.2). Takes note that the latest plan (Intended Mean 2) is stacked on top of the plan (Intended Means 1) that triggers it.
3.2.2.1 Inference Procedure

The notion of relevant and applicable plan lays down the principal requirement that establishes the operative design in this mode of operation. The ultimate objective of the inferencing task is to derive from a set of plans available: firstly, a set of relevant plans $R$ and finally an acceptable applicable plan $\hat{A}$ that will respond to the trigger event. There are basically two sub-operations (genrelplan and genapplplan) taking place during an inference procedure and there are two selection functions ($S_E$, $S_O$) involved with these two sub-operations.

3.2.2.1.1 Generate Relevant Plan Set (genrelplan)

The inference procedure is initiated by an event trigger $r$ that has been picked by $S_E$ from $E$. $r$ is the triggering event that will be used to unify with the invocation condition of as many plans as possible. The set of plans where the invocation conditions so unified are termed as relevant plans $R$. The unifier that enables the above unification to take place is the relevant unifier.

Figure 3.2 An example of an intention with two intended plans.
**DEFINITION 3.14**

Let the event returned by $S_E$ be $S_E(E) = (r, i_e)$ and plan $p$ be $e_i : b_1, ..., b_n \leftarrow h_1; ...; h_m$.

Then $p$ is a relevant plan with respect to $(r, i_e)$ if there is a most general unifier $\sigma_r$ and $r\sigma_r = e_i\sigma_r$. $\sigma_r$ is the relevant unifier for $(r, i_e)$.

**3.2.2.1.2 Generate Applicable Plan (genapplplan)**

The relevant plans set from above is passed over to this sub-operation for generation of applicable plans. Here, each relevant plan is tested to determine whether the plan is satisfied with respect to the agent's current beliefs. This is done in a 2 steps sequence as follow:

1. Apply the relevant unifier to the context beliefs of the relevant plan.

2. Derive a correct answer for the resulting context beliefs after 1. Such that the context is an acceptable logical consequence of the agent's base belief set.

The subset of relevant plans that have a correct answer obtained from step 2 above will become applicable plans set $\mathcal{O}$. The composition of relevant unifier and correct answer is termed as the applicable unifier of $(r, i_e)$.

In ConstraintAgentSpeak, a notion of acceptable logical consequence is introduced and is explained below. With the integration of constraint computation, step 2 of the above is augmented with extending deliberation to the relevant constraint domain if and only if a particular context belief needs to be reduced to a more expressive form for more precise evaluation. In doing so, a primitive constraint literal is selected one at a time and handed over to a constraint store for constraint solving. After all the primitive constraints reducible from the context belief have been submitted for solving and the final constraint store is in consistent state with respect to the constraint theory and its intended
interpretation i.e. there exist a set of solution for the constraints in constraint store. the pertinent context belief will become an **acceptable logical consequence**. An **unacceptable logical consequence** occurs when a context belief is a logical consequence of the head of an agent's constraint beliefs but returns a value of \textit{false} by the constraint solver. In this case the entire set of context beliefs of a plan will be rejected and thus the plan will not be accepted as an applicable plan.

**Definition 3.15**

Let $B$ be the base belief set and $b_i$ be a context belief which is also a logical consequence of $B$. Given $b_i \leftarrow l_1, l_2, \ldots, l_n$ and if $\text{solve}(l_1 \land l_2 \land \ldots \land l_n) = \text{true}$, then $b_i$ is an acceptable logical consequence. Otherwise $b_i$ is an unacceptable logical consequence.

**Definition 3.16**

Let $B$ be the base belief set and plan $p$ be $e_i : b_1, \ldots, b_n \leftarrow h_1, \ldots, h_m$. If there is a relevant unifier $\sigma$, and there is a correct answer substitution $\theta$ such that $\sigma \theta (b_1 \land \ldots \land b_n)$ is an acceptable logical consequence (logical consequence) of $B$, then $p$ is an applicable plan and $\sigma \theta$ is the applicable unifier for $(r, i_x)$.

From a set of applicable plans $\hat{O}$ generated for the triggering event, the applicable plan selection function $S_O$ will return an applicable plan $\hat{A}$ to form the intended mean (by applying the applicable unifier to the body of the selected applicable plan $\hat{A}$) of the trigger. Depending on the type of the event, two options are available for the subsequent course of action. Given an initiating event $(r, i_x)$:

1. if $i_x$ is null, then the event is an **external event**. A new intention will be created for this event and appended to the intention set $I$. The new plan instance will be pushed on top of this new intention.

2. if $i_x$ is not null, the event is an **internal event** originated from a sub-goal of an existing intended mean in one of the intentions in $I$. In this case, the new plan
instance will be pushed on top of the intention (stack) that triggered this internal event.

**DEFINITION 3.17**

Given $\hat{O}$ the set of applicable plans, $p$ the applicable plan returned by $S_O(\hat{O})$ and $\rho$ the applicable unifier. Let the initiating event be $\langle r, i_e \rangle$ and $p$ is $e_i : b_1, ..., b_n \leftarrow h_1; ..., h_m$. If $i_e$ is a null intention, then $i_{\text{new}} = \langle \{ e_i : b_1, ..., b_n \leftarrow \rho(h_1; ..., h_m) \}, \text{Active} \rangle$ and $i_{\text{new}} \in I$ where $i_{\text{new}}$ is the new intention.

If $i_e$ is NOT a null intention and $i_e$ is $\langle \{ p \} \parallel p_2 | \parallel ... | p_n \}, \text{Suspend} \rangle$ then $i_e = \langle \{ p \} \parallel p_2 | \parallel ... | p_n \} e_i : b_1, ..., b_n \leftarrow \rho(h_1; ..., h_m) \}, \text{Active} \rangle$ and $i_e \in I$.

**3.2.2.2 Agent-Execution (Intention-Execution)**

Agent-execution follows after the generation of a plan instance for a trigger event and subsequently updating of an intention with the new plan instance. This mode of operation features on the execution of a current intention (with an active status) which was selected and returned by the intention selection function $S_I$. For a selected intention, *executing plan* is the first plan in the plan stack (top of the stack) and *executing formula* is the first formula occurs in the body of the *executing plan*. Execution commences with the *executing formula* and follows one of the following three patterns. The type of formula encountered in the *executing plan* determines a particular pattern of execution.

1. If the formula is an achievement goal, a sub-goal trigger event is created and an event with the sub-goal trigger is appended to the events set for processing.

**DEFINITION 3.18**

Given $i$ is the intention returned by $S_I(I)$ from the intention set $I$. If $i$ is $\langle \{ p \} \parallel p_2 | \parallel ... | p_n \} e_i : b_1, ..., b_n \leftarrow 'g(t) ; h_1 ; ..., h_m \}, \text{Active} \rangle$, then intention is deemed to have been executed if and only if a new event $e_{\text{new}}$ is generated where $e_{\text{new}}$ is $\langle +'g(t), i \rangle \in E$ and $i$ is updated to $\langle \{ p \} \parallel p_2 | \parallel ... | p_n \} e_i : b_1, ..., b_n \leftarrow 'g(t) ; h_1 ; ..., h_m \}, \text{Suspend} \rangle$. 

62
2. If the formula is a query goal, the goal will be used to obtain a most general unifier (mgu) that will unify the goal with the existing base belief set. In the process of generating the mgu, the notion of acceptable logical consequence (described in section 3.2.2.1.2) is applied if the goal formula is unified with a constraint belief that can be reduced to one or more primitive constraints.

**DEFINITION 3.19**

Given  is the intention return by  from the intention set  and  is the base belief set.

If  is \(\{p_1, p_2, \ldots, p_t\} \cup e : \delta_1, \ldots, \delta_t \leftarrow B \vdash : h_1, \ldots, h_n\). Active then intention is deemed to has been executed if and only if there is a mgu  such that \(B \equiv P_g (x_1)\) and  has been updated to \(\{p_1, p_2, \ldots, p_t, e : \delta_1, \ldots, \delta_t \leftarrow \Psi_h, \ldots, \Psi_m\}. Active\).

If  is \(\{p_1, p_2, \ldots, p_t\} \cup e : \delta_1, \ldots, \delta_t \leftarrow B \vdash : h_1, \ldots, h_n\). Active and  \(l_1, l_2, \ldots, l_n\) then intention is deemed to has been executed if and only if there is a mgu  such that \(B \equiv P_g (x_1)\) solved \(l_1, l_2, \ldots, l_n\) = true and  has been updated to \(\{p_1, p_2, \ldots, p_t, e : \delta_1, \ldots, \delta_t \leftarrow \Psi_h, \ldots, \Psi_m\}. Active\).

3. If the formula is an action, the action will be added to the queue of actions to be performed.

**DEFINITION 3.20**

Given  is the intention return by  from the intention set  and  is the set of action. If  is \(\{p_1, p_2, \ldots, p_t\} \cup e : \delta_1, \ldots, \delta_t \leftarrow a : h_1, \ldots, h_n\). Active then intention is deemed to has been executed if and only if \(a : h_1, \ldots, h_n\) is appended to  such that  \(a : h_1, \ldots, h_n\). Active.

The above definitions and descriptions present the operational semantics for the core activities of inference procedure and intention execution. There are other complementary operational details that have not been covered by the above core operations and will be...
described below to complete the task of providing a thorough operational semantics for this agent programming language.

The additional details considered here are essential for the completeness of the overall executing scheme. There will be three supporting sub-operations discussed here and formal definitions for the respective executing processes will be composed.

1. The first sub-operation is the execution control after the current intended mean of the active intention has been done or achieved. In this case, if there are still more plans in the intentions' plan stack, then \textit{nextplan} will be the second plan in the executing intention. As the current executing plan is triggered by the sub-goal of the \textit{nextplan} in the stack, thus any new bindings must be passed on to the \textit{nextplan} that carries the goal that triggers the current plan. This is performed by applying the relevant unifier of the current plan to the body of the \textit{nextplan} after removing the top formula (the triggering sub-goal) from the \textit{nextplan}.

**DEFINITION 3.21**

Given the current active intention \( i \) is \( \langle \{ p_1 \| p_2 \| \ldots \| p_n \| e_i : b_1, \ldots, b_n \leftarrow !g(t); h_1 ; \ldots ; h_m \rangle \), \text{Active} \). If \( \theta \) is the most general unifier (relevant unifier of current executing plan) of \( !g(t) \) and \( e_2 \), the top intended means is said to have been achieved if and only if \( i \) is updated to \( \langle \{ p_1 \| p_2 \| \ldots \| p_n \| e_i : b_1, \ldots, b_n \leftarrow h_1 ; \ldots ; h_m \rangle \), \text{Active} \).

2. The second sub-operation concerned with the generation of an event for an external trigger. This is an important aspect of an agent in the sense that it serves as the gateway or mechanism for communications and interactions. Therefore the two sources of external event that an agent has to deal with are: the environment and other agents. Environment-generated events concerned
with input or request from users whereas events triggered by other agents are in the form of interactions that are equivalent to method or message calls in an object-oriented model.

**Definition 3.22**

Given the current event set \( E = \{ e_1, e_2, ..., e_n \} \) where \( e_1 \) is the first event in the queue and \( e_n \) is the last event in the queue. If \( g(t) \) is a new trigger (new goal) from environment or another agent, then a new external event is generated if and only if the new event set is \( E = \{ e_1, e_2, ..., e_n, (+g(t), i_{null}) \} \) in which \( i_{null} \) is a null intention.

3. The third sub-operation is more appropriate and precise to be described as three separate autonomous functions rather than a single sub-operation. They are the three selection functions \( S_E, S_O, S_I \) that are responsible for shoveling or feeding new members into the core cycle of operation of an agent from the respective sets that each of them is responsible for.

**Definition 3.23**

Given the current event set \( E = \{ e_1, e_2, ..., e_n \} \) where \( e_1 \) is the first event in the queue and \( e_n \) is the last event in the queue. A new event is selected and returned for processing if and only if \( S_E(E) = e_1 \) and \( E = \{ e_2, ..., e_n \} \), where \( S_E \) is the event selection function.

The same definition (3.23) can be applied to selection of an applicable plan \( \hat{A} \) with respect to \( R \), a set of applicable plans using \( S_O \), the selection function.

The selection function for intention adopts a slightly extended operation model of definition 3.23 by taking into consideration the suspend status of an intention in the intention set.
**Definition 3.24**

Given $S_I$ is intention selection function and the current intention set $I$ is $I = \{i_1, i_2, i_3, ..., i_n\}$ where $i_1$ is the first intention in the queue and $i_n$ is the last intention in the queue.

If $i_1$ is $\langle\langle p_1||p_2||...||p_n\rangle\rangle$. Active, a new intention is selected and returned for execution if and only if $S_I(I) = \langle\langle p_1||p_2||...||p_n\rangle\rangle$. Active and $I$ is updated to $I = \{i_2, i_3, ..., i_n\}$.

If $i_1$ is $\langle\langle p_1||p_2||...||p_n\rangle\rangle$, Suspend and $i_2$ is $\langle\langle q_1||q_2||...||q_n\rangle\rangle$. Active, a new intention is selected and returned for execution if and only if $S_I(I) = \langle\langle q_1||q_2||...||q_n\rangle\rangle$. Active and $I$ is updated to $I = \{i_2, ..., i_n, i_1\}$.

Definition 3.22 has provided a basic model of operation (means) for a ConstraintAgentSpeak agent to communicate and interact with external entities. An extra edge can be gained by providing an extension to this basic model. The result of such extension is an agent with reactive behavior that can be adapted to handle computing problem that requires on-line reaction or even real-time respond. Definition 3.25 defines such an extension that can be incorporated into the basic data structure and operation model.

**Definition 3.25**

Given the current event set $E$ is $E = \{e_1, e_2, ..., e_n\}$ where $e_1$ is the first event in the queue and $e_n$ is the last event in the queue. A new internal trigger or a new trigger from environment or another agent is a tuple $\langle\langle e, \ldots \rangle\rangle$, where $e$ is the new goal and $\ldots$ is an associated indicator (which will be either urgent or normal) for the priority of the new goal.

If the new trigger is $\langle\langle \text{gett}, \text{urgent} \rangle\rangle$, then a new external event or internal event is generated if and only if the new event set is $E = \{\langle\langle \text{gett}, i_{\text{null}} \rangle\rangle, e_1, e_2, ..., e_n\}$ in which $i_{\text{null}}$ is a null intention.
If the new trigger is \( \langle !g(t), \text{normal} \rangle \), then a new external event or internal event is generated if and only if the new event set is \( E = \{ e_1, e_2, ..., e_n, \langle +!g(t), i_{null} \rangle \} \) in which \( i_{null} \) is a null intention.

3.3 Proof Theory

This section discusses the proof theory for \textit{ConstraintAgentSpeak} based on the transition system adapted from Rao [Rao, 1996]. The added component that needs to be highlighted is the set of constraint beliefs \( C_n \) as indicates in 3.3.2.

In the previous section, \textit{ConstraintAgentSpeak} has presented itself as a specification language that can be used to create an agent based on the BDI framework and 'equips' the agent with constraint handling capabilities. Within this framework, the beliefs (base beliefs and constraint beliefs) become the set of axioms that play a substantial role in determining the next state of an agent. The set of axioms will contribute partially to configure the mental attitudes (intentions) which eventually define the next possible transition state that an agent will evolve to. As a result, the beliefs of an agent will 'transform' accordingly where there are transitions of an agent's state. In accordance with such working context, the proof theory presented here is based on the 'Gentzen-like natural deduction' proof system [Ben Ari, 1993] [Thayse, 1988] that takes the following form:

\[
\frac{S_1, ..., S_n}{T_1, ..., T_n}
\]

The above \textit{Gentzen-like} notation represents that \( T_1, ..., T_n \) can be proved or inferred from \( S_1, ..., S_n \). The \textit{natural deduction} supports a logical reasoning of the form:

'Given an agent is in a particular state of mental inclination, such state will derive or infer the next state of mental attitude (inclination) of the agent'
Based on the above assertion, it is reasonable to say that the mental state of an agent will naturally become the most appropriate inference rule for itself. Given that such mental state can be adequately represented in a configuration of \( (E, B, C, I) \), in which \( E, B, C, I \) are as defined in definition 3.12. Hence a proof rule or transition rule base on the above notation (3.3.1) can be stated as follow:

\[
\frac{\langle E_n, B_n, C_n, I_n \rangle}{\langle E_{n+1}, B_{n+1}, C_{n+1}, I_{n+1} \rangle}
\]

where it is read or interpreted as: the state \( \langle E_{n+1}, B_{n+1}, C_{n+1}, I_{n+1} \rangle \) is inferred or derived from an earlier state of \( \langle E_n, B_n, C_n, I_n \rangle \).

A major benefit of the above form of inference rule is the ability to provide consideration on the instantaneous semantics of an agent at a particular instant during the occurrence of the inference process. Thus the derivation of an agent from one initial state to another substantially different state will involve the sequential application of a number of proof rule instances.

**Definition 3.26**

A proof in this context transforms a transition state given by the configuration above the line to a new transition state as given by the configuration below the line.

A proof rule instance is the transition rule at a particular instance, \( t \) in which \( \langle E_t, B_t, C_t, I_t \rangle \) is only valid for that instance. Thus for this proof rule instance, the next state of the agent will be \( \langle E_{t+1}, B_{t+1}, C_{t+1}, I_{t+1} \rangle \).

### 3.4 Conclusion

The previous sections have provided a list of definitions for the system and operational model of AgentSpeak. The operational semantic is discussed based on the cycle of operation for an agent. Every cycle of operation consists of two sub options: inference
procedure and agent-execution. Constraint processing is an integrated part of Inference Procedure. It is an addition to the original framework of AgentSpeak(L) and it provides enhancement in terms of constraint solving capability.
Chapter 4

Implementing *ConstraintAgentSpeak*

The chapter presents the implementation of prototype *ConstraintAgentSpeak*’s interpreter. The scope of implementation includes the standard phases involved in any typical programming language: *tokenizing, parsing and interpreting* of program statements coded using the *ConstraintAgentSpeak* language. Although it is a prototype implementation that provides the essential basics to demonstrate the idea of this thesis, however the amount of details covered in this chapter is significant enough to make the task of extending the language to a full-fledge version become easy and uncomplicated.

For instance, to make a *ConstraintAgentSpeak* agent more pragmatic, the following can be implemented to improve its usability.

- The tasks that can be handled by a *ConstraintAgentSpeak* agent can be extended by merely adding new program libraries to provide more executable actions.

- To cater for huge application whereby the number of beliefs of an agent is large and volatile, the base belief of an agent can be updated on-line based on a database or a real-time register via a special routine that can be called by the BDI engine during run-time.

The organization of this chapter is given as: Section 4.1 presents the architecture of the *ConstraintAgentSpeak*’s interpreter. Section 4.2 describes an overview of the
implementation and relations between the components given in 4.1. Section 4.3 gives an illustration of the behaviour of an agent created using the language.

4.1 *ConstraintAgentSpeak*’s Interpreter Architecture

This section provides a structural description of the *ConstraintAgentSpeak*’s interpreter.

Figure 4.1 Structure of *ConstraintAgentSpeak* Interpreter

Figure 4.1 depicts the structure of *ConstraintAgentSpeak* Interpreter. It also shows the constraint engine that is embedded into the BDI agent framework. The solver within the constraint engine is the main constraint-solving unit and is supported by data structures that play the role of intermediaries between the BDI engine and the constraint engine.

The arrows indicate the flow of information within the BDI engine. The scheduling mechanism for information flow within the BDI engine is the responsibility of the three selector functions (*Event Selector, Intention Selector and Applicable Plan Selector*). The figure shows all the essential components that are required to drive the functional behaviours of the reasoning mechanism. It also indicates the role of the constraint engine within the BDI framework, which is to support the reasoning process and extending the computation capabilities of an agent. The three boxes enclosed by dotted lines represent
the three possible outcomes that result from the execution of an intention. Primitive actions are actions that execute directly and deal directly with external entities (users or other agents) or updating agent's own belief set. Query goal is to validate existing beliefs of the agent by generating a correct answer substitution. Sub-goal is used to generate an internal event that is inserted back into the event queue for processing. The plan interpreter is the main processing unit that performs execution of the current intention picks by Intention Selector from the agent's intention set and determines the course of action for the agent. The event queue consists of triggers waiting to be selected for processing. The intention set is a set of applicable plan stacks that will be picked by intention selector for execution.

Both external and internal events enter the event queue asynchronously. The first event (implicitly the highest priority event) in the event queue is selected by the event selector for processing. In the current implementation, events are distinguished into two classes: Urgent and non-urgent (normal) events. Event with an urgent status indicator will be inserted into the head of the queue whereas event with normal status indicator will be appended to the tail of the queue. The system will then attempt to unify the selected event with the triggers of plans in the plan library. The set of plans whose triggers unify with the selected event is called the set of relevant plans. From the resulting set of relevant plans, an appropriate applicable plan (one with the most general unifier) is selected by the Applicable Plan Selector to determine the next course of action. During this applicable plan selection process, computation involved constraint processing (to generate solution to the problem variables) that cannot be handled efficiently by the BDI framework will be taken over by constraint solving. The selected applicable plan becomes an intention of the
agent and is placed in the intention queue waiting to be executed. Eventually when the intention pick by *Intention Selector* is due to be carried out by the plan interpreter and depending on the contents in the body of the plan that form the agent intention, the following three options (courses of action) are available:

- Executes primitive action
- Generates an internal event
- Validates existing agent's belief

Any sub-goal or new internal event generated from intention execution will be placed back in the event queue for processing and the entire execution cycle is repeated until the event queue is empty.

![Diagram](image.png)

Figure 4.2 Structure of a *ConstraintAgentSpeak*’s agent
In Figure 4.2, an overall structure of a *ConstraintAgentSpeak*'s runtime agent is presented. The heart of the architecture of the *ConstraintAgentSpeak*'s interpreter, which consists of the constraint engine and BDI engine, is represented by the bottom large rectangle in the above figure. The rounded rectangles on top of the interpreter are the external entities that an agent has to deal with and the static mental components (beliefs) of an agent. These external entities generate external events which enter the event queue. Interaction with other agents provides via messages to other agents which may appear as external events in their event queue.

Figure 4.3 provides the pseudocode of algorithm for the *ConstraintAgentSpeak* interpreter. The following explanations on the essential functions that form the core components of the interpreter will help to clarify their respective roles within the operation of the interpreter.

\( S_E \): Event selector function – Selects the first event from the *event set* that contains external as well as internal events waiting to be processed.

\( S_O \): Applicable plan selector function – Selects an applicable plan with the most general unifier (mgu) from a set of *applicable plans* derive from the set of relevant plans via function *genapplplan*.

\( S_I \): Executing Intention selector function – Selects the first intention from the *intention set* and submit the intention to plan interpreter for execution.

*genrelplan*: Relevant plans generator function – Generates a set of relevant plans based on the event returned by \( S_E \).

*genapplplan*: Applicable plans generator function - Generates a set of applicable plans based on the base beliefs and the set of relevant plans returned by *genrelplan*.
while (Event.Queue ≠ ∅) do

\( S_E \) select Event \( h_e \) from Event.Queue

\textit{genrelplan} generates set of relevant plan \( P_r \) for \( h_e \) with respective relevant unifier \( \theta_r \)

\textit{genapplplan} generates set of applicable plan \( P_a \) from \( P_r \) with respective correct answer substitution \( \sigma_a \)

\( S_O \) select an applicable plan \( P_o \) with the most general unifier \( mgu \) (composition of \( \theta_r \) and \( \sigma_a \)) from \( P_a \)

applies the \( mgu \) to body of \( P_o \) to generate \textit{intended} mean \( P_i \)

if (\( h_e \) is an external event) then

new Intention.Set = existing Intention.Set ∪ \( P_i \)

else

push \( P_i \) on top of Plan.Stack in Triggering.Intention

end while

while (Intention.Set ≠ ∅) do

\( S_I \) select First.Intention \( I_e \) from Intention.Set

while (Body of \( I_e \) ≠ ∅) do

\textbf{case} (First.Formula of Body of \( I_e \) = Achievement.Goal !g(t))

new Event.Queue = existing Event.Queue ∪ (+!g(t), \( I_e \))

suspend execution for \( I_e \)

break loop

\textbf{case} (First.Formula of Body of \( I_e \) = Query.Goal ?g(t))

generates correct answer substitution \( \theta_s \) for \( ?g(t) \)

applies \( \theta_s \) to remaining formulae in Body of \( I_e \)

remove First.Formula of Body of \( I_e \)

\textbf{case} (First.Formula of Body of \( I_e \) = Action a(t))

remove First.Formula of Body of \( I_e \)

new Action.Queue = existing Action.Queue ∪ a(t)

end while

end while

end while

Figure 4.3 Algorithm for ConstraintAgentSpeak interpreter
4.2 Implementation

When executing an agent program, the ConstraintAgentSpeak interpreter will take a ConstraintAgentSpeak program as input and carry out the necessary actions accordingly. In this context, the implementation amounts to creating a basic programming environment that allows the following phases to be accomplished in a sequential manner.

1. Create or code agent program.
2. Tokenize all program statements.
3. Parse all tokenized program statements.
4. Execute or run the parsed program.

Since step 1 above simply involves creating a text file containing the program code, we shall focus only on step 2 to 4. All of the interpreter code was implemented using Java (Sun's JDK1.1).

4.2.1 Tokenizing Program Statements

Tokenizing program statements is a lexical analysis process that involves isolating words or tokens in the original source program statement based on some pre-assigned grouping of characters to form words and concatenate the individual words into a proper program statement. This includes concatenating words that spread over more than one line into a proper, single line program statement. It is a straight-forward process that requires a standard routine program to pick up a group of characters and break down the group into a series of tokens based on some specific classification. Each of the token or word is separated from the other using space delimiter. The type of words or tokens that are found in a ConstraintAgentSpeak program can be classified into 16 separate categories as shown in table 4.1.
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTRL</td>
<td>Literal for predicate and term (Lowercase String)</td>
</tr>
<tr>
<td>LPRT</td>
<td>Left parenthesis '('</td>
</tr>
<tr>
<td>TVAR</td>
<td>Variable (single uppercase letter/title-case string)</td>
</tr>
<tr>
<td>RPRN</td>
<td>Right parenthesis ')'</td>
</tr>
<tr>
<td>SSUG</td>
<td>Syntactic sugar '←'</td>
</tr>
<tr>
<td>CONJ</td>
<td>Conjunction '&amp;&amp;'</td>
</tr>
<tr>
<td>STOP</td>
<td>Terminating period '.'</td>
</tr>
<tr>
<td>GOAL</td>
<td>Achievement or Test goal '!' or '?'</td>
</tr>
<tr>
<td>OPER</td>
<td>Addition or deletion of belief/goal '+' or '-'</td>
</tr>
<tr>
<td>CTXT</td>
<td>Context of plan's Head ':'</td>
</tr>
<tr>
<td>TERM</td>
<td>Term(s) of atomic formula/belief (Non-variable term)</td>
</tr>
<tr>
<td>ACTN</td>
<td>Action predicate of plan's body</td>
</tr>
<tr>
<td>CNXL</td>
<td>Continuation to next line '@_'</td>
</tr>
<tr>
<td>CONS</td>
<td>Numeric constant (double)</td>
</tr>
<tr>
<td>BOOL</td>
<td>Boolean value (true, false)</td>
</tr>
<tr>
<td>ROPR</td>
<td>Relational operator (&lt;, =, &gt;, ≥, ≤)</td>
</tr>
</tbody>
</table>

Table 4.1 Classification of token type

Base on the above classification, each *ConstraintAgentSpeak* program statement is tokenized into a string made up of concatenated tokens. The following *ConstraintAgentSpeak* program (Program 4.1) will be used to illustrate the behaviors that occur during each of the last three phases (the labelling numbers at the beginning of program statements are not part of the program).

```
1 yearmonth(1999, jun).
2 client(john, valid).
3 client(johan, valid).
4 currentweek(week10).
5 manager(janet,Day,Slot) ←(monday, slot1), (monday, slot4), (tuesday, slot3), (friday, slot6).
6 manager(jesse,Day,Slot) ←(tuesday, slot1), (tuesday, slot3), (friday, slot6).
7 +!appointment(Client,Mgr,Slot,Day,Week):client(Client, valid) & \_ currentweek(Week) & \_ manager(Mgr)
  ← confirm(Client,Day,Slot).
8 +!appointment(Client,Mgr,Day,Week):client(Client, valid) & \_ currentweek(Week) & \_ manager(Mgr)
  ← display(Day, Slot).
```

Program 4.1 Sample codes for appointment scheduling
Using the last plan statement (8) in the above Program 4.1, a sample output from the tokenizing process performed on the statement (plan 8) will be as follow. The resulting statement will be a single line with all the tokens concatenated. The token ‘@_’ which is used to represent continuation to next line is discarded from the tokenized statement.

Table 4.2 shows the classification that has been assigned to each of the token after the tokenizing and parsing process.

**Tokenized statement:**

```
+ ! appointment ( Client Mgr Day Week ) : client ( Client valid ) & currentweek ( Week ) & manager ( Mgr ) <- display ( Day Slot ) .
```

<table>
<thead>
<tr>
<th>Token</th>
<th>Token type</th>
<th>Token type</th>
<th>Token</th>
<th>Token type</th>
</tr>
</thead>
<tbody>
<tr>
<td>appointment</td>
<td>PRED/LTRL</td>
<td>()</td>
<td>LPRN</td>
<td></td>
</tr>
<tr>
<td>Client</td>
<td>TVAR</td>
<td>Mgr</td>
<td>TVAR</td>
<td></td>
</tr>
<tr>
<td>Day</td>
<td>TVAR</td>
<td>Week</td>
<td>TVAR</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>RPRN</td>
<td>:</td>
<td>CTXT</td>
<td></td>
</tr>
<tr>
<td>client</td>
<td>PRED/LTRL</td>
<td>valid</td>
<td>TERM/LTRL</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>CONJ</td>
<td>currentweek</td>
<td>PRED/LTRL</td>
<td></td>
</tr>
<tr>
<td>manager</td>
<td>PRED/LTRL</td>
<td>&lt;-</td>
<td>SSUG</td>
<td></td>
</tr>
<tr>
<td>display</td>
<td>ACTN/LTRL</td>
<td>Slot</td>
<td>TVAR</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>OPER</td>
<td>!</td>
<td>GOAL</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>STOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 Example of token classification

For tokens with more than one character, automaton algorithms [Epp, 1995] are used to determine and verify that the correct type of token is assigned during the tokenizing process. **PRED, TERM or ACTN** token type will be employing the same type automaton algorithm (LTRL) and their exact type will be determined based on their specific location within a statement. For instance, token with all lower case characters occur after a **LPRN** (left parenthesis) but before a **RPRN** (right parenthesis) will be evaluated as a **TERM** token.

Figure 4.4 and 4.5 illustrate the automata used to determine the token type for **TVAR**, **PRED, TERM and ACTN**.
4.2.2 Parsing Program Statements

Parsing ConstraintAgentSpeak program statement is a syntax analysis process with the intention to ensure that all program statements are constructed according to correct grammatical rules. There are 3 basic syntax rules need to be complied with and they are:

1. Rule for constructing base beliefs
2. Rule for constructing constraint belief (user-defined constraint belief)
3. Rule for constructing plan

Syntactically parsed statements are validated statements that are correctly coded with respect to the relevant syntax rule and the interpreter will be able to understand the code and act accordingly. During the parsing process, an error will be thrown if any of the program code contains an error and does not comply with the corresponding syntax rule. When this occurs, the interpreter will discontinue its processing and the program will have to be corrected.
The parsing of *ConstraintAgentSpeak* is carried out using *automaton* algorithms [Epp, 1995] that perform a series of token type checking based on the respective syntax rule whenever a new token is added to the already checked (validated) sequence of tokens. Figure B.1 to Figure B.4 in Appendix B illustrates the automata for the syntax rules given above. The current interpreter’s tokenizer and parser are able to handle constraint beliefs that involve constraint expression in finite constraint domain or linear and non-linear equations in real arithmetic domain (shown by automata in Figure B.2 & B.3 of Appendix B).

Figure 4.7-1 and Figure 4.7-2 below provide a formal description of the *ConstraintAgentSpeak* language syntax in the Backus-Naur Form (BNF).

The meta-symbols used in the above BNF representations are as follows:

<table>
<thead>
<tr>
<th>Meta-symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>is defined to be</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;something&gt;</td>
<td>&lt;something&gt; is to be replaced by its definition</td>
</tr>
<tr>
<td>something</td>
<td>work or token written in bold-face indicating an indivisible <em>ConstraintAgentSpeak</em> element allowing no further replacement.</td>
</tr>
</tbody>
</table>

Figure 4.6 Meta-symbols for BNF representations in Figures 4.7-1 and 4.7-2
Figure 4.7 - 1 BNF declaration for basic elements used in ConstraintAgentSpeak program
Figure 4.7 - 2 BNF syntax representation for statements (belief, constraint belief and plan) used in ConstraintAgentSpeak program
Using the same example of tokenized statement in 4.2.1, a step by step simulation parsing is demonstrated with respect to automaton in Figure B.4 of Appendix B.4 (plan automaton). The entire parsing process for this example will be illustrated using a threecolumn table in Table 4.3. The first column represents the actual token itself, the second column represents the type of token in column one and column three is the corresponding state in the automaton after the token input. By referring to Figure 4.10, the plan statement is said to be a valid, syntactically correct statement if the final state in the last row of Table 4.3 is $S_7$. The token type checking begins from the first row in the table (which is the first token in the tokenized statement) and follows the automaton to move from state to state. An arrow moving from an existing state (except at the initial state where the first arrow enters the state is not from any other previous state) to the next represents the input of a new token. If the input of a new token ends up in a valid state (any appropriate state from $S_0$ to $S_7$) following the arrow in the automaton, the checking process continues. Otherwise an error exception is generated and the parsing process will be terminated. Only when it comes to the last token and the end-state is $S_7$, it is declared that the entire statement is a valid statement. The same type of evaluation strategy is applied to the parsing of other ConstraintAgentSpeak program statements based on different syntax rules with each rule follows a unique automaton to implement the evaluation.

<table>
<thead>
<tr>
<th>Token</th>
<th>Token Type</th>
<th>Automaton State</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL STATE</td>
<td>OPER</td>
<td>$S_0$</td>
</tr>
<tr>
<td>+</td>
<td>GOAL</td>
<td>$S_1$</td>
</tr>
<tr>
<td>!</td>
<td>PRED/LTRL</td>
<td>$S_2$</td>
</tr>
<tr>
<td>Appointment</td>
<td>LPRN</td>
<td>$S_3$</td>
</tr>
<tr>
<td>(</td>
<td>TVAR</td>
<td>$S_4$</td>
</tr>
<tr>
<td>Clt</td>
<td></td>
<td>$S_5$</td>
</tr>
<tr>
<td>Mgr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Day</th>
<th>TVAR</th>
<th>$S_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wk</td>
<td>TVAR</td>
<td>$S_2$</td>
</tr>
<tr>
<td>)</td>
<td>RPRN</td>
<td>$S_3$</td>
</tr>
<tr>
<td>;</td>
<td>CTXT</td>
<td>$S_4$</td>
</tr>
<tr>
<td>Client</td>
<td>PRED LTRL</td>
<td>$S_5$</td>
</tr>
<tr>
<td>(</td>
<td>LPRN</td>
<td>$S_6$</td>
</tr>
<tr>
<td>Clt</td>
<td>TVAR</td>
<td>$S_7$</td>
</tr>
<tr>
<td>Valid</td>
<td>TERM LTRL</td>
<td>$S_8$</td>
</tr>
<tr>
<td>)</td>
<td>RPRN</td>
<td>$S_9$</td>
</tr>
<tr>
<td>&amp;</td>
<td>CONJ</td>
<td>$S_{10}$</td>
</tr>
<tr>
<td>Currentweek</td>
<td>PRED LTRL</td>
<td>$S_{11}$</td>
</tr>
<tr>
<td>(</td>
<td>LPRN</td>
<td>$S_{12}$</td>
</tr>
<tr>
<td>Wk</td>
<td>TVAR</td>
<td>$S_{13}$</td>
</tr>
<tr>
<td>)</td>
<td>RPRN</td>
<td>$S_{14}$</td>
</tr>
<tr>
<td>&amp;</td>
<td>CONJ</td>
<td>$S_{15}$</td>
</tr>
<tr>
<td>Manager</td>
<td>PRED LTRL</td>
<td>$S_{16}$</td>
</tr>
<tr>
<td>(</td>
<td>LPRN</td>
<td>$S_{17}$</td>
</tr>
<tr>
<td>Mgr</td>
<td>TVAR</td>
<td>$S_{18}$</td>
</tr>
<tr>
<td>)</td>
<td>RPRN</td>
<td>$S_{19}$</td>
</tr>
<tr>
<td>←</td>
<td>SSUG</td>
<td>$S_{20}$</td>
</tr>
<tr>
<td>Display</td>
<td>PRED LTRL</td>
<td>$S_{21}$</td>
</tr>
<tr>
<td>(</td>
<td>LPRN</td>
<td>$S_{22}$</td>
</tr>
<tr>
<td>Day</td>
<td>TVAR</td>
<td>$S_{23}$</td>
</tr>
<tr>
<td>Slt</td>
<td>TVAR</td>
<td>$S_{24}$</td>
</tr>
<tr>
<td>)</td>
<td>RPRN</td>
<td>$S_{25}$</td>
</tr>
<tr>
<td>.</td>
<td>STOP</td>
<td>$S_{26}$</td>
</tr>
</tbody>
</table>

Table 4.3 Simulation for statement parsing (based on automaton in Appendix B Figure B.4)

### 4.2.3 Executing Parsed Program

As noted earlier, the ConstraintAgentSpeak framework defines a class of languages parameterized by the choice of constraint domain. The ConstraintAgentSpeak interpreter is similarly parameterized by a choice of constraint domain which in turn determines what constraint solver (constraint engine is used). A brief recap of constraint-based computation: in real-life computing situation dealing with constraint, the world of constraint can be divided into two broad classifications - The finite domain (FD) constraints and the non-finite domain constraints. A classic example of a system for programming with constraint over non-finite constraint is the CLP($\mathcal{R}$) language.
developed by Jaffar et.al. [Jaffar, 1994a]. In CLP(R), the specialised constraint solving techniques involve a combination of linear programming (Gauss-Jordan substitution) and the Simplex algorithm using canonical form of equation representations.

4.2.3.1 Constraint Processing

The current implementation of ConstraintAgentSpeak interpreter utilises the finite domain constraint solvers provided by the Java Constraint Library\(^1\) (JCL) [Bruchez, 1996]. The constraint solvers included in JCL support only binary constraints. However, this has not taken away from the generality of the system since CSPs with constraints of arbitrary arity can be uniformly translated into equivalent binary CSPs.

The Java Constraint Library provides a number of constraints solving techniques ranging from the simple backtracking to the relatively complex forward checking with full arc consistency. The user is allowed to select the solving technique to be used when performing constraint solving processing. In this implementation, the solving algorithm that has been incorporated into the constraint engine is the simple backtracking algorithm. It is important to note that although a finite domain binary constraint solver was used in the current implementation of the interpreter, other constraint solver (such as solver on the domain of reals) could be used in its place, with only minor changes to the syntax of the language.

The constraint engine as depicted in Figure 4.2 is essentially a ‘blackbox’ that performs the constraint solving process for the following two purposes:

- Assists the BDI engine to validate the applicability of a plan
- Generates a set of solution to the problem variables relevant to the application

\(^1\)Java Constraint Library is available at the URL: [http://www.epfl.ch/~torren Project JCL](http://www.epfl.ch/~torren Project JCL)
In the present instance of implementation, the constraint store will take the form of a constraint network (constraint graph). Thus the setting up of constraint store is a process to construct a constraint network based on all relevant constraints imposed when a trigger is used to invoke an applicable plan. This set of relevant constraints will include the following:

- Constraints imposed when a trigger unifies with the invocation section in the head of a plan.
- Constraints resulted from evaluating any constraint belief from the context section in the head of a plan.

This section illustrates the typical construction of the constraint store during the process of selecting an applicable plan for a trigger. A trigger, a constraint belief and two plans are used to demonstrate the derivation of a set of constraints that is subsequently submitted to the constraint solver for processing.

**Trigger**

```plaintext
+:!request(agent1,jun,1999,jane,Date,Slot).
```

**Constraint belief**

```plaintext
:appointperson(jane,Date,Slot) ← (1,slot1)&(1,slot6)&(2,slot3)&(5,slot6).
```

**Plan 1**

```plaintext
+:!request(Agentname,Month,Year,Person,Date,Slot):
    registered(Agentname)
← +/-appointreq(Person,Month,Year,Date,Slot).
```

**Plan 2**

```plaintext
+:!appointreq(Month,Year,Person,Date,Slot):
    currentmth(Month) & currentyear(Year) & domain(Person) & appointperson(Person,Date,Slot)
← display(Person,Date,Slot).
```
Table 4.4 below shows the constraint on the value of the variables in the constraint store when processing moves from the plan invocation phase to the context evaluation phase for generating plan 2 as an applicable plan.

<table>
<thead>
<tr>
<th>PHASE</th>
<th>CONSTRAINTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan invocation</td>
<td>Agentname = agent1 ; Month = jun ; Year = 1999 ; Person = jane</td>
</tr>
<tr>
<td>(plan 1)</td>
<td></td>
</tr>
<tr>
<td>Context evaluation</td>
<td>Agentname = agent1 ; Month = jun ; Year = 1999 ; Person = jane</td>
</tr>
<tr>
<td>(plan 1)</td>
<td></td>
</tr>
<tr>
<td>Plan invocation</td>
<td>Month = jun ; Year = 1999 ; Person = jane</td>
</tr>
<tr>
<td>(plan 2)</td>
<td></td>
</tr>
<tr>
<td>Context evaluation</td>
<td>Month = jun ; Year = 1999 ; Person = jane ; (1,slot1) ; (1,slot6) ; (2,slot3) ; (5,slot6)</td>
</tr>
<tr>
<td>(plan 2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4 Constraints formulation in constraint store

In the above example, the relevant problem variables are Person, Date and Slot. The trigger

```+!request(agent1,jun,1999,jane,Date,Slot)```

is first unified with the invocation of the plan 1. This unification generates the relevant plan which eventually becomes an applicable plan and produces a sub-goal

```+!appointreq(Month,Year,jane,Date,Slot)```

The sub-goal subsequently unifies with plan 2 and as context of plan 2 contains constraint beliefs, the constraint store is then updated with constraint imposed on the values of the respective problem variables (both from invocation and context section) as indicated in Table 4.4. In this case the set of problem variables together with their constraints are based on the variables found in the constraint beliefs that appear in plan 2. This final set of constraint is then submitted to the solver for processing.

<table>
<thead>
<tr>
<th>INPUT TO SOLVER</th>
<th>SOLUTION SET (FROM SOLVER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person = jane ;</td>
<td>(jane,1,slot1);(jane,1,slot6);</td>
</tr>
<tr>
<td>(1,slot1);(1,slot6);</td>
<td>(jane,2,slot3);(jane,5,slot6)</td>
</tr>
<tr>
<td>(2,slot3);(5,slot6)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5 Input and output from constraint solver.
Table 4.5 above shows the input (constraint set) to the solver and output (solution set) generated by the solver after constraint processing. Table 4.6 and 4.7 below show the actual parameter passing of constraints to the constraint engine at the implementation level and the formulation of constraints using the functions provided by JCL.

<table>
<thead>
<tr>
<th>CONSTRAINTS</th>
<th>VARIABLE, DOMAIN, CONSTRAINT TO SOLVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person = jane;</td>
<td>AddVariable(Person)</td>
</tr>
<tr>
<td>(1,slot1);(1,slot6);</td>
<td>AddDomain(Person)</td>
</tr>
<tr>
<td>(2,slot3);(5,slot6)</td>
<td>AddValue(Person, jane)</td>
</tr>
<tr>
<td></td>
<td>SetVariableDomain(Person, Person)</td>
</tr>
<tr>
<td>AddVariable(Date)</td>
<td></td>
</tr>
<tr>
<td>AddDomain(Date)</td>
<td></td>
</tr>
<tr>
<td>AddValue(Date, 1);</td>
<td></td>
</tr>
<tr>
<td>AddValue(Date, 2);</td>
<td></td>
</tr>
<tr>
<td>AddValue(Date, 5)</td>
<td></td>
</tr>
<tr>
<td>SetVariableDomain(Date, Date)</td>
<td></td>
</tr>
<tr>
<td>AddVariable(Slot)</td>
<td></td>
</tr>
<tr>
<td>AddDomain(Slot)</td>
<td></td>
</tr>
<tr>
<td>AddValue(Slot, slot1);AddValue(Slot, slot3);</td>
<td></td>
</tr>
<tr>
<td>AddValue(Slot, slot6)</td>
<td></td>
</tr>
<tr>
<td>SetVariableDomain(Slot, Slot)</td>
<td></td>
</tr>
<tr>
<td>SetConstraint(Date, Slot, 1, slot1)</td>
<td></td>
</tr>
<tr>
<td>SetConstraint(Date, Slot, 1, slot6)</td>
<td></td>
</tr>
<tr>
<td>SetConstraint(Date, Slot, 2, slot3)</td>
<td></td>
</tr>
<tr>
<td>SetConstraint(Date, Slot, 5, slot6)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6 Variable, domain and constraint to solver.

<table>
<thead>
<tr>
<th>SOLUTION SET</th>
<th>VARIABLE-VALUE FROM SOLVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(jane,1,slot1);(jane,1,slot6); (jane,2,slot3);(jane,5,slot6)</td>
<td>Person=jane, Date=1, Slot=slot1; Person=jane, Date=1, Slot=slot6; Person=jane, Date=2, Slot=slot3; Person=jane, Date=5, Slot=slot6</td>
</tr>
</tbody>
</table>

Table 4.7 Variable-value from solver.

As the solution set is not empty, plan 2 is treated as an acceptable applicable plan. Solution generated from the solving session will be returned by the solver to the applicable plan containing the context belief which initiates the constraint solving session. The solution(s) found will be used in the execution of primitive actions or creation of new events for new triggers. The execution of primitive action may change or revise the ‘mental beliefs’ (base beliefs and constraint beliefs) of an agent. Thus the
invocation of the next constraint solving session will be working with a fresh set of constraints.

4.2.3.2 BDI Agent Planning

BDI agent planning is an interpreting process and is responsible for ensuring continuous flow of proper program execution. Figure 4.1 depicts the logical structure of the BDI engine and in implementation. The entire structure can be classified into 3 main functional operations: reasoning, executing and supporting and is performed by the following seven major functions.

4.2.3.2.1 GenRelPlan

The function named GenRelPlan responsible to initiate the reasoning behavior after been provided with a trigger that signifies there is an event needs to be handled. The GenRelPlan function performs the first phase of the reasoning mechanism by attempting to unify the trigger with the invocation condition in the head of a plan. The outputs from GenRelPlan are plans that are unified to become a set of relevant plans which eventually become the input to the GenApplPlan.

4.2.3.2.2 GenApplPlan

GenApplPlan is the next function that will take over to process output generated by GenApplPlan. GenApplPlan accepts as an argument the set of relevant plans and begins to work on (unify) the context section in the head of every relevant plan. This phase of the reasoning process will possibly have some bearing on the constraint engine discussed earlier depends on the kind of beliefs occur in the plan context. The involvement of constraint engine occurs when a context belief takes on a dual role:
• as a base belief
• as an abstract modeling of specific constraint in terms of more expressive primitive constraint for the problem domain.

(The details of handling on a constraint belief during reasoning are given in the earlier section on constraint solving and section on operation semantics in chapter 3) The selector function for an applicable plan is an integrated part of the GenApplPlan whereby the selector is a trivial sub-function to pick up the first member from the resulting set of applicable plans. The final output from the GenApplPlan is a single applicable plan whereby the body of this plan will be applied with the applicable unifier to become an intended mean of the agent.

The GenApplPlan is also obligated to deal with the partially instantiated plan instance. This final responsibility of GenApplPlan will be an augmentation of mental attitudes by creating new intention or push the new plan instance on top of an existing plan stack (intention).

4.2.3.2.3 IntExecution

IntExecution is essentially a plan interpreter which takes a plan instance (a partially instantiated plan) as a parameter. Its main focus on execution will be in the body section of the plan instance. As the body of a plan is a series of applicable ‘maneuvers’, the plan interpreter will semantically parse each formula in the body to determine the right procedure to perform. As explained in the section on operation semantics (chapter 3), there are three options available and each involves an operation that can be:

1. Plain, low-level primitive action (e.g. send an output to printer, access to database, change the base beliefs of the agent or send a trigger to another agent to request for service).
2. Creating a new trigger for the agent and hence reduce the current plan to a lower level plan instance (in the case of a sub-goal).

3. Query the base beliefs of the agent to determine whether certain required condition (constraint) is met or specific belief is true or false.

The operation for option 2 above is provided by a small routine incorporated into the plan interpreter. This section of codes will generate an internal event and append it to the event queue waiting to be processed subsequently.

**4.2.3.2.4 ActionRoutine**

The *ActionRoutine* provides a suit of sub-routines (methods) that are used to perform low-level primitive actions in option 1 above. This component is a collection of all functional methods that will be responsible to deal with entity external to the BDI engine that includes entity external to the agent as well. The ability to add new primitive action by adding new methods into this functional component indicates that merely introducing new sub-routines into *ActionRoutine* can extend the working ability and functional capability of the agent.

**4.2.3.2.5 Unification, Substitution and UnifyBaseBel**

Within the BDI engine, *Unification* takes care of the unification processes that occur during *GenRelPlan* and *GenApplPlan* in the reasoning module.

Similar to *Unification*, the scope of *Substitution* service is primarily focused on providing support in term of instantiation of variables within the reasoning module. *Substitution* handles the tasks of applying the relevant unifier(s) to the context and the applicable unifier(s) to the body during the *GenApplPlan* operation.
UnifyBaseBel provides the service required by option 3 of the executing module. It helps to assess whether a query goal is an acceptable logical consequence or logical consequence of the current set of base beliefs. A positive outcome (true) enables IntExecution to apply the resulting unifier to remaining body of the executing plan. A negative outcome (false) causes IntExecution to abort execution of the current executing plan and the intention that contains the plan will be discarded.

Beside the three main functional components, the other two important supporting program routines depicted in Figure 4.1 and 4.2 are the event selector and intention selector. Each of these selectors consists of a manager and a thread that works in a cooperative manner to ensure smooth running of the agent program. The manager will manage directly the changes made to the event or plan set and the thread will be a running process that scans the event or plan set to pick up the first element from the set for processing.

4.3 Object Classes

The entire suit of program codes for ConstraintAgentSpeak interpreter is organised into two main packages: the BDI package and the JCL package. This section will provide an outline of these program packages and their respective roles in the implementation of the interpreter. In order to provide a clear and explicit explanation of the classes involved, this section has been organised into two short sub-sections based on the principal functional components of the ConstraintAgentSpeak interpreter (constraint processing and reasoning framework). Each of these functional sub-sections will present a brief outline of the classes involved. The relevant class diagrams to give a diagrammatic representation of the program classes (in term of their structural composition such as their
attributes and methods) and relationship among themselves are available in appendix A of this thesis.

4.3.1 Constraint Solving

As described in 4.2.3.1, JCL package is an adopted package and consists of a collection of constraint handling programs that provide the required services in constraint processing. The 3 basic aspects highlighted in section 4.2.3.1 and implemented in the constraint engine are provided by 6 leading classes from JCL: LiteralNetwork, Network, BTSolver, Solver, Solution, CustSolutionManager and SolutionManagerInterface. In this specific instance, the solver (BTSolver) is a backtracking solver that performs Simple Backtracking solving algorithm. Other solving technique can be applied simply by creating a new class which extends the Solver class and providing a Solve() method in the new class. Solver class is a base class for all solving algorithms and defines a framework for major facilities such as:

- the net variable which is referring to the low-level, efficient constraint network to be solved.
- the indexes[] array for containing instantiated variables.
- a suit of methods such as FindMoreSolutions(), telling the solver to find other solutions, NotifySolution(), notify when a solution is found, thread execution control methods such as StartSolving(), SuspendSolving(), ResumeSolving() or StopSolving() and other less important methods for statistical purposes.

SolutionManagerInterface is the Java interface that provides the necessary methods required for handling of solution generated during the solving process. In this particular
instance, these methods are implemented in the Solver class and are used to manipulate and output the solution into appropriate form for use in actions or sub-goal of an agent. Solution is the object class that provides an array to store the instantiated variables’ value as well as the methods used for trivial functions such as value type conversion, return value, display value etc. LiteralNetwork object class offers a high-level, direct representation of the constraint network (literal network) that can be manipulated easily. It provides simple methods (such as AddDomain, AddValue, AddVariable, SetConstraint, and RenameValue) that can be used for construction of a constraint network. Network is a low-level representation of the network built from methods in LiteralNetwork. It is this low-level version that the solving algorithm of the solver works with. It is more efficient in term of implementation (by eliminating the heavy usage of literal character strings) and hence will enable better performance when handled by the solver. A constraint network constructed via method in LiteralNetwork can be transformed into Network class representation by calling BuildNetwork method in the LiteralNetwork. The use of object classes from JCL provides a FD constraint domain in the form of binary CSP. Thus in this case, the result is an instance of ConstraintAgentSpeak with FD constraint handling capabilities.

4.3.2 Agent Planning

Programs for the reasoning framework (the BDI Framework) is packaged into a single program library called BDI package. The object classes in BDI package basically can be divided into two broad categories: the data structures group and the execution programs group. The classes in the data structures group provide the basic constructs required for building the beliefs and plans of an agent while classes in the execution program group
serve as the programs that drive the interpreting process and thus enable the execution of a *ConstraintAgentSpeak* program. Each agent is run as an autonomous thread that is initiated and controlled by the *AppletAgentManager* class. *AppletAgentManager* will invoke two other managers: *AppletEventManager* and *AppletIntentManager* that will take charge of the inference procedures and intention execution respectively. The runtime processing of the inference procedures and intention execution will be handled by two new threads (*AppletEventThread* and *AppletIntentThread*) spawned by the event manager and intention manager. Besides performing the job of creating the event and intention managers, *AppletAgentManager* also instantiate an *AppletBelPlanLib* object which will be responsible for the initialisation of agent's belief set (mental components) and plan library at the moment when an agent is created.

*AppletGenRelPlan* and *AppletGenApplPlan* are the object classes that provide the methods for the inferencing task to pick a relevant plan and eventually an applicable plan. The methods are supported by classes from the data structure group as illustrated by the class diagram A.2 in the appendix A. *AppletActionRoutine* and *AppletIntentExecution* are the classes that drive the intention execution of an agent. *AppletActionRoutine* provides the low level primitive action routines of an agent while the *AppletIntentExecution* acts as the coordinator that coordinates the execution sequence and procedures which take place at the body of the plan instances in the intention set.

**4.4 Conclusion**

The previous sections have looked at the implementation of *AgentSpeak(L) – C*’s interpreter from two perspectives:

1. A high level architectural point of view.
2. A detail analysis of the individual processes during runtime (Interpretation). The first perspective provides description on data structures involved as well as information (data) flows that occur during interpretation. On the other hand the step-by-step analysis of the respective runtime processes has offered technical insight on how the architectural components are cohesively linked together.

The explanation based on the above two perspectives is to provide as sufficient technical specification (algorithm) as possible to make the discussion on implementation issues clear and complete.
Chapter 5

Applications

5.1 Overview

This chapter provides some examples of applications developed using ConstraintAgentSpeak. Two applications are presented. A full implementation exists for the first application – a multi-agent meeting scheduler system and an implementation of the second application – a multi-agent stock broking system is currently underway. This chapter discusses both applications and provides illustrative samples of ConstraintAgentSpeak codes used to develop these systems. In the context of this thesis, an agent is viewed as an autonomous or semi-autonomous software system that performs tasks in non-trivial, dynamic environments.

5.2 Distributed Scheduling

In abstract terms, distributed scheduling problem can be viewed as resolving a collection of local scheduling problems to derive at an optimum or near optimum global solution. This section discusses and evaluates the common features of scheduling problem found in most scheduling environments (e.g. job-shop scheduling in a factory, crew timetabling in an airline, delivery scheduling in a logistic company or appointment scheduling in an office). All of the scheduling tasks ranging from the complex to the simple share some common features as listed below:

1. The existence of a control structure that monitors and coordinates the entire scheduling environment.
2. The scheduling problem can only be solved via a decomposition into sub-problems whereby each of the sub-problems is addressed by a local solver. The local solver used is typically the same for each sub-problem (distributed problem solving).

3. Each of the sub-problems can be modeled using a subset of variables (that occur within the whole scheduling environment) and the constraints that exist between the variables within the subset.

By analysing the above three common features that prevail in most scheduling environments, it is obvious that the architecture of ConstraintAgentSpeak offers a fundamental solution for the problem framework of scheduling. In this case, the BDI framework forms the much needed control structure (item 1 above) supported by the BDI engine together with base beliefs that serve as the high-level empirical knowledge or the general environmental knowledge. Item 2 and 3 above will be modeled using the relevant variables and constraints that define the limitations or relationships between the variables.

The specific and rigid procedures will be standard constraint solving routines applied to the constraints to determine the solution set for the variables of interest.

Based on the above fundamental attributes, a complete scheduling system can be viewed as a distributed scheduling system whereby a group of software agents collectively form a cohesive software entity. In such instance, each agent will be responsible for the scheduling task that needs to be resolved locally. The locally derived solution is then delivered to the relevant agent if the shared constraint that binds the two agents is consistent.
Under this perspective, the constraint-based agents are equipped with constraint-solving capabilities to address features 2 and 3 mentioned above. In [Musliner, 1996], constraint-based agents have been deployed to tackle a distributed scheduling problem. Despite the differences in the nature of the scheduling task involved, the DACC agent (a variant of constraint-based agent) in [Musliner, 1996] is essentially a collection of constraint solvers using Contract Net Protocol (CNP) to achieve an intelligent negotiation mechanism. The constraint-based scheduler (DACC agent) reacts to local scheduling task using the least-commitment approach. In this approach, the set of tasks to be performed is constrained only if they are relevant and dependent on one another. As a result, the extent of constraints becomes more detail when the number of task increases and the interdependency between tasks become more complex as more tasks are assigned to a particular DACC agent. At the same time, the respective DACC agent uses the contract negotiation techniques to handle bidding and award of contract (task) to other agents. The marginal cost derived from the evaluation of value function in constraint-based schedule provides the fundamental decision-making factor for the CNP based negotiation.

Agents in the framework of Musliner et. al. are simply constraint solvers encapsulated as autonomous entities. In constrast, ConstraintAgentSpeak agents are able to plan in reactive setting in addition to having constraint solving capabilities.

It is worth to discuss the work of [Sen et.al., 1998] on distributed meeting scheduler (DMS). In DMS it is recognised that meeting scheduling requires a careful balance between the individual needs and organization requirements. The work on DMS views meeting scheduling as a distributed search process that attempts to strike the best balance between the above two elements. In this instance, heuristics are used to guide distributed
scheduling decisions to improve the performance and efficiency of the individual scheduler (agent) that represents the respective user. However the work on DMS is not directly relevant to constraint-based agents in term of its research emphasis. In DMS the focus is on the heuristic strategies (as opposed to constraint-based strategies) to help scheduling agents make better decisions. It looks into issues concerning the coordination among intelligent schedulers (agents) that interact to address the distributed resource allocation problem (information about preferences or time available). The main strategy adapted in DMS for negotiation and distributed problem-solving is based on the multistage negotiation protocol. There are four flexible algorithmic steps that form a ‘goal-driven’ mechanism in the adapted multistage negotiation protocol. They are briefly described as follows:

1. A host (initiating or requesting agent) attempts to select an earliest possible free meeting slot(s) that is are consistent with its own schedule constraints. One or more than one earliest possible intervals will be selected if more than one meeting participant is expected.

2. Invitees to a meeting receive a broadcast of the meeting proposal and respond by trying to find locally a set of free meeting slots that match the proposed slots and reply (bids) to the initiating host. A counter proposal will be made if required.

3. The host collects all the responses and makes attempts to select a common time slot that satisfies its own schedule constraints. The selected common time slot (award) will be broadcast to all invitees. If no such time slot is available, based on the responses received, a new proposal is computed and broadcast again to the invitees.

4. The invitees on receiving the new proposal will respond as in step 2 above. If an award is received, the invitee checks its own schedule and if the time slot is still available, it will be marked off to reserve the time slot for the meeting.
Agents created using ConstraintAgentSpeak for the appointment scheduling system that will be discussed in section 5.2.1, are also possible to emulate the above four algorithmic steps to achieve the ultimate intention of converging to a global consistent schedule. The converged solution for the global constraint will be the existence of a non-empty set of intersection on the meetings (replies) that have been received from the responding agents. A plan that caters for solving this global constraint and locating the above intersection will deploy an action to handle the task. The action makes use of a counter and a common request identification to ensure responses are received from all agents that the announced appointment has been broadcast to. The process of solving the global constraint for a final solution will be processed only when all participants have responded.

5.2.1 Interactive Appointment Scheduling Agents (ISchAgents)

ISchAgents is an appointment scheduling program coded using ConstraintAgentSpeak. It is visualised as a distributed scheduling problem involving a community of scheduling agents. The distributed scheduling problem can be viewed as a two-level loosely coupled constraint satisfaction problem in the following sense:

1. Local constraint satisfaction problem on appointment slot allocation handled by individual (local) agent.
   (such as availability of appointment slot on particular date for a person)
2. Mutually acceptable global solution generated by a group of local agents through a series of communication and coordination process.
   (such as availability of a particular date of a month for appointment, e.g. the particular date should not fall on Sunday)

Each agent within the community is able to organise appointments for exactly one person and is equipped with the following functionality:
• Send and receive request for an appointment
• Respond to a request for an appointment
• Send and receive respond to a proposal of solution
• Send and receive final confirmation to a request

The four basic categories of scheduling problem that can be handled by IschAgents are:

• Category one
  
  Input : Person
  
  Output : Set of appointment slots available for the Person within a fixed date range

• Category two
  
  Input : Person and Date
  
  Output : Set of appointment slots available for the Person on the given Date

• Category three
  
  Input : Person and Slot
  
  Output : Set of appointment slots available for the Person on all available Date

• Category four
  
  Input : A set of Person
  
  Output : Set of appointment slots available for the set of Person within a fixed date range

ISchAgents displays multi-agents attributes by being able to behave in an interactive manner to achieve the intended objective of the respective agent’s owner. Within this context, an ISchAgent is a constraint-based agent that acts autonomously on behalf of its owner(s) and to schedule appointment for its owner(s) locally. In doing so, it acts base on the beliefs (e.g. `appointperson(jane, Date, Slot)←(1,slot1), (1,slot6)`) that it has.
for its owner as well as the operating environment. Compared to distributed meeting scheduler in [Sen et.al, 1998], information exchange has received more emphasis in IschAgents and communication between the agents is carried out through the use of triggering event initiated by plan. The IschAgent itself serves as an event listener that makes use of the flow of communication adapted and improvised from the event-listener concept in Java development environment. It is deemed efficient and appropriate to the working protocol of ConstraintAgentSpeak agent as external triggering event is the only mean of interaction between agents. Program 5.2 gives a comprehensive illustration of the ISchAgents program coded in ConstraintAgentSpeak, it provides more detail specification of the functionality of ISchAgent through the use of plans for its respective purpose. A requesting agent, on receiving the replies from the responding agents, will attempt to compute an intersection of the appointment slots proposed by the various responders. A non-commitment style of strategy is applied to the booking of an appointment slot until the schedule for the particular slot has been confirmed.
Program 5.2 – 1 Sample program of ISchAgents

```plaintext
/* ***** base beliefs *****/*
currentmth(jun).
currentyear(1999).
valid(jane).
valid(jess).
/* person represent by other agents */
represent(agent1,bream).
represent(agent2,marlin).
/* other agents registered with this agent */
registered(agent1).
registered(agent2).

/* ***** constraint beliefs ******/
domain(Day) ← (mon)&(tue)&(wed)&(thu)&(fri).
domain(Date) ← (1)&(2)&(3)&(4)&(5)&(6)&(7)&(8)&(9)&(10)&(11)&(12)&(13)&(14)&(15)&(16)&(17)&(18)&(19)&(20).
domain(Person) ← (jane)&(jess).
domain(Slot) ← (slot1)&(slot2)&(slot3)&(slot4)&(slot5)&(slot6)&(slot7)&(slot8).
constraint1(Day,Slot) ← (mon, slot1)&(mon, slot2)&(mon, slot3)&(mon, slot4)&(tue, slot5)&(tue, slot6)&(tue, slot7)&(tue, slot8).
constraint2(Day,Date) ← (mon, 1)&(tue, 2)&(wed, 3)&(thu, 4)&(fri, 5)&(sat, 6).
Constraint3(Person,Day) ← (jess, fri).
appointperson(jane,Date,Slot) ← (1, slot1), (1, slot6), (2, slot3), (5, slot6).
appointperson(jess,Date,Slot) ← (2, slot1), (2, slot2), (2, slot3), (5, slot6).
```

```
// Generate a number of outgoing requests based on the number of person required to schedule for appointment, broadcast
// create a new belief that store the number of request for the request ID

++schedulemeets(SrCper,Per1,Per2,Per3,Per4,Per5, __
    Month,Year,Date,Slot) : true __
← genreqid(Outreqid) & __
    broadcast(Outreqid,Per1,Per2,Per3,Per4,Per5) & __
++outrequest(SrCper,Outreqid,Per1,Month,Year,Date,Slot) & __
++outrequest(SrCper,Outreqid,Per2,Month,Year,Date,Slot) & __
++outrequest(SrCper,Outreqid,Per3,Month,Year,Date,Slot) & __
++outrequest(SrCper,Outreqid,Per4,Month,Year,Date,Slot) & __
++outrequest(SrCper,Outreqid,Per5,Month,Year,Date,Slot) & __
++waitresponse(Outreqid,Per1,List1) & __
++waitresponse(Outreqid,Per2,List2) & __
++waitresponse(Outreqid,Per3,List3) & __
++waitresponse(Outreqid,Per4,List4) & __
++waitresponse(Outreqid,Per5,List5) & __
++solve(Outreqid,List1,List2,List3,List4,List5, __
    Per1,Per2,Per3,Per4,Per5).

++waitresponse(Reqid,Toagent,List) : true __
← +!proposefrom(Reqid,Toagent,List).

++waitresponse(Reqid,Toagent,List) : true __
← wait(10) & __
++waitresponse(Reqid,Toagent,List).

++solve(Reqid,L1,L2,L3,L4,L5) : true __
← combine(L1,L2,L3,L4,L5,List) & __
    newbelief("othconstraint(*,Reqid,"Date","Slot", __
        ") ← ",List) & __
++gensolution(Reqid).

// Outgoing request to another agent. After verifying valid month and year. Action newrequest despatch a new external trigger to request for appt. A new belief is added using action newbelief to indicate the outgoing request.
++outrequest(SrCper,Outreqid,Person,Month,Year,Date,Slot) : true __
currentmth(Month) & __
currentyear(Year) & __
represent(Person,Toagent) __
← newrequest(Outreqid,Toagent,Person,Month,Year,Date,Slot) & __
newbelief("request(*,Outreqid,Toagent,*)") & __
newbelief("reqinfo(*,Outreqid,Toagent,Month,Year, *,
    Person,Date,Slot,"*")") & __
newbelief("reqperson(*,Outreqid,SrCper,"*)").

Program 5.2 – 2 Sample program of ISchAgents
//Incoming request from another agent. After verification,
//an internal event is generated by sub-goal (+!appointreq) to
//compute for solution. A new belief (apptrequest) is added
//using action newbelief to indicate the incoming request.
+!inrequest(Inreqid,Fromagent,Person,Month,Year,Date,Slot) : @_
   valid(Person) & @_
   registered(Fromagent) @_
   newbelief("apptrequest(",Inreqid,Fromagent,")") & @_.
   +!appointreq(Inreqid,Fromagent,Month,Year, @_
   Person,Date,Slot).

//Handling of proposed solution returned from responding agent
+!receiveresponse(Reqid,Toagent,List) : @_
   request(Reqid,Toagent) @_
   newbelief("reply("Reqid,Toagent,List,")").

//processlist store the List received for compilation of final
//solution set to be input for constraint processing and
//generation of final intersect solution
+!proposefrom(Reqid,Toagent,List) : @_
   reply(Reqid,Toagent,List) @_
   processlist(Reqid,Toagent,List).

//plan to generate the intersect global solution
//A selected choice (first) of appointment slot is picked
//by action selectapt and reply as +!respondapt
+!gensolution(Reqid,List) : @_
   rcperson((Reqid,Sourceperson) & @_
   reqinfo(Reqid,Toagent,Mth,Yr,Per,Sourcedate,Sourceslot) & @_
   appointperson(Sourceperson,Sourcedate,Sourceslot) & @_
   othconstraint(Reqid) @_
   selectapt(Reqid,List,Per,Finaldate,Finalslot).

//Respond choice receive from requesting agent
//Sub-goal (+!confirmapt) generate an internal event to
//despatch a final confirmation to requesting agent
+!respondapt(Fromagent,Person,Date,Slot) : @_
   apptrequest(Reqid,Fromagent) & @_
   appointperson(Person,Date,Slot) @_
   +!confirmapt(Reqid,Fromagent,Person,Date,Slot).

//appointreq sub-goal:generate solution set (Person,Date,Slot)
//Action genproposeappt will generate an external event
//+!receiveresponse to deliver the proposed solution set to the
//requesting agent
+!appointreq(Inreqid,Fromagent,Month,Year, @_
   Person,Date,Slot) : @_
   currentmth(Month) & @_
   currentyear(Year) & @_
   appointperson(Person,Date,Slot) @_
   display(Inreqid,Fromagent,Listso1n) & @_
   genproposeappt(Inreqid,Fromagent,Listso1n).

Program 5.2 – 3  Sample program of ISchAgents
// Despatch final confirmation (+! finalconfirm) to requesting agent is performed with confirmation after constraint belief is updated.

+!confirmappt(Reqid, Fromagent, Person, Date, Slot) : true @_
  ← updateconstbel(Person, Date, Slot, confirm) & @_
  confirmation(Reqid, Fromagent, Person, Date, Slot, confirm).

// Final confirmation receive from responding agent. Constraint belief is updated to reflect the latest constraint status in requesting (source) agent.

// Updstatus : confirm (remove from available slots) or cancel (add back to available slots)
+!finalconfirm(Reqid, Toagent, Person, Date, Slot, Updstatus) : @_
  request(Reqid, Toagent) @_
  ← updateconstbel(Person, Date, Slot, Updstatus).

// To cancel appt for (Person, Date, Slot) by the requesting agent. Send cancel send out trigger +!cancelappt
+!cancel(Reqid, Toagent, Person, Date, Slot) : @_
  request(Reqid, Toagent) & @_
  ← sendcancel(Reqid, Toagent, Person, Date, Slot, cancel).

// After cancel appt for (Person, Date, Slot). Confirmation is sent to the requesting (source) agent.
+!cancelappt(Reqid, Fromagent, Person, Date, Slot, Updstatus) : @_
  apptrequest(Reqid, Fromagent) & @_
  ← updateconstbel(Person, Date, Slot, Updstatus) & @_
  confirmation(Reqid, Fromagent, Person, Date, Slot, cancel).

Program 5.2 - 4 Sample program of ISchAgents
Figure 5.1 below depicts the system scenario of ISchAgents during run-time with four agents created using `ConstraintAgentSpeak`. Each agent represents a different person and thus is equipped with different set of beliefs pertaining to its owner. The agent has its own interface to communicate with its owner or user and a database to store all the data pertaining to the beliefs of the agent.

![Diagram of ISchAgents system scenario](image)

Figure 5.1 System scenario of ISchAgents during run-time

### 5.3 Multi-Agent Broking System (MABS)

This section describes the design and implementation of MABS - A multi agent broking system. In MABS, the constraint domain is the domain of reals where constraint solving basically involves computing solutions for linear or non-linear equations (through the use of methods that involve substitution and linear programming). The following three properties suggest that `constraint-based agents` are an appropriate approach to developing a stockbroking application:

- A key requirement for this application domain is the ability to plan reactively in a highly dynamic environment.
- Real number expression and real number computation form the major bulk of the computing needs in MABS. Hence constraint solving naturally becomes a competent solution to this application.
There is a need to compute the tightest possible set of constraints on the problem variables, even if these constraints do not assign exact values to these variables. Such partial solutions are useful. For instance, an expression $\text{Balance} = \text{Bank\_Balance} - (\text{Quantity} \times \text{Price})$ can be partially evaluated to $\text{Balance} = 70000 - (6000 \times \text{Price})$ if the exact values of $\text{Bank\_Balance}$ and $\text{Quantity}$ can be computed via constraint solving. Several high-level constraint programming language, such as CLP(R) [Jaffar, 1992] supports this facility.

In the subsequent sections, the design of the MABS system is presented as another example of application program that ConstraintAgentSpeak is applicable. An actual implementation is planned for the near future.

5.3.1 Objective

By looking at the nature of a typical stock trading environment and the intensive level of communication involved, the objective of MABS is to simulate such a controlled trading environment and enable the trading activities of a stockbroker (remisier or dealer) to be automated. A stockbroker is able to enter an order placed by a client and MABS will take over the monitoring of the entire transaction.

The inputs that a stockbroker will enter into the MABS are given as:

- client code
- stock involved
- transaction type: buy or sell
- quantity
- price

Based on the above trade order information and other information pertaining to the client which were stored in the client master file in the database. A transaction is initiated and
the trade is executed when the right price is prevailed. On completion of the transaction, the client is notified the outcome of the trade and other account information are updated. The outputs that are returned to the user will be:

- client code
- stock involved
- transaction type: buy or sell
- quantity bought or sold
- price done

5.3.2 Agent Model

MABS comprises of a number of functionally distinct agents working together. The member agents within MABS are: transaction agent (TA), administrator agent (AA), proxy agent (PA), price info agent (PIA), and trade execution agent (TEA). Figure 5.2 depicts the run-time environment of MABS and the interactions between the member agents. When a trade is first initiated, the administrator agent on receiving instruction from the user, communicates with the proxy agent to verify and validate the client requesting the transaction. Once authorized, administrator agent creates a transaction agent to handle the processing of the trade. The transaction agent will subsequently submit the trade to trade execution agent for execution. On receiving the submitted trade, trade execution agent updates its base belief to capture the trade and relay the trade information to price info agent for price monitoring. Based on the real-time stock price maintained and trade information received, price info agent informs trade execution agent when the requested stock price prevailed. Trade execution agent will only submit the trade for execution when the right requested price is notified.
Programs (5.3–1 to 5.3-5) illustrate the sample programs for the respective agents (TA, AA, PA, PIA and TEA) coded using ConstraintAgentSpeak. The execution of MABS is handled by a number of daemon threads and process threads. Daemon threads are background threads for static agent that constantly listens for triggering events, monitors on activities performed on the contents of relevant registry within an agent etc. They are initiated at the ‘startup’ of the system and remain active throughout the run-time until the entire system is shut down. Process threads are a multiplicity of threads that are spawned each time a dynamic agent is instantiated (e.g. when a transaction agent is created to handle the processing for a particular transaction within a specified time constraint). Each process thread remains active for the specified period of time and is removed at timeout.

Communication and collaboration between the agents are performed via message passing from one agent object to another to invoke the relevant methods or to instantiate the required run-time agent objects to perform specific function within the systems. Each of the agent object will be able to respond immediately to all triggering events by relying on its current beliefs set and plan set, irrespect of whether the triggering event is from neighbouring agent or external source. This enables the agents to display limited reactive behaviour in a real-time environment. Figure 5.2 depicts the overall layout of the system architecture at run-time.

![Figure 5.2 System scenario of MABS during run-time](image)
A high-level description on the functionalities of the respective member agents is given below. The detailed functional explanations on the plans in the respective agents are provided in the sample program 5.3. A limitation which should be highlighted here is that as the implementation of MABS is still at its early design stage, a number of the features or behaviours outlined in the following sub-sections are not provided with their performance plans in the sample program 5.3.

1. **Transaction Agent (TA)**

- **Functionality**: Transaction agent is an agent responsible to handle the processing of a requested transaction. It is able to authorize and submit trades for execution by the trade execution agent. It also has the ability to spawn new transaction agent (e.g. to perform additional buy or sell, to arbitrage) based on its existing beliefs and plans.

- It is an agent object that is instantiated with an initial belief set whenever a new distinct, transaction is entered into the system and will vanish at the end of a transaction cycle. A Transaction agent is a dynamic agent and there can be a set of active transaction agents at run-time.

- A transaction cycle begins when a new transaction is requested and ends when the requested transaction has been executed.

- During the life span of the agent, its belief set will change with addition or deletion of beliefs based on various states of its life cycle.

- Every transaction agent will be assigned a unique identification tag and its computation state is maintained via the assigned tag. Interactions (which may result in modifications of an agent’s beliefs) with other agents are carried out based on the unique identification allocated to the respective agent.

- The state of each of the transaction agents will be constantly monitored by the administrator agent which will act according to its current beliefs regarding each of the transaction agents. Through the collaboration effort with other agents, it will enable the transaction agent to behave reactively and to simulate the trading strategy of a trader in limited fashion.
2. **Administrator Agent (AA)**

- **Functionality**: *Administrator Agent* adopts an interface role that receives instructions from trader. Request for authorization from *proxy agent*, instantiate a new *transaction agent* when a request is approved or authorized.

- *Administrator agent* is a static agent that maintains a registry of all the active *transaction agents* whether they are instantiated by itself or are spawned by another active *transaction agent*.

- It has its own belief set that reflects the state information for all *transaction agents* that are still active. It will 'talk' to them in order to change its belief about the current state of each *transaction agent*. At regular time interval, belief information at designated state will be downloaded to a database for permanent storage.

3. **Proxy Agent (PA)**

- **Functionality**: *Proxy agent* is a static agent that assists the *administrator agent* to verify and to validate all trade requests from the traders.

- These verification and validating process is carried out based on the beliefs it has for each of the trader. The universal set of the belief terms for the entire client (trader) base is stored in a database that is accessible to the *proxy agent*.

- The information verified and validated by the *proxy agent* are trader's validity and trader’s trading status. The *proxy agent* will respond to each of the query by returning an ‘approved’ reply together with relevant transaction information (e.g. fund available) for the *administrator agent* to act on.

4. **Price Info Agent (PIA)**

- **Functionality**: *Price Info Agent* is a static agent that maintains a real-time registry with the pricing information for all the stock counters.

- Its belief set consists of tuples of *<counter, requestor, price requested>* information (e.g. price limit for arbitrage, requested price) about trades waiting for arbitrage opportunities and trades in the on-line registry of *trade execution agent* waiting to be executed.

- When the right price for a particular share is prevailed, it will announce to the *trade execution agent* the prices for the trades that are still outstanding and due to be executed.
5. *Trade Execution Agent (TEA)*

- **Functionality**: *Trade Execution Agent* plays the role of a scheduler that submits trade for execution. It maintains an on-line registry which keeps track of all the outstanding trades waiting to be executed.

- A set of \(<\text{counter, requestor, price requested}\>\) information is relayed to the *price info agent* to facilitate the price monitoring process.

- *Trade execution agent* is a static agent that will relieve the trader from tedious effort of constantly monitoring the price changes. It reacts to price changes in accordance to the instructions given by the *transaction agent*. Online price information provided by *price info agent* enables it to immediately submit trade for execution or hold back and wait for the right timing before it acts.

- Its behaviour is determined by the belief set it holds for each of the outstanding trades on its registry. There are a set of plans (e.g. submit trade immediately, hold back until specified date, hold back until price is above or below specified limit etc.) that will be fired accordingly based on the different beliefs for different trades. Executed trade will be updated to the relevant *transaction agent* and hence change its belief set with the addition of a new belief.
Program 5.3-1 Sample program of MABS

```plaintext
//base beliefs - TA
client(john, active).
returnreq(true).
//belief indicates to arbitrage trade

//Accounting figures for john: acctvalue(X, Sale, Bank, Pur, Margin)
aacctvalue(john, 20000, 80000, 30000, 80000).

//constraint beliefs
retvalue(john, return) ← Return = 10.
expiry(john, ED) ← ED = 18jun1999. //transaction expiry date
fund(john, Sale, Bank, Pur, Qty, Rprice) ← Sale+Bank-Pur ≥ (Qty*Rprice).
fund(john, Sale, Bank, Pur, Margin, Qty, Rprice) ←
    Sale+Bank+Margin-Pur ≥ (Qty*Rprice).
buyarbitr(john, Aprice, Dprice, Ret) ←
    (((Aprice - Dprice)/Dprice) * 100) ≥ Ret.
sellarbitr(john, Aprice, Dprice, Ret) ←
    (((Dprice - Aprice)/Dprice) * 100) ≥ Ret.
buy(Stk, Qty, Rprice, agent2, Dqty, Dprice) ← Dqty=Qty & Rprice2Dprice.
sell(Stk, Qty, Rprice, agent4, Dqty, Dprice) ← Dqty=Qty & Rprice≤Dprice.

//plan library

//despatch() delivers a new trade to trade execution agent for
//submission and include an insertion of a new constraint belief :
//buy( or sell() )
+!buy(X, Stk, Qty, Rprice, Agentid) : client(X, active) & @
    acctvalue(X, Sale, Bank, Pur, Margin) & @
    fund(X, Sale, Bank, Pur, Qty, Rprice) & @
    newbelief("buy(" , Stk, Qty, Rprice, Agentid, Qty, Rprice,")") & @
    despatch(X, buy, Stk, Qty, Rprice, Agentid).

+!sell(X, Stk, Qty, Rprice, Agentid) : client(X, active) & @
    newbelief("sell(" , Stk, Qty, Rprice, Agentid, Qty, Rprice,")") & @
    despatch(X, sell, Stk, Qty, Rprice, Agentid).

//action newtransagent() spawns a new agent to handle arbitrage trade
//and initiates all the required based beliefs. newtrade() generates
//an external trigger event for the new agent
+!newtrnx(sell, X, Stk, Dqty, Aprice) : client(X, active) & @
    acctvalue(X, Sale, Bank, Pur, Margin) & @
    newtransagent(X,active,Sale,Bank,Pur,Margin) & @
    newtrade(X,sell,Stk,Qty,Aprice).

+!newtrnx(buy, X, Stk, Dqty, Aprice) : client(X, active) & @
    acctvalue(X, Sale, Bank, Pur, Margin) & @
    newtransagent(X,active,Sale,Bank,Pur,Margin) & @
    newtrade(X,buy,Stk,Qty,Aprice).
```

Program 5.3 - 1 Sample program of MABS
//plan to allow for alternative action to purchase, with extra fund
//from margin
+!buy(X,Stk,Qty,Rprice,Agentid) : client(X, active) & @
   acctvalue(X,Sale,Bank,Pur,Margin) & @
   fund(X,Sale,Bank,Pur,Margin,Qty,Rprice) @_
   newbelief("buy(,Stk,Qty,Rprice,Agentid,Qty,Rprice,")") & @_
   despatch(X,buy,Stk,Qty,Rprice,Agentid).

//action notify() informs client on done transaction, removal of
//belief returnreq(true) and include insertion of new belief :
// buytradereturn() or selltradereturn(X)
//action updaccount() updates the current accounting details belief
//acctvalue(X,Sale,Bank,Pur,Margin) and generates an external events
//to administrator agent to update the accounting details for client
//action finalise() initiates an external event for administrator
//agent to terminate the agent thread
+!done(X,Stk,Dqty,Dprice):returnreq(true) & @_
   acctvalue(X,Sale,Bank,Pur,Margin) & @_
   retvalue(X,Return) & @_
   buy(Stk,Qty,Rprice,Agentid,Dqty,Dprice) & @_
   notify(X,buy,Stk,Dqty,Dprice) & @_
   updaccount(X,Sale,Bank,Pur,Margin,Dqty,Dprice,Agentid) & @_
   +!arbitrage(X,Stk,Dqty,Dprice,Return) & @_
   finalise(X,buy,Stk,Dqty,Dprice,Agentid).

+!done(X,Stk,Dqty,Dprice):returnreq(true) & @_
   acctvalue(X,Sale,Bank,Pur,Margin) & @_
   retvalue(X,Return) & @_
   sell(Stk,Qty,Rprice,Agentid,Dqty,Dprice) & @_
   notify(X,sell,Stk,Dqty,Dprice) & @_
   updaccount(X,Sale,Bank,Pur,Margin,Dqty,Dprice,Agentid) & @_
   +!arbitrage(X,Stk,Dqty,Dprice,Return) & @_
   finalise(X,sell,Stk,Dqty,Dprice,Agentid).

//Case or plan cater for no return specified
//notify() inform client about the completed transaction
+!done(X,Stk,Dqty,Dprice):acctvalue(X,Sale,Bank,Pur,Margin) & @_
   buy(Stk,Qty,Rprice,Agentid,Dqty,Dprice) & @_
   notify(X,buy,Stk,Dqty,Dprice) & @_
   updaccount(X,Sale,Bank,Pur,Margin,Dqty,Dprice,Agentid) & @_
   finalise(X,buy,Stk,Dqty,Dprice,Agentid).

+!done(X,Stk,Dqty,Dprice):acctvalue(X,Sale,Bank,Pur,Margin) & @_
   sell(Stk,Qty,Rprice,Agentid,Dqty,Dprice) & @_
   notify(X,sell,Stk,Dqty,Dprice) & @_
   updaccount(X,Sale,Bank,Pur,Margin,Dqty,Dprice,Agentid) & @_
   finalise(X,sell,Stk,Dqty,Dprice,Agentid).

//Spawn new transactions with arbitrage price computed
+!arbitrage(X,Stk,Dqty,Dprice,Ret):buyarbitr(X,Aprice,Dprice,Ret) @_
   +!newtr(x(sell,X,Stk,Dqty,Aprice).

+!arbitrage(X,Stk,Dqty,Dprice,Ret):sellarbitr(X,Aprice,Dprice,Ret) @_
   +!newtr(x(buy,X,Stk,Dqty,Aprice).

Program 5.3 - 2 Sample program of MABS
dealer(d01).
dealer(d02).
registered(agent1, james).
registered(agent2, john).
registered(agent3, james).
registered(agent6, john).
status(agent1, active).
status(agent1, active).
status(agent1, active).
status(agent1, active).
request(john, d01, buy, telstra, 1000, 8.00, req001).

******plan library******

//action validate() creates a new event to be processed by the proxy
//agent and insert a request belief with an identification key.
+!traderesuest(X, Remis, Type, Stk, Qty, Price, ReqId) : dealer(Remis) & @_
  <- newbelief("request(\"X, Remis, Type, Stk, Qty, Price, ReqId,\")") & @_
    validate(X, Remis, Reqld).

//action agent() instantiates a new agent with a unique agent ID and
//the relevant beliefs to handle the request transaction.
//a belief is maintained for every active agent with its agent ID
//action newtrade() generates an external event for the new agent
//action updrequest() updates request to database
+!proxyreply(X, active, Remis, Sale, Bank, Pur, Margin, Requestid) : @_
  request(X, Remis, Type, Stk, Qty, Price, Requestid) @_
  <- newtransagent(X, active, Sale, Bank, Pur, Margin,) & @_
    newtrade(X, Type, Stk, Qty, Price) & @_
    updrequest(X, Type, Stk, Qty, Price).

//action updtrade() updates transaction to database
+!finalise(X, Type, Stk, Dqty, Dprice, Agentid) : registered(Agentid, X) @_
  <- updtrade(X, Type, Stk, Dqty, Dprice).

//action updclient() update to database clients latest accounting
//information
+!updaccountdet(X, Sale, Bank, Pur, Margin, Agentid) : @_
  registered(AgentID, X) @_
  <- updclient(X, Sale, Bank, Pur, Margin).

Program 5.3 - 3 Sample program of MABS
//****base beliefs - PA*****/
client(jahn, valid).
client(james, valid).
client(jasper, valid).
client(john, valid).
acctinfo(john, active, d01, 20000, 80000, 30000, 80000)

//****plan library*****/

//action retrieve() retrieves a valid client details and inserted as the new base belief of the proxy agent on the relevant client
+!validatequery(X, Remis, Requestid) : client(X, valid) & @_
  ¬retrieve(X, Status, Remis, Sale, Bank, Pur, Margin) & @_
  ¬newbelief("acctinfo("X, Status, Remis, Sale, Bank, Pur, Margin, ")") & @_
    +!check(X, Remis, Requestid).

//action reply() returns a genuine, active client to the administrator agent with the request ID for further processing
+!check(X, Remis, Requestid) : @_
  acctinfo(X, active, Remis, Sale, Bank, Pur, Margin) & @_
  reply(X, active, Remis, Sale, Bank, Pur, Margin, RequestID).

//****base beliefs - TEA*****/

trading(on).
trade(john, buy, telstra, 6000, 10.00, agent6).
tradinglimit(Lqty, Lprice) ≤ (Lqty * Lprice) ≤ 100000.

//****plan library*****/

//action newtrade(X, buy, Mstk, Mqty, Mprice, Magentid) inserts a new base belief for the trade waiting to be executed.
//action monitor(Mstk, Mprice, Magentid) relays the required trading information for price tracking by price info agent.
+!despatch(X, buy, Mstk, Mqty, Mprice, Magentid) : trading(on) & @_
  tradinglimit(Mqty, Mprice) & @_
  ¬newbelief("trade("X, buy, Mstk, Mqty, Mprice, Magentid, ")") & @_
    monitor(buy, Mstk, Magentid, Mprice).
+!despatch(X, sell, Mstk, Mqty, Mprice, Magentid) : trading(on) & @_
  ¬newbelief("trade("X, sell, Mstk, Mqty, Mprice, Magentid, ")") & @_
    monitor(sell, Mstk, Magentid, Mprice).

//action submit() deliver trade for execution when was informed of the right stock price by the price info agent
+!priceinfo(Pstk, Pprice, Pagentid) : @_
  trade(X, Type, Pstk, Qty, Price, Pagentid) & @_
    Type = buy & @_
    Pprice ≤ Price & @_
    submit(X, Type, Pstk, Qty, Pprice).
+!priceinfo(Pstk, Pprice, Pagentid) : @_
  trade(X, Type, Pstk, Qty, Price, Pagentid) & @_
    Type = sell & @_
    Pprice ≥ Price & @_
    submit(X, Type, Pstk, Qty, Pprice).

Program 5.3 - 4 & 5 Sample programs of MABS
****base beliefs - PIA*****/
counter(telstra,8.00).
counter(bhp,10.00).
counter(westpac,16.00).
counter(nab,18.00).
pendtrade(sell,bhp,11.00,agent3).
pendtrade(sell,telstra,9.00,agent5).
pendtrade(buy,westpac,15.00,agent2).
pendtrade(buy,nab,17.00,agent4).

****plan library****/
//action reqtrade() inserts a new base belief pertaining to the
//pending new request price
+!requestprice(Type,Mstk,Mprice,Magentid) : counter(Mstk, Price) = newbelief("pendtrade("Type,Mstk,Mprice,Magentid,"").)

//price changed or new price input, update counter price and check
//for outstanding trade
+!newprice(Ctr,Newprice) : counter(Ctr, Price) & updctr(Ctr, Newprice) & checktrade(Ctr, Newprice).

//action reportprice() return a trigger to trade execution agent to
//inform the current price change
+!checktrade(Cter,Cprice) : pendtrade(buy,Cter,Reqprc,Agent) & Cprice ≤ Reqprc reportprice(Cter,Cprice,Agent).
+!checktrade(Cter,Cprice) : pendtrade(sell,Cter,Reqprc,Agent) & Cprice ≥ Reqprc reportprice(Cter,Cprice,Agent).

Program 5.3 - 6 Sample program of MABS
5.4 Other Constraint Agents

This section provides three other examples on agent-based research that involve constraint in one way or another. A brief description for each research is given and any resemblance to 

ConstraintAgentSpeak will be highlighted and its role in its respective platform will be emphasized.

5.4.1 ¹EXCALIBUR

In EXCALIBUR - an adaptive constraint-based agent, constraint is utilised as a ‘tool’ for agent’s planning system which in some respect is similar to the approach adopted in ConstraintAgentSpeak. Under EXCALIBUR, the constraints serve as the specifications that can be used to determine whether the final plan is applicable. In this context, values generated from constraint solving are collected through an objective function which would then be used to evaluate the quality of a plan. Based on the quality evaluated, adjustments are made accordingly to refine the plan to its desired state. Thus essentially, constraints serve as catalyst for an agent planning system to achieve its goal.

5.4.2 ²MarCon

In MarCon algorithm, it adopts a market oriented agent based approach to distributed constraint satisfaction problem. Two major classes of agent exist in MarCon: constraint agent and variable agent. Constraint agent interacts with another constraint agent via variable agents in which they share common interest. The variable agents on the other hand, will provide feedback that enables the constraint to shrink or converge to a solution. The constraint also serves as the utility function that determines the set of

¹EXCALIBUR is available at the URL: http://www.first.gmd.de/concorde/EXCALIBURhome.html
²MarCon is available at the URL: http://www.erim.org/~van/papers.htm
value assignment to variables of interest. Assignment that yields higher utility will eventually become the preferred set of assignment. Basically this is an agent-based approach for solving constraint satisfaction problem rather than deploying constraint computation to enhance and improve performance of an agent.

5.4.3 3DENEGOT

In the extension of DENEGOT, the focus is on the negotiation among agents to arrive at a satisfying solution to a constrained problem. In the DENEGOT architecture, each agent is regarded as an autonomous entity with its own constraints as well as available resources. Collectively these agents are able to communicate and exchange information to determine whether a constraint problem be solved locally or appeal to other agents for spare resources that would assist in solving the locally constrained problem. DENEGOT is similar to MarCon in term of its objective: using agent-based approach to solve constrained problem. However the two differ by the method employed in the respective approach. MarCon uses a utility function to assist in converging to a solution while DENEGOT exchanges information to appeal for additional resource to solve locally constrained problem.

5.4.4 4AKL – AGENTS and Penny

AKL (AGENTS Kernel Language) is a concurrent constraint programming language developed to program efficient parallel scheme (through a group of constraint-based agents) for parallel performance in concurrent constraint system. AGENTS (for integer finite domain constraint) and Penny are implementations of AKL that provide a programming environment with built-in agents and libraries support. Computation within

---

3DENEGOT is available at the URL: http://dis.cs.umass.edu/research.arm.html
4AKL – AGENTS and Penny is available at the URL: http://www.sics.se/isl/akl/
the AKL concurrent constraint solving is executed by AKL agents that interact through stores of constraint. In another perspective, these AKL agents can be viewed as a set of constraints that drive the concurrent constraint processing.

5.5 Conclusion

This chapter provides a discussion in substantial details two sample applications coded using ConstraintAgentSpeak. The sample codes illustrate how an agent can be created in a declarative manner. Agents with respective functionality from the two samples are specified for its beliefs and its functional plans using the high-level agent specification language. Low-level, indivisible primitive actions are implemented using special routines (coded using procedural language) and can be invoked from the body of a plan. Action routines are the only non-declarative components of the ConstraintAgentSpeak specification language and in most occasion, require customisation to cater for specific purposes.
Chapter 6

Conclusion

6.1 Overview of research

The research reported in this thesis develops the concept of incorporating constraint-based processing into the specification of an agent. It provides the operational semantics for an agent specification language, technical description on prototype implementation of an interpreter for the specification language and illustrative sample programs coded using the specification language. The proposed conceptual framework represents a confluence of:

- AgentSpeak(L) for programming and specification of a BDI agent.
- Constraint directed computation with improved expressive power as well as higher precision in term of specification.

The core work reported here is to amalgamate constraint-based computation and agent programming to form a more realistic and pragmatic combination. The essential idea of agent programming is to offer a more intelligent computing mechanism to cater to increasingly dynamic and flexible computing environments. The lack of fundamental computing efficiency (specialized computing techniques such as constraint processing) in most application problems has made deployment of agent-based technology hard to be justified. The significance of constraint processing has been particularly useful for certain specialized areas of application (such as production scheduling, logistic scheduling etc). However computing efficiency has not been able to provide answers for the requirements
of intensive observation or monitoring of the environment in order to cater to the highly 'volatile' working environments. The primary objective of the thesis is to investigate the viability of having an agent programming language that provides an intelligent framework and a domain specific constraint framework equipped with specialized computation technique required in certain computing environments. The ConstraintAgentSpeak is the result of the work described in this thesis with the capabilities of providing richer data structures for programming a BDI agent.

6.2 Contribution

The contribution of the work described in this thesis is two-fold and can be divided into two main sections based on its impact on the research community and industry practitioners.

6.2.1 Contribution to research

It was the initial intention of this thesis to explore the possibilities of developing an intelligent computing environment to create a more flexible solution provider rather than a rigid, routine problem solver. To address such flexible requirement, the primary concern is a computing framework with sufficient intelligent attributes coupled with a competent computing mechanism for the relevant application area. The endeavor described here is doing exactly the work of creating an integrated prototype (intelligent agent framework and constraint processing) that meets this basic requirement. The prototype is hoped to serve as a foundation framework that can be used for comparison purposes or can be improved on by other researchers in future. From a high level, abstract theoretical perspective, it is an acceptable assumption that such association of agent-based technology and constraint-based processing is only one of the possible contribution
to a more intelligent and yet workable computing framework. As the bulk of this work is based on the BDI architecture which is a form of deliberative framework, it is logical to highlight the possibility of other agent platforms such as reactive agent could also be improvised or modified to offer:

- An equivalent model with customization caters for particular segment of application.

or

- An improved framework that provide for greater computing efficiency.

6.2.2 Contribution to Practitioners

It is hoped that this research work serves not only as a viable proof of the possibilities to have such a working combination. The prototype interpreter illustrates the conceptual integration and also served as a bridging tool to transfer the theoretical foundations in the area of agent-based technology into a more practical framework suitable for more realistic application. This is especially significant in view of the increasing demand for computing software with greater flexibility as well as ability for higher level of self-improvisation. The incorporation of constraint processing provides the prospect of more intelligent application of this specialized computing techniques. As the current trend of computing development is to achieve better and greater autonomy in term of operation efficiency, the ability to take advantage of seamlessly merged computing frameworks to create intelligent computing environment has become increasingly important. Agent-based computing model has been frequently rejected by many industry practitioner mainly because of only having sound theoretical framework but lack of realistic commercial application value. However, the conceptual integration proposed in this work has at least partially addressed the above issue by having a combined framework with:
• agent-based technology as the central control strategy.
• constraint processing as the efficient computation strategy.

Current commercial computing technology which tend to focus on efficient routine solver for a particular set of problem will be benefit from the intelligent attributes which endow in all agent-based technology. The prototype interpreter will hope to provide insight on the linkage utilized to integrate the two disparate computing paradigms. On an overall basis, such integration of framework is hoped to induce initiatives to create a more declarative, flexible computing environment with more comprehensive problem solving power in the respective application domain.

6.3 Limitations

From the abstract, high level perspective, ConstraintAgentSpeak discussed in this thesis represents a generic model which can be treated as a general framework to mould different variants of ConstraintAgentSpeak. Different variants here refer to different versions of ConstraintAgentSpeak that apply to different domain areas and applications. It is important to highlight the term constraint-base agent has implied that the ConstraintAgentSpeak discussed in this thesis is applicable only to program agent for computing environment that involved constraint processing. In addition, the kind of constraint domain involved will further restrict the type constraint-based computation that can be incorporated into the agent framework. Thus in terms of application deployment, the main restriction on the use of ConstraintAgentSpeak to specify and create an agent is directly determine by the class of constraint domain associates with the agent framework.

From the point of prototype implementation described in this thesis, chapter 4 has provided a detailed report on the implementation of a variant of ConstraintAgentSpeak in
which the constraint domain involved is finite constraint domain. Based on the above assertion, the constraint engine incorporated into this prototype essentially will be able to perform solving only for constraints with variables that has a fixed number of allowable values. Making a comparison between this variant of ConstraintAgentSpeak (with finite domain) to the MABS example (ConstraintAgentSpeak with infinite real arithmetic domain) discussed in chapter 5, it is obvious that a different form of constraint expression is involved in MABS and thus a different form of constraint beliefs have appeared in the resulting agent specifications.

The different computation domains involved can result in the use of different constraint solvers in an agent framework. Such differences have restricted the intention to develop a standardised agent programming language that is similar to other imperative programming language. ConstraintAgentSpeak has extended the applicability of agent-based technology but unfortunately this scope of application is still dictated by the constraint domain involved.

6.4 Future research

Looking at the current trend of development in the design of complex multi-agent systems, the need of expressive, high-level specification language that can offer efficient computation processing (at least in a specific area of application) has become an instrumental factor to a successful implementation. This is especially true if agent-based technology is to be well accepted by the practitioner and thrive in the commercial application. ConstraintAgentSpeak has been able to meet the above mentioned requirement with limitation. However ConstraintAgentSpeak is predominantly a deliberative framework that based on the BDI architecture. Thus deliberative actions in
the form of symbolic reasoning or symbolic processing has become the main strategy for manipulation of agent’s behaviour.

In order to offer a more complete, comprehensive agent framework, other notions of agent behaviour will need to be considered and embraced. Two elements that ought to receive extensive consideration for improvement and enhancement on the agent’s specification language in order to program for more precise or sensible agent’s behaviour are:

- Situated-based approach or activity-oriented design to accommodate for reactive behaviour that is critical for on-line or real-time requirement.
- Coordination mechanism that focuses on communication and resolution strategy that will be able to provide flawless information exchange among agents.

An ideal solution to a desirable excellent agent programming language would be one that is able to program a hybrid agent that can display the above two attributes and equip with an extensive deliberative inclination.
Appendix A

A.1 Class diagram for thread managers of ConstraintAgentSpeak agent
A.2 Class diagram for trigger event, plan, the generator of relevant/applicable plan and intention of an *ConstraintAgentSpeak* agent
A.3 Class diagram for base belief of ConstraintAgentSpeak agent
A.4 Class diagram for FD constraint solver of ConstraintAgentSpeak agent
A.5 Class diagram for constraint belief of ConstraintAgentSpeak agent
Appendix B

B.1 Automaton for constructing base belief

B.2 Automaton for constructing constraint belief (FD constraint domain)
B.3 Automaton for constructing constraint belief with linear equations or non-linear equations (real arithmetic domain).

B.4 Automaton for constructing agent plan.
Appendix C

This section contains all the Java source codes for the implementation of the `ConstraintAgentSpeak` interpreter. In order to facilitate easy reference on how these source codes are related to one another, they have been arranged in the sequence based on the class diagrams given in Appendix A. A list of the name of the source codes are given at the beginning of each sub-section C1, C2, C3, C4 and C5.

Except for the program `CustSolutionManager.java` (which has been extensively customised to cater to the work of this research), source codes for the constraint processing part of the interpreter are not listed here as they are actually not programs developed in this research work. The programs on constraint processing as indicated in class diagrams A4 and A.5 are downloaded from the Java Constraint Library (JCL) available at URL: [http://liawww.epfl.ch.ch/~torren/Project.JCL](http://liawww.epfl.ch.ch/~torren/Project.JCL)
C.1 Java source codes for thread managers of ConstraintAgentSpeak agent

- AppletAgentManager.java
- AppletIntentManager.java
- AppletEventManager.java
- AppletBelPlanLib.java
- AppletIntentThread.java
- AppletEventThread.java
public class AppletAgentManager extends ASAgent {
    //protected static AppletEventManager eventmanager;
    //protected static AppletIntentManager intentmanager;

    protected AppletEventManager eventmanager;
    protected AppletIntentManager intentmanager;

    protected AppletBelPlanLib beliefsplanlib;
    protected static TextArea outtext;

    public AppletAgentManager(TextArea program, TextArea ta) {
        super("agent1");

        outtext = ta;
        //Generate agent's belief set and plan library from .PSR file
        beliefsplanlib = new AppletBelPlanLib(program);
    } //Constructor

    public void process() {
        register(this);  // add this agent to the register of ALL agents

        runnit = new Thread(this);
        runnit.start();
    }

    public void stop() {
        runnit.stop();
    }

    //Retrieve argument object and insert trigger (eventnode) into event queue
    public void asEventFired(ASAgentEvent e) {
        System.out.println("ASAgent: AASAgentEvent received by " + name + " from " + e.getSource() + " with args " + e.getArgObject());

        ASEventMsg evtmsg = (ASEventMsg)e.getArgObject();
        this.eventmanager.addevent((eventnode)evtmsg.content);
    }

    //Runnable interface for thread
    public void run() {
        //Spawn an event manager and begin to scan for triggering event. The event manager will
        //generate an event thread that handle the reasoning part of an agent.
        eventmanager = new AppletEventManager(this);

        //eventmanager.addevent(AppletTrigParser.parser(AppletTrigTokenizer.tokenizer(trigger)));

        //Spawn an intention manager and begin to scan for active intention. An intention thread
        //will monitor and process current active intention of an agent.
        intentmanager = new AppletIntentManager(this);
    }
} //AppletAgentManager
public class AppletIntentManager{
    protected static intentionlist intentset;
    protected static AppletIntentThread intentthread;
    protected static boolean suspendactive = false; //Event associated
    protected static intplanstack eventintent; //Event associated
    protected static boolean intwait=false;  //Intention thread on
    public ASAgent agent; //Agent that initiates

    public AppletIntentManager(ASAgent an) {
        agent = an;
        intentset = new intentionlist(); //Sets up intention list and
        populations with intention if any

        intentthread = new AppletIntentThread(this);
        intentthread.setDaemon(true);
        intentthread.start();
    }  //Constructor

    public void addintent(intplanstack is) {
        //public synchronized void addintent(intplanstack is) {
        intentset.addEnd(is);
        AppletIntentThread.intentdone = true;
        //notify();
    }  //addintent

    public synchronized void notice() {
        notify();
    }  //notice

    public intplanstack getintent() {
        //public synchronized intplanstack getintent() {
        //for (;;) {
        try {
            AppOut.appout("AppletIntentManager.suspendactive 1: "+AppletIntentManager.
            suspendactive);
            if (AppletIntentManager.suspendactive) {
                AppOut.appout("AppletIntentManager.suspendactive = false;
                intplanstack tmpevtint = AppletIntentManager.eventintent;
                AppOut.appout("AppletIntentManager.intentset-suspendactive (Before):  "+
                AppletIntentManager.intentset.size());
                AppOut.appout("AppletIntentManager.eventintent);  
                AppOut.appout("AppletIntentManager.intentset-suspendactive (After): "+
                AppletIntentManager.intentset.removeNode(AppletIntentManager.
                eventintent);
                AppOut.appout("AppletIntentManager.intentset-suspendactive (After): "+
                AppletIntentManager.intentset.size());
                return tmpevtint;
            } //if
            else {
                AppOut.appout("AppletIntentManager.suspendactive 2: "+AppletIntentManager.
                suspendactive);
                AppOut.appout("AppletIntentManager.intentset.size() 1 : "+
                AppletIntentManager.intentset.size());
                intplanstack intent = intentset.removeFront();
            //**************************
            //Set intention status to active bad pass on to be executed
            if (intent != null) {
                AppOut.appout("AppletIntentManager.intentset.size() 2 : "+
                AppletIntentManager.intentset.size());
                AppOut.appout("intent.suspend : "+intent.suspend);
                AppOut.appout("intent.status : "+intent.status);
            }
        } //try
    }
if (intent.suspend) {
    // Insert back to intention set
    AppletIntentManager.intentset.addEnd(intent);
    return null;
    // wait();
} else {
    intent.status = true;
    return intent;
} // else
} // if
else {
    AppOut.appout("AppletIntentManager.intwait : " + AppletIntentManager.intwait);
    AppletIntentManager.intwait = true;
    AppOut.appout("AppletIntentManager.intwait : " + AppletIntentManager.intwait);
    return null;
    // wait();
} // else
} // try
// catch (InterruptedException ex) {
    // continue;
//}
} // for
} // getIntent

public void finalise() {
// AppOut.appout("AppletIntentThread.isAlive() : " + intentthread.isAlive());
    if (intentthread.isAlive()) {
        intentthread.stop();
    }
} // finalise
} // AppletIntentManager
public class AppletEventManager{
    protected static eventlist eventset; //Event
    protected static AppletEventThread eventthread; //Event
    protected static boolean evtwait=false;  //Event
    public AASAgent agent; //Agent that
    inititates the thread
    public AppletEventManager(AASAgent en) {
        agent = en;
        eventset = new eventlist(); //Sets up event set and populates with
        values if any
        }  //Constructor
    public void addevent(eventnode ne) {
        //public synchronized void addevent(eventnode ne) {
            eventset.addEnd(ne);
            if (AppletEventManager.evtwait)  {
                AppletEventManager.evtwait = false;
                AppletEventThread.eventdone = true;
            }
        }  //addevent
    public synchronized void notice() {
        notify();
    }  //notice
    public eventnode getevent() {
        //public synchronized eventnode getevent() {
            eventnode evt = eventset.removeFront() ;
            AppOut.appout(  .
            "EventManager - AppletEventManager.eventset.size() "+AppletEventManager.eventset.size());
            return evt;
        }  //getevent
    }
if (evt != null) {
    return evt;
} else {
    AppletEventManager.evtwait = true;
    AppOut.appout("EventManager - AppletEventManager.getevent() (AppletEventManager) : WAIT !!!");
    return null;
    //wait();
    
    //} //try
    //catch (InterruptedException ex) {
    //    //continue;
    //}
    //} //for
} //getevent

public void finalise() {
    AppOut.appout("EventManager - eventthread.isAlive() : "+eventthread.isAlive());
    if (eventthread.isAlive()) {
        eventthread.stop();
    }
    AppOut.appout("EventManager - eventthread.isAlive() : "+eventthread.isAlive());
} //finalise

} //AppletEventManager
Program Name : AppletBelPlanLib.java
Program Function : Construction of belief set and plan library by calling
                    SpawnPlanLib.java and SpawnBelSet.java.
                    The belief set and plan library generated are for a
                    single agent running as a single thread.
                    Receive input from parsed textarea, constructs and
                    populates data structures

Last Update : 09 Feb 1999
Code By : Boon

Remarks : protected static beliefnodelist thebeliefset = new
           beliefnodelist();
           protected static plannodelist theplanlib = new plannodelist();
           protected static constrtbellistFD theconbel = new
           constrtbellistFD();
           protected static consaxiomlistFD theconaxiom = new
           consaxiomlistFD();
           protected static consvardomlistFD thevardom = new
           consvardomlistFD();
           protected static constorelistFD theconstore = new
           constorelistFD();
           protected static replysolnlist thesolnlist = new replysolnlist();
           ALL of the above are GLOBAL data structures for a runtime
           AgentSpeak(L) agent.

import java.awt.*;
import java.io.*;
import java.util.StringTokenizer;

public class AppletBelPlanLib{
    protected static beliefnodelist thebeliefset;
    protected static plannodelist theplanlib;
    protected static constrtbellistFD theconbel;
    protected static consaxiomlistFD theconaxiom;
    protected static consvardomlistFD thevardom;
    //Declaration of the global constraint store to be used for constraint
    solving
    protected static constorelistFD theconstore;
    protected static replysolnlist thesolnlist;

    public AppletBelPlanLib(TextArea psrsrc) {
        thebeliefset = new beliefnodelist();
        theplanlib = new plannodelist();
        theconbel = new constrtbellistFD();
        theconaxiom = new consaxiomlistFD();
        thevardom = new consvardomlistFD();
        //Declaration of global constraint store
        theconstore = new constorelistFD();
        thesolnlist = new replysolnlist();

        StringBuffer tknbuffer = new StringBuffer();

        StringBuffer inputstream = new StringBufferInputStream(psrsrc.getText());

        //*****PROCESSING TOKENIZED & SYNTACTICALLY PARSED AgentSpeak(L) FILE
        (psr_file)*****/
        try {
            DataInputStream inline = new DataInputStream(ps);

            String arraystring = "";

            //Input and process program line/statement from .psr file
            while ((arraystring = inline.readLine()) != null) {
                AppOut.appout("BelPlanLib - " + arraystring);
                StringTokenizer arrstr = new StringTokenizer(arraystring, " ");

                //Split statement string into an array of tokens & populates array
```java
String[] stmttoken = new String[arrstr.countTokens()];
int nooftoken = arrstr.countTokens();
//AppOut.appout("BelPlanLib - arrstr.countTokens(1) :
  "+arrstr.countTokens());
for (int i = 0; i <= nooftoken - 1; i++) {
  stmttoken[i] = arrstr.nextToken();
//AppOut.appout("BelPlanLib - arrstr.countTokens(2) :
  "+arrstr.countTokens());
//AppOut.appout("BelPlanLib - stmttoken[i] :
  "+i+-"+stmttoken[i]);
}
//AppOut.appout("BelPlanLib - *stmttoken.length);
//AppOut.appout("BelPlanLib - *stmttoken[0]);

//First token is indicator of belief or plan
if (Integer.parseInt(stmttoken[0]) == 1) {  //BELIEF
  String[] beliefstmt = new String[stmttoken.length - 1];
  SpawnBeliefSet beliefset = new SpawnBeliefSet();
  beliefnode newbelief = new beliefnode();
  //AppOut.appout("BelPlanLib - Creating Beliefs »»»");
  //for (int b=0; b<stmttoken.length; b++) {
  //  AppOut.appout("BelPlanLib - stmttoken.length : 
  //  +stmttoken.length);
  //  AppOut.appout("BelPlanLib - stmttoken[b] :
  //  +b+-"+stmttoken[b]);
  //}
  //Exclude first token which indicate 1-Belief
  for (int k = 0; k <= stmttoken.length - 2; k++) {
    beliefstmt[k] = stmttoken[k+1];
    //AppOut.appout("BelPlanLib - & "+beliefstmt[k]);
    //AppOut.appout("BelPlanLib - & "+stmttoken[k+1]);
  }
  //AppOut.appout("BelPlanLib - *"+beliefstmt.length);
  //AppOut.appout("BelPlanLib - *"+stmttoken.length);
  //for (int a=0; a<beliefstmt.length; a++) {
  //  AppOut.appout("BelPlanLib - beliefstmt.length : 
  //  +beliefstmt.length);
  //  AppOut.appout("BelPlanLib - beliefstmt[a] :
  //  +a+-"+beliefstmt[a]);
  //}
  //Build data structures and populate to respective
  //section of the belief node
  newbelief = beliefset.spawnbelief(beliefstmt);
  //AppOut.appout("BelPlanLib - "+newbelief.beliefAtom.predSymbol);
  //AppOut.appout("BelPlanLib - "+newbelief.beliefAtom.noofTerm);
  //Append the new belief node generated to the belief set
  thebeliefset.addEnd(newbelief.beliefAtom);
  }
else if (Integer.parseInt(stmttoken[0]) == 2) {  //CONSTRAINT BELIEF
  String[] conbelstmt = new String[stmttoken.length - 1];
  SpawnConBel conbelset = new SpawnConBel();
  constrbelnodeFD newconbel = new constrbelnodeFD();
  //AppOut.appout("BelPlanLib - Creating Constraint Beliers »»»");
  //for (int b=0; b<stmttoken.length; b++) {
  //  AppOut.appout("BelPlanLib - stmttoken.length : 
  //  +stmttoken.length);
  //  AppOut.appout("BelPlanLib - stmttoken[b] :
  //  +b+-"+stmttoken[b]);
  //}
  //Exclude first token which indicate 2-Constraint Belief
  for (int k = 0; k <= stmttoken.length - 2; k++) {
    conbelstmt[k] = stmttoken[k+1];
    //AppOut.appout("BelPlanLib - *conbelstmt.length);
    //AppOut.appout("BelPlanLib - *conbelstmt[0]);
    //for (int a=0; a<conbelstmt.length; a++) {
    //  AppOut.appout("BelPlanLib - conbelstmt.length : 
    //  +conbelstmt.length);
    //  AppOut.appout("BelPlanLib - conbelstmt[a] :
    //  +a+-"+conbelstmt[a]);
    //}
    //Build data structures and populate to respective section of the
    //belief node
```
newconbel = conbelset.spawnconbelset(conbelstmt);

//AppOut.appout("BelPlanLib - newconbel.constbel.belPredSym");
//AppOut.appout("BelPlanLib - newconbel.constbel.belNoofTerm");

//Append the new belief node generated to the belief set theconbel.addEnd(newconbel);
}  //else if CONSTRAINT BELIEF
else if (Integer.parseInt(stmttoken[0]) == 3) {  //PLAN
String[] planstmt = new String[stmttoken.length - 1];
SpawnPlanLib planlib = new SpawnPlanLib();
plannode newplan = new plannode();

//Exclude first token which indicate 3-Plan
for (int j = 0; j <= stmttoken.length - 2; j++) {
  planstmt[j] = stmttoken[j+1];
}  //for

//Build data structures and insert value to respective
//section of the plan node
newplan = planlib.spawnplan(planstmt);

//Append the new plan node generated to the plan library
theplanlib.addEnd(newplan);
}  //else if PLAN
else if (Integer.parseInt(stmttoken[0]) == 5) {  //VARIABLE
DOMAIN
String[] vardomstmt = new String[stmttoken.length - 1];
SpawnVarDom vardomset = new SpawnVarDom();
consvardomFD newvardom = new consvardomFD();

//AppOut.appout("BelPlanLib - Creating Variable Domain >>>");
//for (int b=0; b<stmttoken.length; b++) {
//  AppOut.appout("BelPlanLib — stmttoken.length : "+stmttoken.length);
//for (int b=0; b<stmttoken.length; b++) {
//  AppOut.appout("BelPlanLib - stmttoken[b] : "+b+"-"+stmttoken[b]);
//}

//Exclude first token which indicate 2-Constraint Belief
for (int k = 0; k <= stmttoken.length - 2; k++) {
  vardomstmt[k] = stmttoken[k+1];
}  //for

//AppOut.appout("BelPlanLib - +vardomstmt.length");
//AppOut.appout("BelPlanLib - +vardomstmt[0]");
//for (int a=0; a<vardomstmt.length; a++) {
//  AppOut.appout("BelPlanLib - vardomstmt.length : "+vardomstmt.length)
//for (int a=0; a<vardomstmt.length; a++) {
//  AppOut.appout("BelPlanLib - vardomstmt[a] : "+a+"-"+vardomstmt[a]);
//}

//Build data structures and populate to respective section of the
belief node
newvardom = vardomset.spawnvardom(vardomstmt);

//AppOut.appout("BelPlanLib - +newvardom.varSymbol");
//AppOut.appout("BelPlanLib - +newvardom.noofDomVal");

//Append the new belief node generated to the belief set thevardom.addEnd(newvardom);
}  //else if VARIABLE DOMAIN
else if (Integer.parseInt(stmttoken[0]) == 6) {  //CONSTRAINT AXIOM
String[] conaxmstmt = new String[stmttoken.length - 1];
SpawnConAxiom conaxmset = new SpawnConAxiom();
consaxiomFD newconaxm = new consaxiomFD();

//AppOut.appout("BelPlanLib - Creating Constraint Axiom >>>");
//for (int b=0; b<stmttoken.length; b++) {
//  AppOut.appout("BelPlanLib - stmttoken.length : "+stmttoken.length);
//for (int b=0; b<stmttoken.length; b++) {
//  AppOut.appout("BelPlanLib - stmttoken[b] : "+b+"-"+stmttoken[b]);
//}

//Exclude first token which indicate 2-Constraint Belief
for (int k = 0; k <= stmttoken.length - 2; k++) {
  conaxmstmt[k] = stmttoken[k+1];
}  //for

//AppOut.appout("BelPlanLib - +conaxmstmt.length");
//AppOut.appout("BelPlanLib - +conaxmstmt[0]");
for (int a=0; a<conaxmstmt.length; a++) {
  AppOut.appout("BelPlanLib - conaxmstmt.length : "+conaxmstmt.length);
  AppOut.appout("BelPlanLib - conaxmstmt[a] : "+a=""+conaxmstmt[a]);
}

//Build data structures and populate to respective section of the belief node
newconaxm = conaxmset.spawnconsaxiom(conaxmstmt);

AppOut.appout("BelPlanLib - "+newconaxm.noofVarLtrl);
AppOut.appout("BelPlanLib - "+newconaxm.noofDomVal);

//Append the new belief node generated to the belief set
conaxiom.addEnd(newconaxm);
}
//else if CONSTRAINT AXIOM
}
//while not null (NOT EOF)
}
//try
catch (IOException e) {
  System.err.println(e);
  return;
}
//catch
//AppletBelPlanLib (Constructor)
public class AppletIntentThread extends Thread{
    protected int planstack intentpick;
    protected static boolean intentdone; //To control multithreading
    temporary
    private AppletIntentManager intentmgr;

    public AppletIntentThread(AppletIntentManager im) {
        intentmgr = im;
        intentpick = null;
        intentdone = true;
    } //Constructor

    public synchronized void run() {
        for (; ;) {
            if (AppletIntentThread.intentdone) {
                AppOut.appout("IntentThread - AppletIntentManager.intentset.size() : "+
                AppletIntentManager.intentset.size());
                AppletIntentThread.intentdone = false; //Reset true after
                execution has been completed
                AppOut.appout("IntentThread - AppletIntentThread.intentdone : "+
                AppletIntentThread.intentdone);
                intentpick = intentmgr.getintent();

                if (intentpick != null) {
                    AppOut.appout("IntentThread - this.intentpick.status : "+this.intentpick.
                    status);

                    //If intention status is ACTIVE
                    if (this.intentpick.status == true) {
                        AppletIntExecution intexecution = new AppletIntExecution(this.
                        intentpick, AppletAgentManager.outtext, this.intentmgr.agent);
                        intexecution.start();
                    } //if
                    } //if
                } //for
            } //run
        } //AppletIntentThread
public class AppletEventThread extends Thread{
    protected eventnode eventpick;
    protected static boolean eventdone;  //To control multithreading temporary
    private AppletEventManager eventmgr;

    public AppletEventThread(AppletEventManager em) {
        eventmgr = em;
        eventpick = null;
        eventdone = true;
    }  //Constructor

    public synchronized void run() {
        for (; ;  )  {
            //AppOut.appout("EventThread - AppletEventManager.eventset.size() 1 : "+AppletEventManager.eventset.size());
            //AppOut.appout("EventThread - AppletEventThread.eventdone 1 : " +AppletEventThread.eventdone);
            if (AppletEventThread.eventdone) {
                AppOut.appout("EventThread - AppletEventManager.eventset.size() 1 : " +
                AppletEventManager.eventset.size());
                AppOut.appout("EventThread - AppletEventThread.eventdone 1 : " +
                AppletEventThread.eventdone);
                AppOut.appout("EventThread - AppletEventManager.eventset.size () 2 : " +
                AppletEventManager.eventset.size());
                AppOut.appout("EventThread - this.eventpick.evtPredSym : " +this.eventpick.
                evtPredSym);
                for (int z=C ;  z < this.eventpick.evtTerm.length; z++) {
                    AppOut.appout("EventThread - this.eventpick :"*
                    +this.eventpick.evtTerm[z][0]);
                }
                AppletGenRelPlan genreiplan = new AppletGenRelPlan(  
                        AppletBelPlanLib.theplanlib, this.eventpick,  
                        this.eventmgr.agent);  
                genreiplan.start();
            }  //if
            size());
            //if
        }  //for
    }  //run
}  //AppletEventThread
C.2 Java source codes for trigger event, plan, the generator of relevant/applicable plan and intention of an **ConstraintAgentSpeak** agent

- AppletGenRelPlan.java
- AppletGenApplPlan.java
- AppletIntExecution.java
- eventnode.java
- eventlist.java
- relplannode.java
- relplanlist.java
- intplanstack.java
- plannode.java
- plannodelist.java
- intentionlist.java
- bindingstack.java
- bindingnode.java
- invocation.java
- bodynodelist.java
- bodynode.java
- contextnode.java
- contextnodelist.java
public class AppletGenRelPlan extends Thread {
    protected plannodelist planlib; //Agent's plan library
    protected eventnode event;
    protected relplanlist relplans; //Relevant plans for the event
    public ASAgent eventagent; //Name of agent that initiates the thread

    public AppletGenRelPlan(plannodelist plans, eventnode evt, ASAgent et) {
        planlib = plans;
        event = evt;
        relplans = new relplanlist();
        eventagent = et;
    }  //Constructor

    public void run() {
        //Enumerate through the plan library
        /*AppOut.appout("GenRelPlan - this.planlib.size() 1 :
        +this.planlib.size());
        */
        //AppOut.appout("GenRelPlan - this.event :
        +this.event.evtOper);
        //AppOut.appout("GenRelPlan - this.event :
        +this.event.evtIdentifier);
        //AppOut.appout("GenRelPlan - this.event :
        +this.event.evtPredSym+'
        */
        for (int i = 0; i <= this.planlib.size() ; i++) {
            /*AppOut.appout("GenRelPlan - this.planlib.size() 2 :
            Plan"+i+"-"+this.planlib.size());
            if (this.planlib.peekNode(i).invEvent.invOper == this.event.evtOper
            & this.planlib.peekNode(i).invEvent.invIdentifier == this.event.
            evtIdentifier ) {
            */
            literal tmpinv = new literal();
            literal tmpevt = new literal();
            tmpinv.predSymbol = this.planlib.peekNode(i).invEvent.invPredSym;
            tmpinv.noofTerm = this.planlib.peekNode(i).invEvent.invNoofTerm;
            tmpevt = this.planlib.peekNode(i).invEvent.invTerm;
            /*
            for (int a=0; a < this.planlib.peekNode(i).invEvent.invTerm.length;
            a++) {
                for (int b=0; b <
                this.planlib.peekNode(i).invEvent.invTerm[a].length; b++) {
                    tmpevt.term[a][b] =
                    this.planlib.peekNode(i).invEvent.invTerm[a][b];
                }
            */
            tmpevt.predSymbol = this.event.evtPredSym;
            tmpevt.noofTerm = this.event.evtNoofTerm;
            tmpevt = this.event.evtTerm;
            /*
            for (int a=0; a < this.event.evtTerm.length; a++) {
                for (int b=0; b < this.event.evtTerm[a].length; b++) {
                */
        }
tmpevt.term[a][b] = this.event.evtTerm[a][b];

} /* */
/* */
AppOut.appout("GenRelPlan - tmppinv.predSymbol : "+tmppinv.predSymbol);
AppOut.appout("GenRelPlan - tmppinv.noofTerm : "+tmppinv.noofTerm);
for (int a=0; a < tmppinv.term.length; a++) {
    for (int b=0; b < tmppinv.term[a].length; b++) {
        AppOut.appout("GenRelPlan - tmppinv.term :  "+tmppinv.term[a][b]);
    }
}
AppOut.appout("GenRelPlan - tmpevt.predSymbol :  "+tmpevt.predSymbol);
AppOut.appout("GenRelPlan - tmpevt.noofTerm :  "+tmpevt.noofTerm);
for (int a=0; a < tmpevt.term.length; a++) {
    for (int b=0; b < tmpevt.term[a].length; b++) {
        AppOut.appout("GenRelPlan - tmpevt.term :  "+tmpevt.term[a][b]);
    }
}
/* */

//Try unify plan and event
bindingstack tmppbindstk = Unification.unification(tmpinv, tmpevt);

//Insert relevant plan into relevant plan list (relplans)
if (tmppbindstk != null) {
    /* */
    AppOut.appout("GenRelPlan - tmppbindstk.size() :  "+tmppbindstk.size());
    for (int p=1; p<=tmppbindstk.size(); p++) {
        AppOut.appout("GenRelPlan - tmppbindstk.peekNode(p).bindTerm1 :  "+tmppbindstk.peekNode(p).bindTerm1);
        AppOut.appout("GenRelPlan - tmppbindstk.peekNode(p).bindTerm2 :  "+tmppbindstk.peekNode(p).bindTerm2);
        AppOut.appout("GenRelPlan - tmppbindstk.peekNode(p).nextBind :  "+tmppbindstk.peekNode(p).nextBind);
    } /* */
    //New relplannode (duplicate/clone) object for relplanlist
    //Does not refer to the original plannode in the plan library
    relplannode tmppplan = new relplannode();
    tmppplan.planID = this.planlib.peekNode(i).planID;
    tmppplan.invEvent = (invocation)
        this.planlib.peekNode(i).invEvent.clone();
    tmppplan.contextlist = (contextnodelist) this.planlib.peekNode(i).
        contextlist.clone();
    tmppplan.bodylist = (bodynodelist) this.planlib.peekNode(i).
        bodylist.clone();
    tmppplan.nextPlan = null;
    tmppplan.bindstack = tmppbindstk;
    this.relplans.addEnd(tmppplan);
}

//if relevant plan
else {
    AppOut.appout("GenRelPlan - NOT a relevant plan for this event");
}
/* */
for (int p=1; p<=this.relplans.size(); p++) {
    AppOut.appout("GenRelPlan - Rel. Plan :  "+this.relplans.peekNode(p).
        bodylist.peekNode(1).bdyPredSym);
}
AppletGenApp1Plan genapplplan = new AppletGenApp1Plan(this.relplans,
    this.event, this.eventagent);
    genapplplan.start();
} //run
}
/* */
/* */
//AppletGenRelPlan thread
AppletGenApplPlan.java

/******************************************************************************
 Program Name : AppletGenApplPlan.java
 Program Function : Thread to generate applicable plan(s) from the relevant plans and their respective relevant unifiers
 Last Update : 18 Jan 1999/15 Sept 1999
 Code By : Boon
 Remarks : TO INVESTIGATE AND MODIFY:
 Setting up of variables and domain values for domain constraint beliefs immediately after a context belief is selected for processing. (20 Oct 1999)
*******************************************************************************/

import JCL.*;

public class AppletGenApplPlan extends Thread {
    protected relplanlist relplans; //Selected relevant plans
    protected relplanlist appplans; //Selected applicable plans for the event
    protected relplannode selapplplan; //Selected final applicable plan
    protected eventnode event;
    protected LiteralNetwork literalNet;
    protected CustSolutionManager solmgr;
    public ASAgent applagent; //Name of agent that initiates the thread

    public AppletGenApplPlan(relplanlist plans, eventnode evt, ASAgent lagc) {
        relplans = plans;
        appplans = new relplanlist();
        selapplplan = null;
        event = evt;
        literalNet = null;
        solmgr = null;
        applagent = lagc;
    } //Constructor

    public void run() {
        //Enumerate through the relevant plan set
        for (int i = 0; i <= relplans.size(); i++) {
            relplannode selrelclan = relplans.peakNode(i); //A reference to the
            actual node object
            boolean applplan = true; //Indicator of an applicable plan
            solmgr = null; // Fresher solution manager
            //New external events, setting up of new constraint network
            constraint store
            if (this.event.literalcons == null) {
                // literalNet : High level constraint network literal representation
                literalNet = new LiteralNetwork();
                literalNet.SetName("Appointment");
                literalNet.SetAuthor("Boon");
            }
            AppOut.appout("GenApplPlan - : Create High Level Constraint Network");
            /***/
            // *** Setting up of constraints in constraint network
            /***/
            // *** The constraint network become the natural constraint store for
            //*** each constraint solving session
            // *** This section is CUSTOMISED for handling of BINARY CONSTRAINTS only! ***/
            if (!AppletBelPlanLib.thevardom.isEmpty()) {
                //initialisation and setting up of constraint network
                with variables and its
                // domain values from AppletBelPlanLib.thevardom (the global variable domain list)
                for (int dm=1; dm<AppletBelPlanLib.thevardom.size(); dm++) {
                    constvardomFD tmpvardom = AppletBelPlanLib.thevardom. peakNode(dm);
            } //if (!AppletBelPlanLib.thevardom.isEmpty())
        } //for (int i=0; i <= relplans.size(); i++)
    } // public void run()
literalNet.AddVariable(tmpvardom.varSymbol);
literalNet.AddDomain(tmpvardom.varSymbol);

//AppOut.appout("GenApplPlan - tmpvardom.varSymbol :
"+tmpvardom.varSymbol);
for (int vl=0; vl<tmpvardom.domVal.length; vl++) {
    literalNet.AddValue(tmpvardom.varSymbol, tmpvardom.domVal[vl]);
    //AppOut.appout("GenApplPlan - tmpvardom.domVal [vl] :
    "+tmpvardom.domVal[vl]);
}  //for
//if vardom is empty

//If constraint axiom list is not empty, initialised
//network with constraint axioms
if ([! AppletBelPlanLib.theconaxiom.isEmpty()) {
    for (int xx=l; xx<=AppletBelPlanLib.theconaxiom.size(); xx++) {
        String varl = AppletBelPlanLib.theconaxiom.peekNode(xx).varLtrl[0];
        String var2 = AppletBelPlanLib.theconaxiom.peekNode(xx).varLtrl[1];
        //AppOut.appout ("GenApplPlan - AppletBelPlanLib. theconaxiom var2 : "+var2);
        conaxmbodylistFD tmpaxmbody = AppletBelPlanLib.theconaxiom.peekNode(xx).axmBody;
        for (int yy=l; yy<=tmpaxmbody.size(); yy++) {
            String vail =
            tmpaxmbody.peekNode(yy).axmBodyTerm[0];
            String val2 =
            tmpaxmbody.peekNode(yy).axmBodyTerm[1];
            //AppOut.appout ("GenApplPlan - tmpaxmbody. peekNode (yy) : "+yy+
            /n+val2);
            if (! literalNet.CheckConstraint(varl, var2, vail, val2)) {
                literalNet.SetConstraint(varl, var2, vail, val2);
                AppOut.appout("GenApplPlan !!!!!! - literalNet. SetConstraint :
                "+varl);
                AppOut.appout("GenApplPlan !!!!!! - literalNet. SetConstraint :
                "+var2);
                AppOut.appout("GenApplPlan !!!!!! - literalNet. SetConstraint :
                "+val1);
                AppOut.appout("GenApplPlan !!!!!! - literalNet. SetConstraint :
                "+val2);
            }
        //for each specific constraint
        }  //for every constraint axioms
    }  //if conaxiom != null

="/*********/
else {
    literalNet = this.event.literalcons ;
}

contextlist.size());

if (selrelplan.contextlist.size() == 1 && selrelplan.contextlist.head.
ctxPredSym.equals("true")) {
    AppOut.appout("Plan context : true");
} else {
    //Enumerate through context list (each context belief) of the
    //relevant plan
    for (int j=l; j <= selrelplan.contextlist.size(); j++) {
        //solmgr = null;  //move to beginning of processing for each
        //new relevant plan
        contextnode tmpnode = selrelplan.contextlist.peekNode(j);
        String tmpvar = new String(); //Temporary store to retain the
        actual VARIABLE
        String tmpvar1 = new String(); //Temporary store to retain the
        actual variables
        String tmpvar2 = new String(); //of the last 2 terms in context
        belief
```java
AppletGenApplPlan.java

Retain the actual value for the last 2 terms before applying substitution
int cntTerm = selrelplan.contextlist.peekNode(j).ctxTerm.length;
if (cntTerm > 1) {
    tmpvar1 = selrelplan.contextlist.peekNode(j).ctxTerm[cntTerm-2][0];
    tmpvar2 = selrelplan.contextlist.peekNode(j).ctxTerm[cntTerm-1][0];
}
else {
    tmpvar1 = selrelplan.contextlist.peekNode(j).ctxTerm[0][0];
}
if (tmpnode.ctxPredSym.equals("domain") && tmpnode.ctxTerm.length == 1) {
    tmpvar = tmpnode.ctxTerm[0][0];
}

Applying relevant unifiers to a context, a context belief of relevant plan
Substitution.substitution(selrelplan.bindstack, selrelplan.
contextlist.peekNode(j).ctxTerm);

if the TVAR is substituted, indicates domain of TVAR is restricted to the substituted value
if predicate is domain and term is not TVAR

Set domain for constraint network : for domain(Var) predicate only.
***** if (selrelplan.contextlist.peekNode(j).ctxPredSym.equals("domain")) {*****
    if the single term is still a variable, indicate not a preassigned value constraint
    if (Character.isUpperCase(selrelplan.contextlist.peekNode(j)).
        ctxTerm[0][0].charAt(0) == 'A') { TAAR;
        for (int cd=0; cd < AppleBelPlanLib.thecontextbel.size(); cd++) {
            AppOut.appout("GenApplPlan - AddVariable tmpvar-domain :
                            AppleBelPlanLib.thecontextbel.peekNode.cd .
                            thecontextbel.bePredSym;")
            AppOut.appout("GenApplPlan - AddVariable tmpvar :
                            AppleBelPlanLib.
                            thecontextbel.peekNode(cd).constbel[2];")
            AppOut.appout("GenApplPlan - AddValue tmpvar :
                            AppleBelPlanLib.
                            blackTerm[]");
        } for
        Associates variable to domain
        literalNet.SetVariableDomain tmpvar, tmpvar;
    }
    else {
        Domain exist, Add new value only : ?????
        if (literalNet.CheckVariable tmpvar) {
            literalNet.AddVariable tmpvar;
        }
    }
    for (int dv=1;
    dv<AppleBelPlanLib.thecontextbel.peekNode.cd .bodylist.size(); dv++) {
        if ( literalNet.CheckValue tmpvar, AppleBelPlanLib.
            thecontextbel.peekNode.cd .bodylist.peekNode[dv].blackTerm[];
        literalNet.AddValue tmpvar, AppleBelPlanLib.
        thecontextbel.peekNode(cd).bodylist.peekNode[dv].blackTerm[];
    } for
}
```
/**/ // Associates variable to domain
/**/ literalNet.SetVariableDomain(tmpvar, tmpvar);
/**/ } // else ??????
/**/ } // if
/**/ } // for
/**/ continue;
/**/ } // if Uppercase
/**/ else {
/**/ if (! literalNet.CheckDomain(tmpvar)) {
/**/ if (! literalNet.CheckVariable(tmpvar)) {
/**/ literalNet.AddDomain(tmpvar);
/**/ literalNet.AddValue(tmpvar, selrelplan.contextlist.peekNode(j).ctxTerm[0][0]);
/**/ // Associates variable to domain
/**/ literalNet.SetVariableDomain(tmpvar, tmpvar);
/**/ }
/**/ } // else if Uppercase
// if domain belief
// if predicate is domain and term is not TVAR
*******************************************
// Enumerate through terms of above context belief (j) to determine
// if it is ground
boolean groundbel = true; // Indicator for ground context belief
for (int k=0; k < selrelplan.contextlist.peekNode(j).ctxTerm.length; k++) {
   if (Character.isUpperCase(selrelplan.contextlist.peekNode(j).ctxTerm[k][0].charAt(0))) {
      groundbel = false;
      break;
   } // if
} // for each context belief's term (k)
// Determine if the context belief (j) is logical consequence
if (groundbel) {
   // Indicator of current context belief is GROUND and LOGICAL CONSEQUENCE
   boolean grdlgcsq = false;
   // Enumerate through base beliefs in beliefs set
   for (int l=1; l <= AppletBelPlanLib.thebeliefset.size(); l++) {
      beliefnode tmpbelnode = AppletBelPlanLib.thebeliefset.peekNode(l);
      // Similar functor and arity
      if (tmpbelnode.beliefAtom.predSymbol.equals(selrelplan.contextlist.peekNode(j).ctxPredSym) &&
         tmpbelnode.beliefAtom.noofTerm == selrelplan.contextlist.peekNode(j).ctxNoofTerm) {
         // Indicator of current context belief (j) is log. conseq. of current base belief (l)
         boolean logiconseq = true;
         for (int m=0; m < tmpbelnode.beliefAtom.term.length; m++) {
            if (tmpbelnode.beliefAtom.term[m][0].equals(selrelplan.contextlist.peekNode(j).ctxTerm[m][0])) {
               logiconseq = false;
            } // if
            break; // Stop scanning through the remaining terms as the current context belief cannot be the
logical consequence of
//current base belief
}
    } //if equal term

} //for each base belief term and context belief term (m)

//If the current context belief is logical consequence of
current base belief
if (logiconseq) {
    grdlgcsq = true; //Context belief(j) is ground and
logical consequence to base belief(1)
break; //Stop scanning through the remaining
base beliefs
    } //if

} //if similar functor & arity (base beliefs set)

} //for each base belief in beliefs set [thebeliefset] (1)

//--BEGIN OF CONSTRAINT PROCESSING for GROUND context
belief*************************************************************************
if (! grdlgcsq) {
    //Enumerate through constraint belief in constraint beliefs set
    for (int c=1; c <= AppletBelPlanLib.theconbel.size(); c++)
        {:
            AppOut.appout("GenApplPlan - AppletBelPlanLib. theconbel.size () :  "+
            AppletBelPlanLib.theconbel.size());
            constrtbelnodePD tmpconbel = AppletBelPlanLib.theconbel.
peekNode(c);

            String tmpvar5 = new String(); //Temporary store to
retain the actual variables
            String tmpvar6 = new String(); //of the last 2 terms in
constraint belief

            //Retain the actual value for the last 2 terms
            int numtermgrd = tmpconbel.constbel.belTerm.length;
            if (numtermgrd >1) {
                tmpvar5 = tmpconbel.constbel.belTerm[numtermgrd-2][0];
                tmpvar6 = tmpconbel.constbel.belTerm[numtermgrd-1][0];
            } else {
                tmpvar5 = tmpconbel.constbel.belTerm[0][0];
            }

            AppOut.appout("GenApplPlan - (tmpvar5) :  "+tmpvar5);
            AppOut.appout("GenApplPlan - (tmpvar6) :  "+tmpvar6);

            literal tmpbelgrd = new literal();
            literal tmpctxgrd = new literal();

            tmpbelgrd.predSymbol = tmpconbel.constbel.belPredSym;
            tmpbelgrd.noofTerm = tmpconbel.constbel.belNoofTerm;
            tmpbelgrd.term = tmpconbel.constbel.belTerm;

            tmpctxgrd.predSymbol = selrelplan.contextlist.peekNode(j).
ctxPredSym;
            tmpctxgrd.noofTerm = selrelplan.contextlist.peekNode(j).
ctxNoofTerm;
            tmpctxgrd.term = selrelplan.contextlist.peekNode(j).ctxTerm;

            bindingstack tmpbdstkgrd = Unification.unification(tmpbelgrd,
tmpctxgrd);

            boolean validconbel = false; //Indicator of current
constraint belief is valid and consistent
            //with respect to constraints
in the constraint store

            int termcntgrd = selrelplan.contextlist.peekNode(j).ctxTerm.
length; //No of term in header predicate
            if (tmpbdstkgrd != null) {
                //value for the substituted last 2 terms
                String second = selrelplan.contextlist.peekNode(j).
ctxTerm[cntterm-2][0];

                //If the HEAD of constraint belief is unifiable with the
current context belief
                //If unifiable (tmpbdstk not null). Proceed with constraint
processing
                if (tmpbdstkgrd != null) {
                    //value for the substituted last 2 terms
                    String second = selrelplan.contextlist.peekNode(j).
ctxTerm[cntterm-2][0];

                    //If the current context belief is logical consequence of
current base belief
                    if (logiconseq) {
                        grdlgcsq = true; //Context belief(j) is ground and
logical consequence to base belief(1)
break; //Stop scanning through the remaining
base beliefs
                    } //if

                } //if similar functor & arity (base beliefs set)
        } //for each constraint belief in constraints set [theconset] (1)
String end = selrelplan.contextlist.peekNode(j).ctxTerm[cntterm-1][0];

/*Var Dom*/ if (! literalNet.CheckDomain(tmpvar5)) {
    literalNet.AddDomain(tmpvar5);
    literalNet.AddValue(tmpvar5, secend);
    //Associates variable to domain
    literalNet.SetVariableDomain(tmpvar5, tmpvar5);
}
else {
    if (!literalNet.CheckVariable(tmpvar5)) {
        literalNet.AddVariable(tmpvar5);
        literalNet.SetVariableDomain(tmpvar5, tmpvar5);
    }
    if (! literalNet.CheckValue(tmpvar5, secend)) {
        literalNet.AddValue(tmpvar5, secend);
    }
    //Associates variable to domain
    literalNet.SetVariableDomain(tmpvar5, tmpvar5);
}

if (! literalNet.CheckDomain(tmpvar6)) {
    if (!literalNet.CheckVariable(tmpvar6)) {
        literalNet.AddVariable(tmpvar6);
        literalNet.SetVariableDomain(tmpvar6, tmpvar6);
    }
    if (! literalNet.CheckValue(tmpvar6, end)) {
        literalNet.AddValue(tmpvar6, end);
    }
    //Associates variable to domain
    literalNet.SetVariableDomain(tmpvar6, tmpvar6);
}
else {
    if (!literalNet.CheckVariable(tmpvar6)) {
        literalNet.AddVariable(tmpvar6);
        literalNet.SetVariableDomain(tmpvar6, tmpvar6);
    }
    if (! literalNet.CheckValue(tmpvar6, end)) {
        literalNet.AddValue(tmpvar6, end);
    }
    //Associates variable to domain
    literalNet.SetVariableDomain(tmpvar6, tmpvar6);
}

if (! literalNet.CheckConstraint(tmpvar5, tmpvar6, secend, end)) {
    literalNet.SetConstraint(tmpvar5, tmpvar6, secend, end);
    literalNet.SetConstraint(tmpvar6, tmpvar6, secend, end);
    literalNet.SetConstraint(tmpvar5, tmpvar5, secend, end);
    literalNet.SetConstraint(tmpvar6, tmpvar5, secend, end);
    literalNet.SetConstraint(tmpvar5, tmpvar6, end, secend);
    literalNet.SetConstraint(tmpvar6, tmpvar5, end, secend);
    literalNet.SetConstraint(tmpvar5, tmpvar5, end, secend);
    literalNet.SetConstraint(tmpvar6, tmpvar6, end, secend);
}

AppOut.appout("GenApplPlan - end : "+end);

//Translates the high level representation into a more efficient
//implementation and performs constraint solving to
generate solution set
Network net = new Network();
net = literalNet.BuildNetwork();

AppOut.appout("GenApplPlan - BuildNetwork()");
solmgr = new CustSolutionManager();

BTSolver solver = new BTSolver();
solver.SetNetwork(net);
AppOut.appout("GenApplPlan - SetNetwork(net)");
solver.SetNumberOfSolutionsToFind(-1);
AppOut.appout("GenApplPlan - SetNumberOfSolutionsToFind(-1)");
solver.SetSolutionManager(solmgr);
AppOut.appout("GenApplPlan - SetSolutionManager(solmgr)");
solver.run();
AppOut.appout("GenApplPlan - run()");

//If solution set if not empty (not null), set valid
constraint belief indicator to true
//to reflect that the current context belief is an
ACCEPTABLE context belief.
//Thus the context belief is also ground and logical
consequence.
if (solmgr.NumOfSolution() > 0) {
    validconbel = true;
}  //if

if (validconbel) {
    grdlgcsq = true;  //Context belief(j) is ground and
logical consequence to constraint belief(c)
    //and is in consistent with the
existing constraint store
    //break;  //Stop scanning through the
remaining constraint beliefs
}  //if validconbel
}  //if tmpbdstkgrd != null
}  //for each constraint belief in constraint belief set
//END OF CONSTRAINT PROCESSING for GROUND context
belief********************************************************************/

//If the current context belief (j) is GROUND but NOT LOGICAL
CONSEQUENCE,
//the current relevant plan is NOT APPLICABLE. Break from the
loop for each
//context belief and move on to next relevant plan
if (! grdlgcsq) {
    applplan = false;
    break;  //Break from the loop for each context belief (j)
}  //if
//if groundbel - there will be no applicable unifiers
######## Else if it is not ground context belief : NOT groundbel
########
else {
    //Indicator of current context belief is unifiable and a logical
consequence
    boolean unilgcsq = false;

    //Enumerate through base beliefs in beliefs set
    for (int n=1; n <= AppletBelPlanLib.thebeliefset.size(); n++) {
        beliefnode tmpbelnd = AppletBelPlanLib.thebeliefset.peekNode(n);
        literal tmpbel = new literal();
        literal tmpctx = new literal();
        tmpbel .predSymbol = tmpbelnd.beliefAtom.predSymbol;
        tmpbel.noofTerm = tmpbelnd.beliefAtom.noofTerm;
        tmpbel.term = tmpbelnd.beliefAtom.term;
        tmpctx.predSymbol = selrelplan.contextlist.peekNode(j).ctxPredSym;
        tmpctx.noofTerm = selrelplan.contextlist.peekNode(j).ctxNoofTerm;
        tmpctx.term = selrelplan.contextlist.peekNode(j).ctxTerm;
        bindingstack tmpbdstk = Unification.unification(tmpbel, tmpctx);

        //If unifiable (tmpbdstk not null). Proceed to next context
belief
        if (tmpbdstk != null) {
            //Insert new bindings into existing bindstack of relevant
plan
            while (tmpbdstk.size() > 0) {
                selrelplan.bindstack.push(tmpbdstk.pop());
            }  //while
        }  //if unilgcsq
context belief \( j \) is unifiable and logic consequence
break; //Stop scanning through the remaining base beliefs
} //if
} //for each base belief in beliefs set \{thebeliefset\} (n)

//BEGIN OF CONSTRAINT PROCESSING for non-GROUND context belief
****************************************************************************/
if (! unilgcsq) {
  //Enumerate through constraint belief in constraint beliefs set
for (int cc=1; cc <= AppletBelPlanLib.theconbel.size(); cc++) {
  constrtbelnodeFD tmpconbel = AppletBelPlanLib.theconbel.
  peekNode(cc);
  String tmpvar3 = new String(); //Temporary store to retain the actual variables
  String tmpvar4 = new String(); //of the last 2 terms in constraint belief
  //Retain the actual value for the last 2 terms
  int numterm = tmpconbel.constbel.belTerm.length;
  if (numterm > 1) {
    tmpvar3 = tmpconbel.constbel.belTerm[numterm-2][0];
    tmpvar4 = tmpconbel.constbel.belTerm[numterm-1][0];
  } else {
    tmpvar3 = tmpconbel.constbel.belTerm[0][0];
  }

  AppOut.appout("GenApplPlan - (tmpvar3) ; +tmpvar3;
  AppOut.appout("GenApplPlan - (tmpvar4) ; +tmpvar4;

  literal tmpbel = new literal();
  literal tmpctx = new literal();
  tmpbel.predSymbol = tmpconbel.constbel.belPredSym;
  tmpbel.noofTerm = tmpconbel.constbel.belNoofTerm;
  tmpbel.term = tmpconbel.constbel.belTerm;

  tmpctx.predSymbol = selrelplan.contextlist.peekNode(j).
  ctxPredSym;
  tmpctx.noofTerm = selrelplan.contextlist.peekNode(j).
  ctxNoofTerm;
  tmpctx.term = selrelplan.contextlist.peekNode(j).ctxTerm;

  bindingstack tmpbdstk = Unification.unification(tmpbel,
  tmpctx);
  boolean validconbel = false; //Indicator of current constraint belief is valid and consistent
  //with respect to constraints in the constraint store
  int termcnt = selrelplan.contextlist.peekNode(j).ctxTerm.
  length; //No of term in header predicate
  //If unifiable (tmpbdstk not null). Proceed to constraint processing
  if (tmpbdstk != null) {

    AppOut.appout("*GenApplPlan - (selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-2][0]) ;
    *selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-2][0]);
    AppOut.appout("*GenApplPlan - (selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-1][0]) ;
    *selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-1][0]);

    //if (tmpbdstk.size()==0) {
    //For the last 2 terms
    /**Var Domain**/
    for (int tc=termcnt-2; tc<termcnt; tc++) {
      if (Character.isUpperCase(selrelplan.contextlist.
        peekNode(j).ctxTerm[tc][0].charAt(0))) {
/*
   if (! literalNet.CheckDomain(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0])) {
     if (!literalNet.CheckVariable(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0])) {
       literalNet.AddVariable(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]);
     }
     literalNet.AddDomain(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]);
   }

   AppOut.appout("GenApplPlan - literalNet.AddVariable : \"+selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]\"; \n   for (int cd=1; cd <= AppletBelPlanLib.theconbel.size(); cd++) {
     if (! literalNet.CheckDomain(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0])) {
       literalNet.AddDomain(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]);
     }
     literalNet.AddValue(bodylist.size() ***** \"+AppletBelPlanLib.theconbel.peekNode(cd).constbel.belPredSym\"; \n     for (int dv=1; dv<=AppletBelPlanLib.theconbel.peekNode(cd).bodylist.size(); dv++) {
       if (! literalNet.CheckVariable(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0])) {
         literalNet.AddVariable(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]);
       }
       literalNet.SetVariableDomain(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0], selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]);
     }
     //for
     //if
     //for
     //Associates variable to domain
     if (AppletBelPlanLib.theconbel.peekNode(cd).constbel.belPredSym.equals("domain") && AppletBelPlanLib.theconbel.peekNode(cd).constbel.belTerm[0][0].equals(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0])) {
       for (int dv=1; dv<=AppletBelPlanLib.theconbel.peekNode(cd).bodylist.size(); dv++) {
         if (! literalNet.CheckValue(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0], AppletBelPlanLib.theconbel.peekNode(cd).bodylist.peekNode(dv).belTerm[0])) {
           literalNet.AddValue(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0], AppletBelPlanLib.theconbel.peekNode(cd).bodylist.peekNode(dv).belTerm[0]);
         }
       }
       //for
     }
   }
   //if
   //Domain exist, Add new value only !
   } else {  //Domain exist, Add new value only !
     if (!literalNet.CheckVariable(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0])) {
       literalNet.AddVariable(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]);
     }
     for (int cd=1; cd <= AppletBelPlanLib.theconbel.size(); cd++) {
       if (AppletBelPlanLib.theconbel.peekNode(cd).constbel.belPredSym.equals("domain") && AppletBelPlanLib.theconbel.peekNode(cd).constbel.belTerm[0][0].equals(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0])) {
         for (int dv=1; dv<=AppletBelPlanLib.theconbel.peekNode(cd).bodylist.size(); dv++) {
           if (! literalNet.CheckValue(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0], AppletBelPlanLib.theconbel.peekNode(cd).bodylist.peekNode(dv).belTerm[0])) {
             literalNet.AddValue(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0], AppletBelPlanLib.theconbel.peekNode(cd).bodylist.peekNode(dv).belTerm[0]);
           }
         }
       }
     }
     //for
     //if
     //if
     //for
     //Associates variable to domain
     literalNet.SetVariableDomain(selrelplan.contextlist.peekNode(j).ctxTerm[tc][0], selrelplan.contextlist.peekNode(j).ctxTerm[tc][0]);
   }
   //else
   //if Uppercase
/** ***/ else {
/** */ if (tc == selrelplan.contextlist.peekNode(j).ctxTerm.length-2) {
/** */
if (! literalNet.CheckDomain(tmpvar1)) {
/** */
if (!literalNet.CheckVariable(tmpvar1)) {
/** */
literalNet.AddVariable(tmpvar1);
/** */
}
/** */
literalNet.AddDomain(tmpvar1);
/** */
contextlist.peekNode(j).ctxTerm[tc][0]);
/** */
//Associates variable to domain
/** */
literalNet.SetVariableDomain(tmpvar1, selrelplan.
/** */
AppOut.appout("GenApplPlan ****** - literalNet.CountValues(tmpvar1));
/** */
/** */
else if (tc == selrelplan.contextlist.peekNode(j).ctxTerm.length-1) {
/** */
if (! literalNet.CheckDomain(tmpvar2)) {
/** */
if (!literalNet.CheckVariable(tmpvar2)) {
/** */
literalNet.AddDomain(tmpvar2);
/** */
}
/** */
literalNet.AddValue(tmpvar1, selrelplan.
/** */
contextlist.peekNode(j).ctxTerm[termcnt-2][0][0]);
/** */
//Associates variable to domain
/** */
literalNet.SetVariableDomain(tmpvar1, tmpvar1);
/** */
}
/** */
else {
/** */
if (!literalNet.CheckVariable(tmpvar2)) {
/** */
literalNet.AddVariable(tmpvar2);
/** */
}
/** */
if (! literalNet.CheckValue(tmpvar2, selrelplan.
/** */
contextlist.peekNode(j).ctxTerm[termcnt-1][0][0]);
/** */
//Associates variable to domain
/** */
literalNet.SetVariableDomain(tmpvar2, tmpvar2);
/** */
}
/** */
else if (tc == selrelplan.contextlist.peekNode(j).
/** */
contextlist.peekNode(j).ctxTerm[termcnt-1][0][0]) {
/** */
AppOut.appout("GenApplPlan ****** - literalNet.AddValue : "+tmpvar1+" = "+
/** */
literalNet.CountValues(tmpvar1));
/** */
/** */
else if (tc == selrelplan.contextlist.peekNode(j).
/** */
contextlist.peekNode(j).ctxTerm[termcnt-1][0][0]) {
/** */
if (! literalNet.CheckDomain(tmpvar2)) {
/** */
if (!literalNet.CheckVariable(tmpvar2)) {
/** */
literalNet.AddDomain(tmpvar2);
/** */
}
/** */
literalNet.AddValue(tmpvar2, selrelplan.
/** */
contextlist.peekNode(j).ctxTerm[termcnt-1][0][0]);
/** */
//Associates variable to domain
/** */
literalNet.SetVariableDomain(tmpvar2, tmpvar2);
/** */
}
/** */
else {
/** */
if (!literalNet.CheckVariable(tmpvar2)) {
/** */
literalNet.AddVariable(tmpvar2);
/** */
}
/** */
}
/** */
else if (tc == selrelplan.contextlist.peekNode(j).ctxTerm.length-2) {
/** */
if (! literalNet.CheckDomain(tmpvar1)) {
/** */
if (!literalNet.CheckVariable(tmpvar1)) {
/** */
}
/** */
literalNet.AddDomain(tmpvar1);
/** */
contextlist.peekNode(j).ctxTerm[tc][0]);
/** */
//Associates variable to domain
/** */
literalNet.SetVariableDomain(tmpvar1, selrelplan.
/** */
AppOut.appout("GenApplPlan ****** - literalNet.AddValue : "+tmpvar1+" = "+
/** */
literalNet.CountValues(tmpvar1));
/** */
/** */
else if (tc == selrelplan.contextlist.peekNode(j).ctxTerm.length-1) {
/** */
if (! literalNet.CheckDomain(tmpvar2)) {
/** */
if (!literalNet.CheckVariable(tmpvar2)) {
/** */
}
/** */
literalNet.AddDomain(tmpvar2);
/** */
}
/** */
literalNet.AddValue(tmpvar1, selrelplan.
/** */
contextlist.peekNode(j).ctxTerm[termcnt-2][0][0]);
/** */
//Associates variable to domain
/** */
literalNet.SetVariableDomain(tmpvar1, tmpvar1);
/** */
}
/** */
else {
/** */
if (!literalNet.CheckVariable(tmpvar2)) {
/** */
literalNet.AddVariable(tmpvar2);
/** */
}
/** */
}
/** */
else if (Character.isUpperCase(selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-1][0].charAt(0)) {
/** */
if (Character.isUpperCase(selrelplan.contextlist.
/** */
peekNode(j).ctxTerm[termcnt-1][0].charAt(0))) {
/** */
for (int zl=1; zl<=tmpconbel.bodylist.size(); zl++) {
/** */
String value1 = tmpconbel.bodylist.peekNode(zl).belTerm.
/** */
[0];
/** */
String value2 = tmpconbel.bodylist.peekNode(zl).belTerm.
/** */
[1];
/** */
if (! literalNet.CheckConstraint(tmpvar3, tmpvar4,
/** */
value1, value2)) {
/*Set Cons*/
literalNet.SetConstraint(tmpvar3, tmpvar4, value1,
/*Set Cons*/
//Initialised network with constraints from body of
/*Set Cons*/
//based on 4 possible combinations of constraints variables
/*Set Cons*/
if (Character.isUpperCase(selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-2][0].charAt(0)) {
/*Set Cons*/
& Character.isUpperCase(selrelplan.contextlist.
/*Set Cons*/
peekNode(j).ctxTerm[termcnt-1][0].charAt(0))) {
/*Set Cons*/
for (int zl=1; zl<=tmpconbel.bodylist.size(); zl++) {
/*Set Cons*/
String value1 = tmpconbel.bodylist.peekNode(zl).belTerm.
/*Set Cons*/
[0];
/*Set Cons*/
String value2 = tmpconbel.bodylist.peekNode(zl).belTerm.
/*Set Cons*/
[1];
/*Set Cons*/
if (! literalNet.CheckConstraint(tmpvar3, tmpvar4,
/*Set Cons*/
value1, value2)) {
AppletGenApplPlan.java

```java
value2);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+tmpvar3);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+tmpvar4);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+value1);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+value2);

} //for

else if (Character.isUpperCase(selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-2][0].charAt(0))

&& ! Character.isUpperCase(selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-1][0].charAt(0)) ) {

String secterm = selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-1][0];

for (int z2=1; z2<tmpconbel.bodylist.size(); z2++) {

String value1 = tmpconbel.bodylist.peekNode(z2).belTerm[0];

String value2 = tmpconbel.bodylist.peekNode(z2).belTerm[1];

if (secterm.equals(value2)) {

if (! literalNet.CheckConstraint(tmpvar3, tmpvar4, value1, value2)) {

literalNet.SetConstraint(tmpvar3, tmpvar4, value1, value2);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+tmpvar3);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+tmpvar4);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+value1);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+value2);

} //if

else if (! Character.isUpperCase(selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-2][0].charAt(0))

&& ! Character.isUpperCase(selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-1][0].charAt(0)) ) {

String firstterm = selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-2][0];

for (int z3=1; z3<tmpconbel.bodylist.size(); z3++) {

String value1 = tmpconbel.bodylist.peekNode(z3).belTerm[0];

String value2 = tmpconbel.bodylist.peekNode(z3).belTerm[1];

if (firstterm.equals(value1)) {

if (! literalNet.CheckConstraint(tmpvar3, tmpvar4, value1, value2)) {

literalNet.SetConstraint(tmpvar3, tmpvar4, value1, value2);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+tmpvar3);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+tmpvar4);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+value1);
AppOut.appout("GenApplPlan $$$$$$ - literalNet.SetConstraint : "+value2);

} //if

else {

String seclast = selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-2][0];

String last = selrelplan.contextlist.peekNode(j).ctxTerm[termcnt-1][0];

for (int z4=1; z4<tmpconbel.bodylist.size(); z4++) {

String value1 = tmpconbel.bodylist.peekNode(z4).belTerm[0];

String value2 = tmpconbel.bodylist.peekNode(z4).belTerm[1];

if (seclast.equals(value1)) {

AppOut.appout("GenApplPlan $$111 - literalNet.SetConstraint : "+tmpvar3);
```
AppOut.appout("GenApplPlan ^^^^^ - literalNet.SetConstraint : "+tmpvar4); 
if (! literalNet.CheckValue(tmpvar4, value2)) { 
    literalNet.AddValue(tmpvar4, value2); 
} 
if (! literalNet.CheckConstraint(tmpvar3, tmpvar4, valuel, value2)) { 
    literalNet.SetConstraint(tmpvar3, tmpvar4, valuel, value2); 
}
if (! literalNet.CheckValue(tmpvar3, valuel)) { 
    literalNet.AddValue(tmpvar3, valuel); 
} 
if (! literalNet.CheckConstraint(tmpvar3, tmpvar4, valuel, value2)) { 
    literalNet.SetConstraint(tmpvar3, tmpvar4, valuel, value2); 
}
if (last.equals(value2)) { 
    AppOut.appout("GenApplPlan ^^^^^ - literalNet.SetConstraint : "+valuel); 
    AppOut.appout("GenApplPlan ^^^^^ - literalNet.SetConstraint : "+value2); 
} 
if (last.equals(value2)) { 
    AppOut.appout("GenApplPlan ^^^^^11A - literalNet.SetConstraint : "+tmpvar4); 
    AppOut.appout("GenApplPlan ^^^^^ - literalNet.SetConstraint : "+tmpvar4); 
} 
if ((seclast.equals(valuel) && last.equals(value2)) { 
    if (! literalNet.CheckConstraint(tmpvar3, tmpvar4, valuel, value2)) { 
        literalNet.SetConstraint(tmpvar3, tmpvar4, valuel, value2); 
    } 
} 
while (tmpbdstk.size() > 0) { 
    selrelplan.bindstack.push(tmpbdstk.pop()); 
} 
Network net = new Network(); 
solmgr = new CustSolutionManager(); 
solver = new BTSolver(); 
solver.SetNetwork(net); 
solver.SetNumberOfSolutionsToFind(1); 
solver.SetSolutionManager(solmgr); 
solver.run();
// If solution set is not empty (not null), set valid
// constraint belief indicator to true
// to reflect that the current context belief is an
// ACCEPTABLE context belief.
// Thus the context belief is also unifiable and logical
// consequence.
if (solmgr.NumOfSolution() > 0) {
    validconbel = true;
}  // if

if (validconbel) {
    unilgcsq = true; // context belief (j) is unifiable
    and logic consequence
    ///break;      //Stop scanning through the
    remaining constraint beliefs
}  // if tmpbdstk != null
}  // for every constraint beliefs
}  // if

// END OF CONSTRAINT PROCESSING for non-GROUND context
belief******************************************************************************

// If current context belief (j) is not unifiable and hence NOT
// logical consequence,
// the current relevant plan is not applicable. Break from the
// loop for each
// context belief (j) and move on to next relevant plan
if (! unilgcsq) {
    applplan = false;
    break;
}  // if

else if !  groundbel
AppOut.appout("GenApplPlan - Next Context Belief !");
}  // for each context belief (j)
}  // else if selrelplan.contextlist.size() <> 1 or more than 1

AppOut.appout("GenApplPlan - applplan : "+applplan);
// An applicable plan
if (applplan) {
******************************************************************************
// Insert APPLICABLE PLAN into applicable plan list
//*
    if (this.relplans.size() > 1) {
        //*
AppOut.appout("GenApplPlan - APPL PLAN 1 !!!!!!!!!!");
/*
    relplannode newapplplanl = (relplannode) selrelplan.clone(true);
    /*
    newapplplan1.consnetwork = literalNet;
    /*
    if (solmgr != null) {
        /*
        newapplplan1.solutionset = solmgr.SolutionValue();
        /*
        newapplplan1.solutionnum = solmgr.NumOfSolution();
        /*
        newapplplan1.solutionvar = solmgr.SolutionVariable();
        /*
    }
    /*
AppOut.appout("GenApplPlan - APPL PLAN 2 !!!!!!!!!!");
/*
    this.appplans.addEnd((relplannode) selrelplan.clone(true));
    /*
    this.appplans.addEnd(newapplplanl);
    /*
AppOut.appout("GenApplPlan - APPL PLAN 3 !!!!!!!!!!");
/*
}  // if
    /*
else {
  AppOut.appout("GenApplPlan - APPL PLAN 11 !!!!!!!!!!");
  //*
  relplannode newapplplan2 = (relplannode) selrelplan.clone(true);
  //*
  newapplplan2.consnetwork = literalNet;
  //*
  if (solmgr != null) {
    newapplplan2.solutionset = solmgr.SolutionValue();
    newapplplan2.solutionnum = solmgr.NumOfSolution();
    newapplplan2.solutionvar = solmgr.SolutionVariable();
  }
  //*
  AppOut.appout("GenApplPlan - APPL PLAN 22 !!!!!!!!!!");
  //*
  this.appplans.addEnd((relplannode) selrelplan.clone(true));
  //*
  this.appplans.addEnd(newapplplan2);
  //*
  AppOut.appout("GenApplPlan - APPL PLAN 33 !!!!!!!!!!");
  //*
}  //else
  //*
  //this.appplans.addEnd((relplannode) selrelplan.clone(true));
  //*
if (solmgr != null) {
  newapplplan2.solutionset = solmgr.SolutionValue();
  newapplplan2.solutionnum = solmgr.NumOfSolution();
  newapplplan2.solutionvar = solmgr.SolutionVariable();
}  //else
  //*
AppOut.appout("GenApplPlan - NEXT PLAN !!!!!!!!!!");
}  //for each relevant plan (i) ******
//To select AN applicable plan with m.g.u. (One with the largest number of unifiers)
if (this.appplans.size() > 0) {
  intselapplplan = this.appplans.peekNode(1);
  for (int q=2; q <= this.appplans.size(); q++) {
    if (this.appplans.peekNode(q).bindstack.size() > this.selpapplplan.
      bindstack.size()) {
      this.selpapplplan = this.appplans.peekNode(q);
    }
  }  //for
//Instantiated applicable plan (applying applicable unifier to body's formulae)
for (int r=1; r <= this.selpapplplan.bodylist.size(); r++) {
  Substitution.substitution(this.selapplplan.bindstack, this.
    selpapplplan.bodylist.peekNode(r).bdyTerm);
}  //for
//If EXTERNAL event
if (this.event.evtIntention == null) {
  //Brand' new intention with status & suspend set to false (non active)
  intplanstack newintent = new intplanstack();
  //Duplicate/clone selected applicable plan & insert into the NEW intention
  newintent.push((relplannode) selapplplan.clone(true));
  relplannode intapplplan1 = (relplannode) selapplplan.clone(true);
  intapplplan1.consnetwork = selapplplan.consnetwork;
  intapplplan1.solutionset = selapplplan.solutionset;
  intapplplan1.solutionnum = selapplplan.solutionnum;
  newintent.push(intapplplan1);
  //Insert NEW intention into current intention set & notify intention thread (if on wait)
  //***AppletIntentManager.addintent(newintent);
((AppletAgentManager)this.applagent).intentmanager.addintent(newintent);
}
else if INTERNAL event (this.event.evtIntention != null)
else {
    relplannode intapplplan2 = (relplannode) selapplplan.clone(true);
    intapplplan2.consnetwork = selapplplan.consnetwork;
    intapplplan2.solutionset = selapplplan.solutionset;
    intapplplan2.solutionnum = selapplplan.solutionnum;
    this.event.evtIntention.push(intapplplan2);
    this.event.evtIntention.status = true; //Set event's intention to active
    this.event.evtIntention.suspend = false; //Unsuspend intention
    AppletIntentManager.suspendactive = true;
    AppletIntentManager.eventintent = this.event.evtIntention;
    if (AppletIntentManager.intwait) {
        AppletIntentManager.intwait = false;
        AppletIntentThread.intentdone = true;
        //((AppletAgentManager)this.applagent).intentmanager.notice();
    }
} //else

//Reset eventdone to true to enable next event to be picked up after current trigger event
//has been posted as intention.
AppletEventThread.eventdone = true;

AppOut.appout("GenApplPlan - AppletEventManager.eventset.size() (GenApplPlan) : " +
AppletEventManager.eventset.size());
AppletEventThread.eventdone);
} //run
} //AppletGenApplPlan thread
import java.awt.*;
import java.awt.TextComponent.*;

public class AppletIntExecution extends Thread {
    protected int planstack activeint; //Current active intention
    protected TextArea execoutput;
    public ASAgent intentagent; //Name of agent that initiates the thread

    public AppletIntExecution(int planstack ai, TextArea output, ASAgent it) {
        activeint = ai; //intention receive from intention thread
        execoutput = output;
        intentagent = it;
    } //Constructor

    public void run() {
        //Enumerate through the intended means in the plan stack of intention
        boolean suspendint = false; //Suspend intention
        boolean abortint = false; //Abort intention
        StringBuffer outbuffer = new StringBuffer();

        for (int i=1; i <= this.activeint.size(); i++) {
            boolean proceed = true;
            relplannode execuplan = this.activeint.peekNode(i); //Executing plan

            //Enumerate through formula in the body of intended means
            while (execuplan.bodylist.size() > 0) {
                bodynode execuformula = null;
                if (execuplan.bodylist.head.bdyldentifier == '!') {
                    execuformula = execuplan.bodylist.head; //Return head
                    //Achievement goal formula is not explicitly removed, retain for
                    //subsequent derivation of relevant unifier(s) with the new plan
                    //generated by the achievement goal event.
                } else {
                    //Executing formula explicitly remove from the body of the
                    //executing plan
                    //for processing
                    execuformula = execuplan.bodylist.removeFront(); //Return head
                } //else

            } //while

        } //for

        if (execuformula.bdyOper == '\0' && execuformula.bdyldentifier == '\0') {
            String termstr = "";
            String[] termargs = new String[execuformula.bdyTerm.length];
            for (int t=0; t < execuformula.bdyTerm.length; t++) {
                if (t == execuformula.bdyTerm.length - 1) {
                    termstr = termstr + execuformula.bdyTerm[t][0];
                    termargs[t] = execuformula.bdyTerm[t][0];
                } else {
                    termstr = termstr + execuformula.bdyTerm[t][0] + ", ";
                    termargs[t] = execuformula.bdyTerm[t][0];
                }
            } //for

            if (execuformula.bdyPredSym.equals("move")) {
                AppletActionRoutine.runaction(execuformula.bdyPredSym, termargs);
                String actionstr = "ACTION : " + execuformula.bdyPredSym + "("
            } //if
        } //for
termstr+"\n";
actionstr = actionstr + \n;

if (actionstr != null || actionstr != \"") {
  outbuffer.append(actionstr);
}

String fnloutput = outbuffer.toString();
//this.execoutput.setText(fnloutput);
this.execoutput.appendText(fnloutput+\n\n);
}  //if
else if (execuformula.bdyPredSym.equals("display") ) {
  ActionRoutine.display(execuplan.consnetwork, execuplan.
solutionset, execuplan.solutionnum);
  String actionstr = "ACTION : display "+execuformula.bdyPredSym+"\n\n";
  System.out.println(actionstr);
}  //else if
else {  //Generic action routine
  ActionRoutine.runaction(execuplan.consnetwork, execuformula.
bdyPredSym, termargs, execuplan.
solutionvar, this.intentagent, execuplan.
solutionnum);
  String actionstr = "ACTION : "+execuformula.bdyPredSym+"\n\n";
  System.out.println(actionstr);
  //outbuffer.append(this.execoutput.getText()+\n\n); to be
  actionstr = actionstr + \n;
  outbuffer.append(actionstr);
  String fnloutput = outbuffer.toString();
  //this.execoutput.setText(fnloutput);
  this.execoutput.appendText(fnloutput+\n\n);
}  //else
}  //****** if action

****** Achievement goal (!) - an internal event is created and
append to event queue.
//Default OPER is + if not specified. Otherwise event is created
according to
//OPER provided in bdyOper
if (execuformula.bdyIdentifier == '\!') {
  if (execuformula.bdyOper != '\0') {
    //Post to event queue
    eventnode newevent = new eventnode();
    this.activeint.status = false;
    this.activeint.suspend = true;  //Suspend intention pending
    achievement of goal
    newevent.evtOper = execuformula.bdyOper;
    newevent.evtIdentifier = '\!';
    newevent.evtPredSym = execuformula.bdyPredSym;
    newevent.evtNoofTerm = execuformula.bdyNoofTerm;
    newevent.evtTerm = new String[newevent.evtNoofTerm][2];
    newevent.evtTerm = execuformula.bdyTerm;
    newevent.evtIntention = this.activeint;

    //AppOut.appout("IntExecution :  Insert into event queue");
    //Insert into event queue
    ((AppletAgentManager)this.intentagent).eventmanager.addevent(
newevent);
  }
  //if
else {
    //Post to event queue
    eventnode newevent = new eventnode();
    this.activeint.status = false;
    this.activeint.suspend = true;  //Suspend intention pending
    //Suspension of goal
    newevent.evtOper = '+';
    newevent.evtIdentifier = '!';
    newevent.evtPredSym = execuformula.bdyPredSym;
    newevent.evtNoofTerm = execuformula.bdyNoofTerm;
    newevent.evtTerm = new String[newevent.evtNoofTerm][2];
    newevent.evtTerm = execuformula.bdyTerm;
    newevent.evtIntention = this.activeint;
    //Insert into event queue
    ((AppletAgentManager)this.intentagent).eventmanager.addevent(newevent);

    //Intention is insert back (append) to the intention set with
    //Suspend set to true
    //Pending locating an additional new plan to achieve the above
    //trigger event goal
    AppletIntentManager.intentset.addEnd(this.activeint);
    suspendint = true;  //Suspend intention
    proceed = false;  //Discontinue
}  //else
}  //*****★ j_f achievement goal

//****** Query- goal (?)
if (execuformula.bdyIdentifier == '?') {
    bindingstack tmpbindstk = null;  //Tmp binding stack
    literal tmpformula = new literal();  //Tmp formula
    tmpformula.predSymbol = execuformula.bdyPredSym;
    tmpformula.noofTerm = execuformula.bdyNoofTerm;
    tmpformula.term = execuformula.bdyTerm;

    //IF NOT ground formula
    if (! UnifyBaseBel.groundformula(tmpformula)) {  
        tmpbindstk = UnifyBaseBel.unifybasebel(tmpformula);
        if (tmpbindstk != null) {
            //Apply m.g.u. to remaining formulae in executing plan's body
            for (int j=1; j <= execuplan.bodylist.size(); j++) {
                Substitution.substitution(tmpbindstk, execuplan.bodylist.
                peekNode(j).bdyTerm);
            }  //for remaining formulae
        }  //if
    }  //else
    else {  //NOT logical consequence/query goal CANNOT be unified with
    //current base beliefs
    //Abort executing plan & executing/selected intention
    String termstr = "";
    for (int k=0; k < execuformula.bdyTerm.length; k++) {
        if (k == execuformula.bdyTerm.length - 1) {
            termstr = termstr + execuformula.bdyTerm[k][0];
        }  //else
        else {
            termstr = termstr + execuformula.bdyTerm[k][0] + ", ";
        }  //for
    }  //else
    abortint = true;  //Abort & discard intention
    proceed = false;  //Discontinue
}  //if
}  //if not ground formula
//To test if ground and logical consequence
else if (! UnifyBaseBel.logconseq(tmpformula)) {
//Abort executing plan & executing/selected intention as the query state/belief 
//is not tested positive/true in the current beliefs, i.e. the required state 
//does not exist.

String termstr = "";
for (int k=0; k < execuformula.bdyTerm.length; k++) {
    if (k == execuformula.bdyTerm.length - 1) {
        termstr = termstr + execuformula.bdyTerm[k][0];
    } else {
        termstr = termstr + execuformula.bdyTerm[k][0] + ", ";
    }
}  //for

AppOut.appout("Intention ABORT and DISCARDED ! ");
AppOut.appout("Required state : ?"+execuformula.bdyPredSym+"("+termstr+")");

abortint = true;  
//Abort & discard intention
proceed = false;  
//Discontinue
else ground formula & NOT log.conseq.

//By default, if it is ground and logical consequence: processing roll on to
//next formula in the body of executing plan. The query state exist in current
//beliefs.

}  //***** if query goal

if (! proceed) {
    break;  //Break while (execuplan.bodylist.size() > 0) and intention was suspended
}  //if
else {
//No more formula in the current executing plan
if (execuplan.bodylist.size() == 0) {

//AppOut.appout("this.activeint.size() : "+this.activeint.size());

//more plans in current plan stack of active intention
//Propagate additional binding constraints (m.g.u.) into next plan in the intention
if ((this.activeint.size() - 1) > 0) {

//AppOut.appout("this.activeint.size() - 1 : "+(this.activeint.size()-1));
//if ((this.activeint.size() - 1) < 2
//&& execuplan.nextPlan.bodylist.size() > 1) {

invocation tmpcurrplaninvoc = execuplan.invEvent;
relplannode nextplan = this.activeint.peekNode(i+1);
bodynode tmpNextplantrigger =
nextplan.bodylist.removeFront();  //Return head

literal tmplinv = new literal();
literal tmplevt = new literal();
tmplinv.predSymbol = tmpcurrplaninvoc.invPredSym;
tmplinv.noofTerm = tmpcurrplaninvoc.invNoOfTerm;
tmplinv.term = tmpcurrplaninvoc.invTerm;
tmplevt.predSymbol = tmpNextplantrigger.bdyPredSym;
tmplevt.noofTerm = tmpNextplantrigger.bdyNoOfTerm;
tmplevt.term = tmpNextplantrigger.bdyTerm;

//Unify invocation & trigger
bindingstack tmpbindstk = Unification.unification(tmplinv, tmplevt);

//AppOut.appout("nextplan.bodylist.size() : "+nextplan.bodylist.size());
//Applying unifiers (tmpbindstk) to body of next plan
if (nextplan.bodylist.size() > 0) {
for (int 1=1; 1 <= nextplan.bodylist.size(); 1++) {
    Substitution.substitution(tmpbindstk, nextplan.bodylist.
        peekNode(1).bdyTerm);
}  // for
}  //if
}  //if there are still more plans in current plan stack of
active intention
}  //if last formula in current executing formula
}  //if ! proceed
}  //while (execuplan.bodylist.size() > 0)

if (suspendint) {
    break; //Break for loop/intention was suspended
}  //if

if (abortint) {
    break; //Break for loop/intention will be
    aborted/discard
}  //if

}  //for every plan in the intention stack

//Reset intentdone to true to enable next intention to be picked up
after
//execution for the current intention has been completed or aborted.
AppletIntentThread.intentdone = true;

}  //run

}  //AppletIntExecution thread
import JCL.*;
import java.util.Vector;

class eventnode {
    public int eventID;
    public char evtOper;
    public char evtldentifier;
    public String evtPredSym;
    public byte evtNoofTerm;
    public String[][] evtTerm;
    public intplanstack evtlntention;
    public LiteralNetwork literalcons;
    public Vector solnset; //Set of solutions(including attributes)
    public int solnnum; //Number of solution
    public String[] solnvar; //Array of matching variables as in solution set (solnset)
    public eventnode nextEvent;

    public eventnode() {
        eventID = 0;
        evtOper = '\0';
        evtldentifier = '\0';
        evtPredSym = " ";
        evtNoofTerm = 0;
        //evtTerm = new String[evtNoofTerm][2];
        evtlntention = null;
        literalcons = null;
        solnset = null;
        solnnum = 0;
        solnvar = null;
        nextEvent = null;
    }

    public eventnode(char op, char id, String pred, byte nt, eventnode ne, intplanstack ei) {
        eventID = 0;
        evtOper = op;
        evtldentifier = id;
        evtPredSym = pred;
        evtNoofTerm = nt;
        evtlntention = ei;
        literalcons = null;
        solnset = null;
        solnnum = 0;
        solnvar = null;
        nextEvent = ne;
    }
}
/* Return true if list is empty */
public boolean isEmpty() {
    return head == null;
}

/* Insert first node */
public void insert(eventnode enode) {
    eventnode tmphead = head;
    head = new eventnode();
    head = enode;
    head.nextEvent = tmphead;
    nodenum++;
}

public synchronized void addEnd(eventnode aenode) {
    if (this.isEmpty())
        insert(aenode);
    else {
        eventnode tmp = head;
        while ( tmp.nextEvent != null )
            tmp = tmp.nextEvent;
        eventnode newnode = new eventnode();
        tmp.nextEvent = aenode;
        tmp.nextEvent.nextEvent = null;
        nodenum++;
    } /* else */
} /* addEnd */

public synchronized eventnode removeFront() {
    if (this.isEmpty())
        return null;
    eventnode tmphead = head;
    head = tmphead.nextEvent;
    nodenum--;
    return tmphead;
}

public synchronized eventnode removeEnd() {
    if (this.isEmpty())
        return null;
    if (head.nextEvent == null)
        return removeFront();
    eventnode tmp = head;
    while (tmp.nextEvent.nextEvent != null)
        tmp = tmp.nextEvent;
    eventnode tmpevent = tmp.nextEvent;
    tmp.nextEvent = tmp.nextEvent.nextEvent; /* Set tmp.nextEvent = null */
    nodenum--;
    return tmpevent;
}
public eventnode peekNode(int nnum) {
    if (nnum == 1) {
        return head;
    }
    else {
        eventnode tmpnode = head;
        // for (int i = 0; i < nnum && tmpnode != null; i++)
        for (int i = 1; i < nnum && tmpnode != null; i++)
            tmpnode = tmpnode.nextEvent;
        return tmpnode;
    }
}

public void print(int nnum) {
    eventnode tmpnode = head;
    // for (int i = 0; i < nnum && tmpnode != null; i++)
    for (int i = 1; i < nnum && tmpnode != null; i++)
        tmpnode = tmpnode.nextEvent;
    String tmpterm = "Term";
    System.out.println(tmpnode.evtPredSym+" ("+tmpterm+")");
}
/* Class eventlist */
//***********************************************************
Program Name : relplannode.java
Program Function : Plan node with relevant & application unifiers
(stack) for the plan.
(Applicable plan share the same data structure)
Last Update : 16 Jan 1999
06 Oct 1999-Include constraint network to be passed
down to subgoal.
Code By : Boon
***********************************************************/
import java.util.Vector;
import JCL.*;

class relplannode extends Object implements Cloneable{
    public int planID;
    public invocation invEvent;
    public contextnodelist contextlist;   //List of context nodes (beliefs List)
    public bodynodelist bodylist;   //List of body's formula (action/subgoal)
    public relplannode nextPlan;   //Next plan in the plan library for an agent
    public bindingstack bindstack;   //Relevant and applicable unifiers
down to sub-goal
    public LiteralNetwork consnetwork;   //Constraint network to be passed
down to sub-goal
    public Vector solutionset;  //Set of solutions(including solution attributes)
    public int solutionnum;  //Number of solution
    public String[] solutionvar;   //Array of variables in constraint network

    public relplannode() {
        planID = 0;
        invEvent = null;
        contextlist = null;
        bodylist = null;
        nextPlan = null;
        bindstack = null;
        consnetwork = null;
        solutionset = null;
        solutionnum = 0;
        solutionvar = null;
    }

    //single-node : if singlenode (true) -> clone as an individual & independent node
    public Object clone(boolean singlenode) {
        relplannode cln = new relplannode();

        if (this.invEvent != null) {
            cln.invEvent = (invocation) this.invEvent.clone();
        }
        if (this.contextlist != null) {
            cln.contextlist = (contextnodelist) this.contextlist.clone();
        }
        if (this.bodylist != null) {
            cln.bodylist = (bodynodelist) this.bodylist.clone();
        }
        if (! singlenode && this.nextPlan != null) {
            cln.nextPlan = (relplannode) this.nextPlan.clone(false);
        }
        if (this.bindstack != null) {
            cln.bindstack = (bindingstack) this.bindstack.clone();
        }
        return cln;
    }
    //clone
}
//relplannode
import relplannode;

public class relplanlist {
    protected relplannode head;
    protected int nodenum;

    public relplanlist() {
        head = null;
        nodenum = 0;
    }

    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    public void insert(relplannode pnode) {
        relplannode tmphead = head;
        head = pnode;
        head.nextPlan = tmphead;
        nodenum++;
    }

    public void addEnd(relplannode aepnode) {
        if (isEmpty())
            insert(aepnode);
        else {
            relplannode tmp = head;
            while (tmp.nextPlan != null)
                tmp = tmp.nextPlan;
            tmp.nextPlan = aepnode;
            tmp.nextPlan.nextPlan = null;
            nodenum++;
        }
    }

    public relplannode removeFront() {
        if (isEmpty())
            return null;
        relplannode tmp = head;
        head = tmp.nextPlan;
        nodenum--;
        return tmp;
    }

    public relplannode removeEnd() {
        if (isEmpty())
            return null;
        if (head.nextPlan == null)
            return removeFront();
        relplannode tmp = head;
        while (tmp.nextPlan.nextPlan != null)
            tmp = tmp.nextPlan;
        relplannode tmpplan = tmp.nextPlan;
        tmp.nextPlan = tmp.nextPlan.nextPlan; /* Set tmp.nextPlan = null */
        nodenum--;
        return tmpplan;
    }
}
relplanlist.java

4/02/1999

public relplannode peekNode(int nnum) {
    if (nnum == 1) {
        return head;
    }
    else {
        relplannode tmpnode = head;
        // for (int i = 0; i < nnum && tmpnode != null; i++)
        for (int i = 1; i < nnum && tmpnode != null; i++)
            tmpnode = tmpnode.nextPlan;
        return tmpnode;
    }
}

public void print(int nnum) {
    relplannode tmpnode = head;
    // for (int i = 0; i < nnum && tmpnode != null; i++)
    for (int i = 1; i < nnum && tmpnode != null; i++)
        tmpnode = tmpnode.nextPlan;
    String tmpterm = "Term";
    System.out.println(tmpnode.invEvent.invPredSym+"{"+tmpterm+"}");
}
/* Class relplanlist */
import relplannode;

public class intplanstack {
    protected boolean status; //Active status indicator
    protected boolean suspend;
    protected relplannode head;
    protected int nodenum;
    protected intplanstack nextStack; //Next intention (plan stack) down the list

    public intplanstack() {
        status = false;
        suspend = false;
        head = null;
        nodenum = 0;
        nextStack = null;
    }

    /* Return true if stack is empty */
    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    /* Insert first node */
    public void push(relplannode hdnode) {
        relplannode tmphead = head;
        // head = new relplannode();
        head = hdnode;
        head.nextPlan = tmphead;
        nodenum++;
    }

    public relplannode pop() {
        if (isEmpty())
            return null;
        relplannode tmp = head;
        head = tmp.nextPlan;
        nodenum--;
        return tmp;
    }

    public relplannode peekNode(int nnum) {
        if (nnum == 1) {
            return head;
        }
        else {
            relplannode tmpnode = head;
            //for (int i=0; i < nnum && tmpnode != null; i++)
            for (int i=1; i < nnum && tmpnode != null; i++)
                tmpnode = tmpnode.nextPlan;
            return tmpnode;
        }
    }

    public void print(int nnum) {
        relplannode tmpnode = head;
        //for (int i =0; i < nnum && tmpnode != null; i++)
        for (int i = 1; i < nnum && tmpnode != null; i++)
            tmpnode = tmpnode.nextPlan;
        System.out.println(tmpnode.invEvent.invPredSym+"(Inv Pred Sym)");
    }
}
```java
import invocation;
import contextnode;
import bodynode;

class plannode {
    public int planID;
    public invocation invEvent;
    public contextnodelist contextlist;  //List of context nodes (beliefs
        List)
    public bodynodelist bodylist;  //List of body's formula
    (action/subgoal)
    public plannode nextPlan;  //Next plan in the plan library for an
        agent

    public plannode() {
        planID = 0;
        invEvent = null;
        contextlist = null;
        bodylist = null;
        nextPlan = null;
    }

    public plannode(invocation ie, contextnodelist ct, bodynodelist by,
            plannode np) {
        planID = 0;
        invEvent = ie;
        contextlist = ct;
        bodylist = by;
        nextPlan = np;
    }
}  //plannode
```
import plannode;

public class plannodelist{
    protected plannode head;
    protected int nodenum;

    public plannodelist() {
        head = null;
        nodenum = 0;
    }

    /* Return true if list is empty */
    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    /* Insert first node */
    public void insert(plannode pnode) {
        plannode tmphead = head;
        // head = new plannode();
        head = pnode;
        head.nextPlan = tmphead;
        nodenum++;
    }

    public void addEnd(plannode aepnode) {
        if (isEmpty())
            insert(aepnode);
        else {
            plannode tmp = head;
            while (tmp.nextPlan != null)
                tmp = tmp.nextPlan;
            // plannode newnode = new plannode();
            tmp.nextPlan = aepnode;
            tmp.nextPlan.nextPlan = null;
            nodenum++;
        }  /* else */
    }  /* addEnd */

    public plannode removeFront() {
        if (isEmpty())
            return null;
        plannode tmp = head;
        head = tmp.nextPlan;
        nodenum--;
        return tmp;
    }

    public plannode removeEnd() {
        if (isEmpty())
            return null;
        if (head.nextPlan == null)
            return removeFront();
        plannode tmp = head;
        while (tmp.nextPlan.nextPlan != null)
            tmp = tmp.nextPlan;
        plannode tmpplan = tmp.nextPlan;
        tmp.nextPlan = tmp.nextPlan.nextPlan; /* Set tmp.nextPlan = null */
        nodenum--;
        return tmpplan;
    }

    public plannode peekNode(int nnum) {
        ...
if (nnum == 1) {
    return head;
} else {
    plannode tmpnode = head;
    // for (int i = 0; i < nnum && tmpnode != null; i++)
    for (int i = 1; i < nnum && tmpnode != null; i++)
        tmpnode = tmpnode.nextPlan;
    return tmpnode;
}

public void print(int nnum) {
    plannode tmpnode = head;
    // for (int i = 0; i < nnum && tmpnode != null; i++)
    for (int i = 1; i < nnum && tmpnode != null; i++)
        tmpnode = tmpnode.nextPlan;
    String tmpterm = "Term";
    System.out.println(tmpnode.invEvent.invPredSym+"("+tmpterm+");
}
} /* Class plannodelist */
public class intentionlist {
    protected intplanstack head;
    protected int nodenum;

    public intentionlist() {
        head = null;
        nodenum = 0;
    }

    /* Return true if list is empty */
    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    /* Insert first node */
    public void insert(intplanstack inode) {
        intplanstack tmphead = head;
        // head = new intplanstack();
        head = inode;
        head.nextStack = tmphead;
        nodenum++;
    }

    public synchronized void addEnd(intplanstack aeinode) {
        if (isEmpty())
            insert(aeinode);
        else {
            intplanstack tmp = head;
            while (tmp.nextStack != null)
                tmp = tmp.nextStack;
            intplanstack newnode = new intplanstack() ;
            tmp.nextStack = aeinode;
            tmp.nextStack.nextStack = null;
            nodenum--;
        }
    }

    public synchronized intplanstack removeFront() {
        if (isEmpty())
            return null;
        intplanstack tmp = head;
        head = tmp.nextStack;
        nodenum--;    
        return tmp;
    }

    public synchronized intplanstack removeEnd() {
        if (isEmpty())
            return null;
        if (head.nextStack == null)
            return removeFront();
        intplanstack tmp = head;
        while (tmp.nextStack.nextStack != null)
            tmp = tmp.nextStack;
        intplanstack tmpstack = tmp.nextStack;
        tmp.nextStack = tmp.nextStack.nextStack; /* Set tmp.nextStack = null */
        nodenum--;
    }
```java
public int planstack peekNode(int nnum) {
    if (nnum == 1) {
        return head;
    }
    else {
        int planstack tmpnode = head;
        for (int i = 0; i < nnum && tmpnode != null; i++)
            for (int i = 1; i < nnum && tmpnode != null; i++)
                tmpnode = tmpnode.nextStack;
        return tmpnode;
    }
}

public void removeNode(int planstack rnode) {
    int planstack beforenode = null;
    int planstack currentnode = head;
    for (int i=1; i <= size(); i++) {
        if (i == 1) {
            if (rnode == currentnode) {
                removeFront();
                break;
            }  //if
            else {
                beforenode = currentnode;
                currentnode = currentnode.nextStack;
            }  //else
        }  //if i==1
        else {
            if (rnode == currentnode) {
                beforenode.nextStack = currentnode.nextStack;
                nodenum--;
                break;
            }  //if
            else {
                beforenode = currentnode;
                currentnode = currentnode.nextStack;
            }  //else
        }  //else
    }  //for i
}

public void print(int nnum) {
    int planstack tmptmnode = head;
    // for (int i = 0; i < nnum && tmptmnode != null; i++)
    for (int i = 1; i < nnum && tmptmnode != null; i++)
        tmptmnode = tmptmnode.nextStack;
    String tmpterm = "Term";
    System.out.println(tmptmnode.peekNode(3).invEvent.invPredSym+"("+tmpterm+"\)");
} /* Class intentionlist */
```
import bindingnode;

public class bindingstack extends Object implements Cloneable{
    protected bindingnode head;
    protected int nodenum;

    public bindingstack() {
        head = null;
        nodenum = 0;
    }

    public Object clone() {
        bindingstack cln = new bindingstack();
        cln.head = (bindingnode) this.head.clone();
        cln.nodenum = this.nodenum;
        return cln;
    }  //clone

    /* Return true if stack is empty */
    public boolean isEmpty() {
        return head == null;
    }

    public int size() {  
        return nodenum;
    }

    /* Insert first node */
    public void push(bindingnode hdnode) {
        bindingnode tmphead = head;
        // head = new bindingnode();
        head = hdnode;
        head.nextBind = tmphead;
        nodenum++;
    }  

    public bindingnode pop() {
        if (isEmpty())
        return null;
        bindingnode tmp = head;
        head = tmp.nextBind;
        nodenum--;
        return tmp;
    }

    public bindingnode peekNode(int nnum) {
        if (nnum == 1) {
            return head;
        } else {
            bindingnode tmppnode = head;
            for (int i = 0; i < nnum & tmppnode != null; i++)
                for (int i = 1; i < nnum & tmppnode != null; i++)
                    tmppnode = tmppnode.nextBind;
            return tmppnode;
        }
    }

    public void print(int nnum) {
        bindingnode tmppnode = head;
        for (int i = 0; i < nnum & tmppnode != null; i++)
            for (int i = 1; i < nnum & tmppnode != null; i++)
                tmppnode = tmppnode.nextBind;
        System.out.print(tmppnode.bindTerm1+"(Bind Term 1) ");
        System.out.print(tmppnode.bindTerm2+"(Bind Term 2) ");
    }
}  /* Class bindingstack */
public class bindingnode extends Object implements Cloneable{
    public String bindTerm1, bindTerm2;
    public bindingnode nextBind;

    public bindingnode() {
        bindTerm1 = null;
        bindTerm2 = null;
        nextBind = null;
    }

    public bindingnode(String p1, String p2, bindingnode nbind) {
        bindTerm1 = p1;
        bindTerm2 = p2;
        nextBind = nbind;
    }

    public Object clone() {
        bindingnode cln = new bindingnode();

        cln.bindTerm1 = this.bindTerm1;
        cln.bindTerm2 = this.bindTerm2;
        if (this.nextBind != null) {
            cln.nextBind = (bindingnode) this.nextBind.clone();
        }

        return cln;
    }

} // bindingnode
class invocation extends Object implements Cloneable{
    public int planID;
    public char invOper;
    public char invIdentifier;
    public String invPredSym;
    public byte invNccfTerr;
    public String[] invTerm;
    //public contextnode context;

    public invocation() {
        planID = 0;
        invOper = '\0';
        invIdentifier = '\0';
        invPredSym = '\0';
        invNccfTerr = 0;
        //invTerm = new String[invNccfTerm][2];
        //context = null;
    }

    public invocation(char op, char id, String pred, byte nt) {
        planID = 0;
        invOper = op;
        invIdentifier = id;
        invPredSym = pred;
        invNccfTerr = nt;
        invTerm = new String[nt][2];
        //context = null;
    }

    public Object clone() {
        invocation clin = new invocation();
        clin.invOper = this.invOper;
        clin.invIdentifier = this.invIdentifier;
        clin.invPredSym = this.invPredSym;
        clin.invNccfTerr = this.invNccfTerr;
        if (this.invTerm.length > 0) {
            clin.invTerm = new String[clin.invNccfTerr][2];
            for (int i=0; i < this.invTerm.length; i++) {
                for (int j=0; j < this.invTerm[i].length; j++) {
                    clin.invTerm[i][j] = this.invTerm[i][j];
                }
            }
        }
        return clin;
    }
import bodynode;

public class bodynodelist extends Object implements Cloneable {
    protected bodynode head;
    protected int nodenum;

    public bodynodelist() {
        head = null;
        nodenum = 0;
    }

    public Object clone() {
        bodynodelist cln = new bodynodelist();
        cln.head = (bodynode) this.head.clone();
        cln.nodenum = this.nodenum;
        return cln;
    }

    /* Return true if list is empty */
    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    /* Insert first node */
    public void insert(bodynode bn) {
        bodynode tmphead = head;
        head = new bodynode();
        head = bn;
        head.nextBody = tmphead;
        nodenum++;
    }

    public void addEnd(bodynode an) {
        if (isEmpty())
            insert(an);
        else {
            bodynode tmp = head;
            while (tmp.nextBody != null)
                tmp = tmp.nextBody;
            contextnode newnode = new bodynode();
            tmp.nextBody = newnode;
            tmp.nextBody.nextBody = null;
            nodenum++;
        }
    }

    public bodynode removeFront() {
        if (isEmpty())
            return null;
        bodynode tmp = head;
        head = tmp.nextBody;
        nodenum--;
        return tmp;
    }

    public bodynode removeEnd() {
        if (isEmpty())
            return null;
        if (head.nextBody == null)
            return removeFront();
        bodynode tmp = head;
    }
}
while (tmp.nextBody.nextBody != null)
    tmp = tmp.nextBody;
bodynode tmpbody = tmp.nextBody;
tmp.nextBody = tmp.nextBody.nextBody;  // Set tmp.nextBody = null
nodeNum--;  // Decrease node count
return tmpbody;

public bodynode peekNode(int nnum) {
    if (nnum == 1) {
        return head;
    } else {
        bodynode tmpnode = head;
        for (int i = 0; i < nnum && tmpnode != null; i++)
            tmpnode = tmpnode.nextBody;
        return tmpnode;
    }
};

public void print(int nnum) {
    bodynode tmpnode = head;
    for (int i = 0; i < nnum && tmpnode != null; i++)
        tmpnode = tmpnode.nextBody;
String tmpterm = "Term";
System.out.println(tmpnode.bdyPredSym-"("+tmpterm+")") ;
};  // Class bodynodeList  *
class bodynode extends Object implements Cloneable {
    public int planID;
    public int bodyID;
    public char bdyOper;
    public char bdyIdentifier;
    public String bdyPredSym;
    public byte bdyNooffTerm;
    public String[][] bdyTerm;
    public bodynode nextBody;

    public bodynode() {
        planID = 0;
        bodyID = 0;
        bdyOper = '\0';
        bdyIdentifier = '\0';
        bdyPredSym = "";
        bdyNooffTerm = 0;
        // bdyTerm = new String[bdyNooffTerm][2];
        nextBody = null;
    }

    public bodynode(char bop, char bid, String bpred, byte bnt, bodynode nbn) {
        planID = 0;
        bodyID = 0;
        bdyOper = bop;
        bdyIdentifier = bid;
        bdyPredSym = bpred;
        bdyNooffTerm = bnt;
        bdyTerm = new String[bdyNooffTerm][2];
        nextBody = nbn;
    }

    public Object clone() {
        bodynode cln = new bodynode();
        cln.bdyOper = this.bdyOper;
        cln.bdyIdentifier = this.bdyIdentifier;
        cln.bdyPredSym = this.bdyPredSym;
        cln.bdyNooffTerm = this.bdyNooffTerm;
        if (this.bdyNooffTerm > 0) {
            cln.bdyTerm = new String[this.bdyNooffTerm][2];
            for (int i = 0; i < this.bdyTerm.length; i++) {
                for (int j = 0; j < this.bdyTerm[i].length; j++) {
                    cln.bdyTerm[i][j] = this.bdyTerm[i][j];
                }
            }
        } else {
            throw new RuntimeException();
        }
        //if
        if (this.nextBody != null) {
            cln.nextBody = (bodynode) this.nextBody.clone;
        }
        return cln;
    } //clone
} //bodynode
contextnode.java

/******************************************************************************
Program Name : contextnode.java
Program Function : Context (in the head) of plan
if nextContext = null { End of context list for the plan }
Last Update : 01 Jan 1999
Code By : Boon
******************************************************************************/
class contextnode extends Object implements Cloneable{
    public int planID;
    public int contextID;
    public String ctxPredSym;
    public byte ctxNoofTerm;
    public String[][] ctxTerm;
    public contextnode nextContext;

    public contextnode() {
        planID = 0;
        contextID = 0;
        ctxPredSym = "";
        ctxNoofTerm = 0;
        // ctxTerm = new String[ctxNoofTerm][2];
        nextContext = null;
    }

    public contextnode(String cps, byte cnt, contextnode nc) {
        planID = 0;
        contextID = 0;
        ctxPredSym = cps;
        ctxNoofTerm = cnt;
        ctxTerm = new String[cnt][2];
        nextContext = nc;
    }

    public Object clone() {
        contextnode cln = new contextnode();

        cln.ctxPredSym = this.ctxPredSym;
        cln.ctxNoofTerm = this.ctxNoofTerm;
        if (this.ctxNoofTerm > 0) {
            cln.ctxTerm = new String[cln.ctxNoofTerm][2];
            for (int i=0; i < this.ctxTerm.length; i++) {
                for (int j=0; j < this.ctxTerm[i].length; j++) {
                    cln.ctxTerm[i][j] = this.ctxTerm[i][j];
                }
            }
        }
        if (this.nextContext != null) {
            cln.nextContext = (contextnode) this.nextContext.clone();
        }

        return cln;
    }
} //contextnode
import contextnode;

class contextnodelist extends Object implements Cloneable{
    protected contextnode head;
    protected int nodenum;

    public contextnodelist() {
        head = null;
        nodenum = 0;
    }

    public Object clone() {
        contextnodelist cln = new contextnodelist();
        cln.head = (contextnode) this.head.clone();
        cln.nodenum = this.nodenum;
        return cln;
    }

    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    public void insert(contextnode cnode) {
        // head = new contextnode();
        head = cnode;
        head.nextContext = tmphead;
        nodenum++;
    }

    public void addEnd(contextnode aenode) {
        if (this.isEmpty())
            insert(aenode);
        else {
            contextnode tmp = head;
            while (tmp.nextContext != null)
                tmp = tmp.nextContext;
            // contextnode newnode = new contextnode();
            tmp.nextContext = aenode;
            tmp.nextContext.nextContext = null;
            nodenum++;
        } /* else */
    } /* addEnd */

    public contextnode removeFront() {
        if (this.isEmpty())
            return null;
        contextnode tmp = head;
        head = tmp.nextContext;
        nodenum--;
        return tmp;
    }

    public contextnode removeEnd() {
        if (this.isEmpty())
            return null;
        if (head.nextContext == null)
            return removeFront();
        contextnode tmp = head;
while (tmp.nextContext.nextContext != null)
    tmp = tmp.nextContext;
contextnode tmpcontext = tmp.nextContext;
tmp.nextContext = tmp.nextContext.nextContext; /* Set tmp.nextContext = null */
nodenum--;
return tmpcontext;
}

public contextnode peekNode(int nnum) {
    if (nnum == 1) {
        return head;
    }
    else {
        contextnode tmpnode = head;
        // for (int i = 0; i < nnum && tmpnode != null; i++)
        for (int i = 1; i < nnum && tmpnode != null; i++)
            tmpnode = tmpnode.nextContext;
        return tmpnode;
    }
}

public void print(int nnum) {
    contextnode tmpnode = head;
    // for (int i = 0; i < nnum && tmpnode != null; i++)
    for (int i = 1; i < nnum && tmpnode != null; i++)
        tmpnode = tmpnode.nextContext;
    String tmpterm = "Term";
    System.out.println(tmpnode.ctxPredSym+"("+tmpterm+")");
} /* Class contextnodelist */
C.3 Java source codes for for base belief of ConstraintAgentSpeak agent

- beliefnode.java
- beliefnodelist.java
- termpairnode.java
- termpairstack.java
- literal.java
import literal;

class beliefnode {
    protected int beliefID;
    protected literal beliefAtom;
    protected beliefnode nextBelief;

    /*** Constructor ***/
    public beliefnode() {
        beliefID = 0;
        beliefAtom = null;
        nextBelief = null;
    }

    public beliefnode(int belid, literal newbel, beliefnode newnxtbel) {
        beliefID = belid;
        beliefAtom = newbel;
        nextBelief = newnxtbel;
    }

    public beliefnode(literal newbel) {
        beliefID = 0;
        beliefAtom = newbel;
        nextBelief = null;
    }

    public beliefnode(literal newbel, beliefnode newnxtbel) {
        beliefID = 0;
        beliefAtom = newbel;
        nextBelief = newnxtbel;
    }

    //*************************************************************************
    //**** Constructor ****
    //*************************************************************************

    public void setNextBelief(beliefnode nxtbel) {
        this.nextBelief = nxtbel;
    }

    public void setBeliefID(int id) {
        this.beliefID = id;
    }

    public void setBelief(literal bel) {
        this.beliefAtom = bel;
    }

    public int getBeliefID() {
        return this.beliefID;
    }

    public beliefnode getNextBelief() {
        return this.nextBelief;
    }

    public literal getBelief() {
        return this.beliefAtom;
    }

    public String toString() {
        return this.beliefAtom;
    }
}
}  //beliefnode
import literal;
import beliefnode;

public class beliefnodelist {
    protected beliefnode head;
    protected int nodenum;

    public beliefnodelist() {
        head = null;
        nodenum = 0;
    }

    /* Return true if list is empty */
    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    /* Insert first node */
    public void insert(literal nodebel) {
        beliefnode tmphead = head;
        head = new beliefnode();
        head.beliefAtom = nodebel;
        head.nextBelief = tmphead;
        nodenum++;
    }

    public void addEnd(literal nodeend) {
        if (isEmpty())
            insert(nodeend);
        else {
            beliefnode tmp = head;
            while (tmp.nextBelief != null)
                tmp = tmp.nextBelief;
            beliefnode newnode = new beliefnode(nodeend);
            tmp.nextBelief = newnode;
            nodenum++;
        } /* else */
    } /* addEnd */

    public literal removeFront() {
        if (isEmpty())
            return null;
        beliefnode tmp = head;
        head = tmp.nextBelief;
        nodenum--;
        return tmp.beliefAtom;
    }

    public literal removeEnd() {
        if (isEmpty())
            return null;
        if (head.nextBelief == null)
            return removeFront();
        beliefnode tmp = head;
        while (tmp.nextBelief.nextBelief != null)
            tmp = tmp.nextBelief;
        literal tmpbelief = tmp.nextBelief.beliefAtom;
        tmp.nextBelief = tmp.nextBelief.nextBelief; /* Set tmp.nextBelief = null */
        nodenum--;
        return tmpbelief;
    }
public beliefnode peekNode(int nnum) {
    if (nnum == 1) {
        return head;
    } else {
        beliefnode tmpnode = head;
        //for (int i =0; i < nnum && tmpnode != null; i++)
        for (int i =1; i < nnum && tmpnode != null; i++)
            tmpnode = tmpnode.nextBelief;
        return tmpnode;
    }
}

public void print(int nnum) {
    beliefnode tmpnode = head;
    for (int i =0; i < nnum && tmpnode != null; i++)
        tmpnode = tmpnode.nextBelief;
    String tmpterm = "Term";
    System.out.println(tmpnode.beliefAtom.predSymbol + "(" + tmpterm + ")");
}
/* Class beliefnodelist */
class termpairnode {
    public String term1, term2;
    public termpairnode nextPair;

    public termpairnode() {
        term1 = null;
        term2 = null;
        nextPair = null;
    }

    public termpairnode(String t1, String t2, termpairnode np) {
        term1 = t1;
        term2 = t2;
        nextPair = np;
    }
} //termpairnode
import termpairnode;

public class termpairstack {
    protected termpairnode head;
    protected int nodenum;

    public termpairstack() {
        head = null;
        nodenum = 0;
    }

    /* Return true if stack is empty */
    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return nodenum;
    }

    /* Insert first node */
    public void push(termpairnode hdnode) {
        termpairnode tmphead = head;
        // head = new termpairnode();
        head = hdnode;
        head.nextPair = tmphead;
        nodenum++;
    }

    public termpairnode pop() {
        if (isEmpty())
            return null;
        termpairnode tmp = head;
        head = tmp.nextPair;
        nodenum--;
        return tmp;
    }

    public termpairnode peekNode(int nnum) {
        if (nnum == 1) {
            return head;
        } else {
            termpairnode tmpnode = head;
            // for (int i = 0; i < nnum & tmpnode != null; i++)
            for (int i = 1; i < nnum & tmpnode != null; i++)
                tmpnode = tmpnode.nextPair;
            return tmpnode;
        }
    }

    public void print(int nnum) {
        termpairnode tmpnode = head;
        // for (int i =0; i < nnum & tmpnode != null; i++)
        for (int i = 1; i < nnum & tmpnode != null; i++)
            tmpnode = tmpnode.nextPair;
        String tmpterml = "Term 1";
        String tmpterm2 = "Term 2";
        System.out.println(tmpterml + tmpterm2);
        //System.out.println(tmpterml + predSymbol + tmpterm2);
    }
} /* class termpairstack */
class literal {
    public String predSymbol;
    public byte noofTerm;
    public String[] [] term;

    public literal() {
        predSymbol = null;
        noofTerm = 0;
        // term = new String[noofTerm][2];
    }

    public literal(String pred, byte nterm) {
        predSymbol = pred;
        noofTerm = nterm;
        term = new String[nterm][2];
    }
} //literal
C.4 Java source codes for *CustSolutionManager.java* – a customised solution manager that handles solution set generated from constraint processing

- *CustSolutionManager.java*
package JCL;

import java.util.*;
import java.io.PrintStream;
import java.awt.*;

public class CustSolutionManager implements SolutionManagerInterface {

    Vector v;
    int number;

    Network net;
    Solver solver;
    SolutionAttributes attributes;

    SolvingInProgressWindow window;
    ActionInterface intf;

    boolean int_html = false;
    boolean final_html = false;
    boolean int_console = false;
    boolean final_console = true;

    int enters = 0;
    int leaves = 0;
    int insts = 0;
    int checks = 0;

    public CustSolutionManager() {
    }

    /**
     * To handling display of solution
     */
    public CustSolutionManager (LiteralNetwork inet, Vector solnvec, int numsol) {
        this.net = inet.buildNetwork();
        this.v = solnvec;
        this.number = numsol;
    }

    /**
     * Notify that a solving is beginning.
     */
    public void NotifyStart (Network net, Solver solver, SolutionAttributes attributes) {
        this.net = net;
        this.solver = solver;
        this.attributes = attributes;
        // Initialize the variables
v = new Vector();
number = 0;

// Put the attributes in the beginning, in the case there is no solution
v.addElement (attributes);
}

/* Display the number of solution. */
public int NumOfSolution () {
    return number;
}

/* Return vector of solution. */
public Vector SolutionVector () {
    return v;
}

/* Return vector of solution value. */
public Vector SolutionValue () {
    Vector sv = new Vector();
    Enumeration solnenum = this.v.elements();
    while (solnenum.hasMoreElements()) { //loop through all solution set/node in Vector
        int[] tmpsolindex = ((Solution) solnenum.nextElement()).values;
        String[][] solval = new String[tmpsolindex.length][2];
        for (int i=0; i<tmpsolindex.length; i++) {
            solval[i][0] = this.net.GetVariable(i).GetName();
            solval[i][1] = this.net.GetVariable(i).GetDomain().GetValueName(tmpsolindex[i]);
        } // for
        sv.addElement(solval);
    } //while
    return sv;
}

/*Return array of variable. */
public String[] SolutionVariable () {
    String[] var = new String[this.net.GetSize()];
    for (int i=0; i<this.net.GetSize(); i++) {
        var[i] = this.net.GetVariable(i).GetName();
    }
    return var;
}

/* Display the attributes part of a solution. */
private void DisplayAttributesPart (SolutionAttributes attr) {
    PrintStream out = System.out;
    out.println("Network attributes : ");
    out.println(" Network name : " + attr.network_name);
    out.println(" Network author : " + attr.network_author);
    out.println(" Algorithm : " + attr.algorithm);
}

/* Display the solution part. */
private void DisplaySolutionPart (Solution solution) {
    String s = "";
    for (int i = 0; i < net.GetSize(); i++)
        s += "  " + net.GetVariable(i).GetName() + "  :  " +
            net.GetVariable(i).GetDomain().GetValueName(solution.values[i]) + "\n";
    System.out.println (s);
}

public String DisplaySolution (Solution solution) {
    String s = "";
    for (int i = 0; i < net.GetSize(); i++)
        s += "  " + net.GetVariable(i).GetName() + "  :  " +
            net.GetVariable(i).GetDomain().GetValueName(solution.values[i]) + "\n";
    System.out.println (s);
    return s;
}

public void NotifySolution (Solution solution) {
    // Add the solution
    v.addElement (solution);
    ++number;
    // Display to the console
    if (int_console) {
        DisplayAttributesPart (solution.GetAttributes ());
        System.out.println ("Solution " + Integer.toString (number) + ":");
        DisplaySolutionPart (solution);
    }
}

public void NotifyEnd () {
    // Display to the console
    if (final_console) {
        System.out.println ("Number of solutions found : " + Integer.toString
            (number));
        System.out.println ("Number of enters : " + Integer.toString
            (enters));
    }
}
System.out.println(" Number of backtracks : " + Integer.toString (leaves));
System.out.println(" Number of instantiations : " + Integer.toString (insts));
System.out.println(" Number of consistency checks : " + Integer.toString (checks));
}

/**
 * Count the leaves in backjumping algorithms.
 **/
public void NotifyBackjump () {
    leaves++;
}

/**
 * Leave a recursion level.
 **/
public void NotifyLeaveLevel () {
    leaves++;
}

/**
 * Instantiate a variable.
 **/
public void NotifyInstanciation () {
    insts--;
}

/**
 * Enter a new recursion level.
 **/
public void NotifyEnterLevel (){
    enters++;
}

/**
 * Do a consistency check.
 **/
public void NotifyConsistencyCheck () {
    checks--;
}
C.5 Java source codes for lexical analysis, syntax analysis and other supporting functions within the BDI agent framework

- ActionRoutine.java
- Unification.java
- AppletUnifyBaseBel.java
- Substitution.java
- AppletTokenizer.java
- AppletParser.java
import JCL.*;
import java.util.*;

public class ActionRoutine {
  static final String[] actionlist = {"move","display"};

  public static void runaction(LiteralNetwork constrnt, String actpred, String[] actterm, Vector solnset, String[] solvar, ASAgent agtnm, int Numofsoln) {
    if (actpred.equals("genproposeappt")) {
      genproposeappt(constrnt, actterm, solnset, solvar, agtnm, Numofsoln);
    } else if (actpred.equals("processappt")) {
      processappt(constrnt, actterm, solnset, solvar, agtnm, Numofsoln);
    }
  }

  public static void move(String[] mvargs) {
    //Default change of location (for robot) from 1st argument to 2nd argument (new location).
    //Base belief for robot's location represented by 1st argument is to be removed
    //and be replaced by new location of 2nd argument.
    //Delete: location(robot, 1st argument) & Insert: location(robot, 2nd argument)
    if (mvargs.length == 2) {
      for (int a=1; a <= BeliefsandPlanLib.thebeliefset.size(); a++) {
        beliefnode belnode = BeliefsandPlanLib.thebeliefset.peekNode(a);
        beliefnode prevbelnode = BeliefsandPlanLib.thebeliefset.peekNode(a-1);
        if (belnode.beliefAtom.getSym().equalsIgnoreCase("location")
            && belnode.beliefAtom.term[0]().equals("robot")) {
          literal tmpbelatcn = new literal();
          tmpbelatom.predSymbol = "location";
          tmpbelatom.noOfTerm = 2;
          tmpbelatom.term[0] = new String[2][2];
          tmpbelatom.term[0][0] = "robot";
          tmpbelatom.term[0][1] = "LIRL";
          tmpbelatom.term[1][0] = mvargs[1];
          tmpbelatom.term[1][1] = "LIRA";
          tmpbelnode.beliefAtom = tmpbelatom;
          System.out.println("ActionRoutine : tmpbelnode.beliefAtom = ", tmpbelnode.beliefAtom);
        }
      }
    }
  }
}
ActionRoutine.java

"+tmpbelnode.beliefAtom.noofTerm);
for (int p = 0; p < tmpbelnode.beliefAtom.term.length; p++) {
    System.out.println("ActionRoutine : tmpbelnode.beliefAtom - " +tmpbelnode.beliefAtom.term[p][0]);
} 
*/

tmpbelnode.nextBelief = belnode.nextBelief;
prevbelnode.nextBelief = tmpbelnode;

belnode.nextBelief = null;
} //if
} //mvargs.length == 2

//Alternative change of location with additional argument (1st arg.)
//specifying the item
//involved in change of location.
if (mvargs.length == 3) {
}
} //move

//Action to display solution
public static void display(LiteralNetwork ltnet, Vector solnvect, int Noofsoln)
        {
        CustSolutionManager actsolmgr = new CustSolutionManager(ltnet,
solnvect, Noofsoln);
        Vector tmpsol = actsolmgr.SolutionVector();

        System.out.println("Number of solutions : " +Integer.toString(Noofsoln));
        AppOut.append("\n" +Number of solutions : " +Integer.toString(Noofsoln) + "\n");
        if (Noofsoln > 0) {
            for (int i = 1; i <=Noofsoln; i++) {
                AppOut.append("Solutions : " +Integer.toString(i));
                System.out.println("ActionRoutine Solutions : " +Integer.toString(i));
                AppOut.append(actsolmgr.DisplaySolution((Solution)(tmpsol.elementAt(i))));
            }
        } //if
} //display

//Action to generate trigger event to return solution generated.
public static void genproposeappt(LiteralNetwork solnet, String[] actterm, Vector soln, String[] svar, ASAgent sourceagent, int Numofsoln)
{
    eventnode trigger = new eventnode();
    AEventMsg eventmsg = new AEventMsg();
    //CustSolutionManager gensolmgr = new CustSolutionManager(solnet,
soln, Numofsoln);
    //Vector solnvec = gensolmgr.SolutionVector();
    //String[] vararray = gensolmgr.SolutionVariable();

    trigger.evtOper = '-';
    trigger.evtIdentifier = 'E';
    trigger.evtPredSym = "proposefrom";
    trigger.evtNoOfTerm = 5;
    trigger.evtTerm = new String[5][2];
    trigger.evtTerm[0][0] = actterm[0]; // Request ID
    trigger.evtTerm[0][1] = "TR1";
    trigger.evtTerm[1][0] = sourceagent.name; //from Tsolution
    trigger.evtTerm[1][1] = "TR1";
    trigger.evtTerm[2][0] = actterm[2]; // Person
    trigger.evtTerm[2][1] = "TVAR";
    trigger.evtTerm[3][0] = actterm[3]; // Date
    trigger.evtTerm[3][1] = "TVAR";
    trigger.evtTerm[4][0] = actterm[4]; // Slot
    trigger.evtTerm[4][1] = "TVAR";
    trigger.solnset = soln;
    trigger.solnvar = svar;
    eventmsg.perfomative = "EXTrigger";
eventmsg.receiver = actterm[1];
eventmsg.sender = sourceagent.name; //from Toagent
eventmsg.content = trigger;

ASAgentEvent agtevent = new ASAgentEvent(sourceagent, eventmsg);

// Delivered event to source agent (Fromagent)
((ASAgent) ASAgent.allagents.get(actterm[1])).asEventFired(agtevent);

} // genproposeappt

// Action to process proposed solution return from destination (Toagent)
public static void processappt(LiteralNetwork solnnet, String[] actterm, Vector soln, String[] svar, ASAgent sourceagent, int solnnum) {
    boolean updatereply = false;
    // Enumerate through base beliefs in beliefs set to check for any previous reply
    for (int l = 1; l <= AppletBelPlanLib.thebeliefset.size(); l++) {
        beliefnode tmpbel = AppletBelPlanLib.thebeliefset.peekNode(l);
        if (tmpbel.beliefAtom.predSymbol.equals("reply")
            & tmpbel.beliefAtom.term[0][0].equals(actterm[0])
            & tmpbel.beliefAtom.term[1][0].equals(actterm[1])) {
            updatereply = true;
            break;
        }
    }
    if (! updatereply) {
        literal tmplit = new literal("reply", (byte)2);
        tmplit.term = new String[2][2];
        tmplit.term[0][0] = actterm[0]; // Request ID
        tmplit.term[0][1] = "LTRL";
        tmplit.term[1][0] = actterm[1]; // from Toagent
        tmplit.term[1][1] = "LTRL";
        AppletBelPlanLib.thebeliefset.addEnd(tmplit);
    }
    int replycnt = 0; // No. of reply received
    int bcastcnt = 0; // Counter for number of agent broadcast to (broadcast belief)
    // Enumerate through base beliefs in beliefs set to count for no. of reply with same req ID
    for (int b = 1; b <= AppletBelPlanLib.thebeliefset.size(); b++) {
        beliefnode tmpbelief = AppletBelPlanLib.thebeliefset.peekNode(b);
        if (tmpbelief.beliefAtom.predSymbol.equals("reply")
            & tmpbelief.beliefAtom.term[0][0].equals(actterm[0])) {
            replycnt++; // Count number of reply
        }
        else if (tmpbelief.beliefAtom.predSymbol.equals("broadcast")
            & tmpbelief.beliefAtom.term[0][0].equals(actterm[0])) {
            bcastcnt = Integer.getInteger(tmpbelief.beliefAtom.term[1][0]).intValue();
        }
    }
    if (replycnt == bcastcnt) { // ALL replys for the request ID has been received
        // Insert a new belief for all solution nodes received
        // Search from the solution sets(lists) to look for same request ID reply
        solnmodelist tmpsolnset1 = null; // Tmp soln set for the request ID
        for (int e = 1; e <= AppletBelPlanLib.thesolnlist.size(); e++) {
tmpsolnsetnode = AppletBelPlanLib.thesolnlist.peekNode(e);
if (tmpsolnsetnode.solnset.head.reqID.equals(actterm[0])) {
    tmpsolnset1 = tmpsolnsetnode.solnset;
    break;
} }

//Create the HEAD of new constraint belief
constrtnbelnodeFD tmpconstrtnbelnode = new constrtnbelnodeFD();
constrtnFD tmpconstrtnbel = new constrtnFD();
constrtnbodylistFD tmpconstrtnbodylist = new constrtnbodylistFD();
tmpconstrtnbel.belPredSym = "solution";
tmpconstrtnbel.belNoofTerm = 3;
tmpconstrtnbel.beTerm = new String[3][2];
tmpconstrtnbelnode.constbel = tmpconstrtnbel;
if (tmpsolnset1 != null) {
    //enumerate through all solution nodes in temp solution set
    for (int f = 1; f <= tmpsolnset1.size(); f++) {
        solnnode tmpsolnnode = tmpsolnset1.peekNode(f);
        //Create the BODY of new- constraint belief
        Enumeration solnenum = tmpsolnnode.solution.elements();
        while (solnenum.hasMoreElements()) {  //loop through all
            solution set/node in Vector
                String[][] solval = (String[][]) solnenum.nextElement();
                constrtnbodyFD tmpconstrtnbody = new constrtnbodyFD();
                tmconstrtnbody.beTerm = new String[2];
                tmpconstrtnbody.beTerm = new String[2];
                for (int l=0; l<solval.length; l++) {
                    if (solval[l][0].equals("Date")) {
                        tmpconstrtnbody.beTerm[l] = solval[l][1];
                    } else if(solval[l][0].equals("Slot") { 
                        tmpconstrtnbody.beTerm[l] = solval[l][1];
                    } else {
                        break;
                    }
                }
                //check for duplication before insert new constraint body node
                boolean insert=true;
                for (int k=1; k<=tmpconstrtnbodylist.size(); k++) {
                    constrtnbodyFD constrtnbdy = tmpconstrtnbodylist.peekNode(k);
                    if (constrtnbdy.beTerm[l].equals(tmpconstrtnbody.beTerm[0])
                        & constrtnbdy.beTerm[l].equals(tmpconstrtnbody.beTerm[1])) {
                        insert=false;
                        break;
                    }
                }
                if (insert) {
                    tmpconstrtnbodylist.addEnd(tmpconstrtnbody);
                }
            }
        }
        //for all solution nodes in temp solution set (tmpsolnset1)
    } if tmpsolnset1 != null
}

tmpconstrtnbelnode.bodylist = tmpconstrtnbodylist;
AppletBelPlanLib.theconbel.addEnd(tmpconstrtnbelnode);
// Insert an internally generated "external event" into event queue of the same agent
eventnode processtrigger = new eventnode();

processtrigger.evtOper = '+';
processtrigger.evtIdentifier = "!";
processtrigger.evtPredSym = "proposefrom";
processtrigger.evtNoOfTerm = 4;
processtrigger.evtTerm = new String[4][2];
processtrigger.evtTerm[0][0] = actterm[0];  // Request ID
processtrigger.evtTerm[0][1] = "LTRL";
processtrigger.evtTerm[1][0] = sourceagent.name;  // from Toagent
processtrigger.evtTerm[1][1] = "LTRL";
processtrigger.evtTerm[2][0] = actterm[2];
processtrigger.evtTerm[2][1] = "TVAR";
processtrigger.evtTerm[3][0] = actterm[3];
processtrigger.evtTerm[3][1] = "TVAR";
processtrigger.solnset = soln;
processtrigger.solnvar = svar;

if (replyont > 0 & & replyont < bcastcont) {  // there are existing replies for the request ID
    // Enumerate through all the solution sets(lists) to search for same request ID
    tmpsolnset2 = null;
    solnnode tmpsolnsetnode2 = null;  // Tmp soln set for the request ID
    for (int c=1; c <= AppletBelPlanLib.thesolnlist.size(); c++) {
        tmpsolnsetnode2 = AppletBelPlanLib.thesolnlist.peekNode(c);
        if (tmpsolnsetnode2.solnset.head.reqID.equals(actterm[0])) {
            tmpsolnset2 = tmpsolnsetnode2.solnset;
            break;
        }
    }
    for (int d=1; d <= tmpsolnset2.size(); d++) {
        solnnode tmpsolnnode = tmpsolnset2.peekNode(d);
        if (tmpsolnnode.reqID.equals(actterm[0])
            & & tmpsolnnode.toagent.equals(actterm[1])) {  // to check if already exist a previous reply
            tmpsolnset2.removeNode(tmpsolnnode);  // remove previous reply
            break;
        }
    }
    solnnode newsoln = new solnnode(actterm[0], actterm[1], soln, svar);
    tmpsolnset2.add_End newsoln;  // insert latest reply from agent
    if (tmpsolnset2.size() == null)
        else if (replyont == 0) {  // No reply has been received so far
            solnnode newsoln = new solnnode(actterm[0], actterm[1], soln, svar);
            solnnode solntlist = new solnnodeList();
            replysoln = newreplysoln = new replysoln();
            newsolnlist.add_End newsoln;
            newreplysoln.solnset = newsolnlist;
            AppletBelPlanLib.thesolnlist.add_End(newreplysoln);
        } else if
    }  // processapp
public final class Unification {

    public static bindingstack unification(literal invoc, literal event) {
        //termpairnode termpair = new termpairnode();
        //bindingnode bind = new bindingnode();
        termpairstack tpstack = new termpairstack();
        bindingstack bnstack = new bindingstack();

        //Check for similar functor and arity
        if (invoc.predSymbol.equals(event.predSymbol) &
                invoc.noofTerm == event.noofTerm) {
            boolean eqtermsfig = true; //Flag indicator for equality of
            term-pair (if true)
            boolean uniquesubst = true; //Flag indicator for legal/unique
            substitution (if true)

            // Populate the term-pair stack using plan's invocation and event
            selected
            for (int j=0; j < invoc.term.length; j++) {
                termpairnode termpair = new termpairnode();
                termpair.term1 = invoc.term[j][0];
                termpair.term2 = event.term[j][0];

                //AppOut.appout("termpair.term1 : "+termpair.term1);
                //AppOut.appout("termpair.term2 : "+termpair.term2);
                /*
                System.out.println("termpair.term1 : "+termpair.term1);
                System.out.println("termpair.term2 : "+termpair.term2);
                */
                tpstack.push(termpair); // Push into stack
            }
        }
    }
}
//AppOut.appOut("tpstack.peekNode(c) : " + c + "+tpstack.peekNode(c).term2);
/*
System.out.println("tpstack.size() : " + tpstack.size());
for (int c=1; c <= tpstack.size(); c++) {
    System.out.println("tpstack.peekNode(c) : " + c + "+tpstack.peekNode(c).term1);
    System.out.println("tpstack.peekNode(c) : " + c + "+tpstack.peekNode(c).term2);
}
*/

//Unify terms from plan's invoc and event selected
//Enumerate each term pair collected in term-pair stack
for (int k=i; k <= tpstack.size(); k++) {
    if (!
        tpstack.peekNode(k).term1.equals(tpstack.peekNode(k).term2)) {
        //Both term1 and term2 are literal constant: confirm not a
        //relevant plan-BREAK loop
        if (!
            Character.isUpperCase(tpstack.peekNode(k).term1.charAt(0))
            && ! Character.isUpperCase(tpstack.peekNode(k).term2.charAt(0))
        ) {
            eqltermsflg = false;
            break;
        } else {
            //Propagates substitution into bindingstack: to ensure
            //potential idempotency,
            //generate fully dereferenced form of output (unifiers)
        }
    }
    if (bnstack.size() > 0) {
        //Enumerate bindings/replacements in the stack
        for (int b=1; b <= bnstack.size(); b++) {
            if (!
                Character.isUpperCase(bnstack.peekNode(b).bindTerm1.
                charAt(0))
                && ! Character.isUpperCase(bnstack.peekNode(b).bindTerm2.
                charAt(0))
            ) {
                //term1 is a variable (TVar) & term2 is literal
                //constant (LRL)
                if (!
                    Character.isUpperCase(tpstack.peekNode(k).term1.
                    charAt(0))
                    && ! Character.isUpperCase(tpstack.peekNode(k).term2.
                    charAt(0))
                ) {
                    if (!
                        tpstack.peekNode(k).term1.equals(bnstack.peekNode
                        (b).bindTerm1)
                        && ! tpstack.peekNode(k).term2.equals(bnstack.
                        peekNode(b).bindTerm2)
                    ) {
                        uniquesubst = false;
                        break;
                    }
                } else {
                    //if term1 is a variable (TVar) & term2 is literal
                    //constant (LRL)
                    if (!
                        Character.isUpperCase(tpstack.peekNode(k).term1.
                        charAt(0))
                        && ! Character.isUpperCase(tpstack.peekNode(k).term2.
                        charAt(0))
                    ) {
                        if (!
                            tpstack.peekNode(k).term2.equals(bnstack.peekNode
                            (b).bindTerm1)
                            && ! tpstack.peekNode(k).term1.equals(bnstack.
                            peekNode(b).bindTerm2)
                        ) {
                            uniquesubst = false;
                        }
                    }
                }
            }
        }
    }
}
**** Ensure unique substitution: detect for non-unique bindings
*** bindTerm1 is variable (TVAR), bindTerm1 is literal constant (LTL)
if Character.isUpperCase( bnsNode, peekNode b ). bindTerm1 . charAt .
	next is a variable (TVAR), & term1 is literal constant (LTL)

if Character.isUpperCase( bnsNode, peekNode b ). bindTerm1 . charAt .
	next is a variable (TVAR), & term1 is literal constant (LTL)

**** Both bindTerm1 & bindTerm2 are variable (TVAR)
if Character.isUpperCase( bnsNode, peekNode b ). bindTerm1 . charAt .
	next is a variable (TVAR), & term1 is literal constant (LTL)

**** term1 == bindTerm1
if bnsNode, peekNode k . term1 . equals bnsNode, peekNode

**** term1 == bindTerm1
if term1 == bindTerm1
else if bnsNode, peekNode k . term1 . equals bnsNode, peekNode

**** term1 == bindTerm1
if term1 == bindTerm1
else if term1 is a TVAR

**** term1 is a TVAR, & term1 is a literal constant (LTL)
if term1 . equals bnsNode, peekNode

**** term1 is a TVAR, & term1 is a literal constant (LTL)
if term1 is a variable (TVAR), & term1 is a literal constant (LTL)
variable (TVAR)
    if (! Character.isUpperCase(tpstack.peekNode(k).terml.
        charAt(0))
        && Character.isUpperCase(tpstack.peekNode(k).terml.
        charAt(0))
            ///***term2 == bindTerm1
        if (tpstack.peekNode(k).terml.equals(bnstack.peekNode
            (b).bindTerm1))
            bnstack.peekNode(b).bindTerm1 =
                tpstack.peekNode(k).terml;
        } ///***term1 == bindTerm1
    } ///***term2 == bindTerm2
        else if (tpstack.peekNode(k).term2.equals(bnstack.
            peekNode(b).bindTerm2))
            bnstack.peekNode(b).bindTerm2 =
                tpstack.peekNode(k).terml;
        } ///***term1 == bindTerm2
    } ///if term2 is TVAR
} ///if bindTerm1 & bindTerm2 are TVAR

} ///for - Enumerate existing bindings in stack *****
} ///********** Propagates into bindingstack -
  (bnstack.size() > 0)

//Not legal unique substitution
if (! uniquesubst) {
    break;
} ///if

//Propagates substitution into remaining term-pairs in
term-pair stack

//**********************************************************
//for (int t=k+1; t <= tpstack.size(); t++) {
    //*******Ensure unique substitution : detect for non-unique
    bindings
    //*** term1 is variable (TVAR)
    if (Character.isUpperCase(tpstack.peekNode(t).terml.
        charAt(0))
        && ! Character.isUpperCase(tpstack.peekNode(t).term2.
        charAt(0))
            { /*term1 is a variable (TVAR) & term2 is a literal
            constant
            if (Character.isUpperCase(tpstack.peekNode(k).terml.
                charAt(0))
                && ! Character.isUpperCase(tpstack.peekNode(k).term2.
                charAt(0))
                    { if
                        (tpstack.peekNode(k).terml.equals(tpstack.
                        peekNode(t).terml)
                        && ! tpstack.peekNode(k).term2.equals(tpstack.
                        peekNode(t).term2))
                        { uniquesubst = false;
                            break;
                        } ///if
                    } ///if term1 is a variable (TVAR) & term2 is a literal
            constant
            } //if term2 is a variable (TVAR) & term1 is a literal
            constant
            if (! Character.isUpperCase(tpstack.peekNode(k).term1.
                charAt(2))
                && Character.isUpperCase(tpstack.peekNode(k).term2.
                charAt(0))
                    { if
                        (tpstack.peekNode(k).term2.equals(tpstack.
                        peekNode(t).terml)
                        && ! tpstack.peekNode(k).terml.equals(tpstack.
                        peekNode(t).term2))
                        { uniquesubst = false;
                            break;
                        } if
                    } if
                } //if term2 is a variable (TVAR) & term1 is a literal
            constant
        } if
    } //for
constant

// Unique substitution - term1 is variable (TVAR)

***** Ensure unique substitution: detect for non-unique bindings

*** term2 is variable (TVAR)
if (Character.isUpperCase(tpstack.peekNode(t).term1.chaarAt(c)))
    && ! Character.isUpperCase(tpstack.peekNode(t).term1.chaarAt(c))
        term1 is a variable (TVAR) & term2 is a literal constant
if (Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
    && ! Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
        term2 is a variable (TVAR) & term1 is a literal constant
if (! Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
    && ! Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
        term2 is a variable (TVAR) & term1 is a literal constant
if (! Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
    && ! Character.isUpperCase(tpstack.peekNode(t).term1.chaarAt(c))
        uniqueSubst = false;
    break;
} id
} // if term1 is a variable (TVAR) & term2 is a literal constant

// term2 is a variable (TVAR) & term1 is a literal constant
if (! Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
    && ! Character.isUpperCase(tpstack.peekNode(t).term1.chaarAt(c))
        term2 is a variable (TVAR) & term1 is a literal constant
if (tpstack.peekNode(k).term1.equals(tpstack.peekNode(t).term1)
    && ! tpstack.peekNode(k).term1.equals(tpstack.peekNode(t).term1)
        uniqueSubst = false;
    break;
} id
} // if term2 is a variable (TVAR) & term1 is a literal constant

*** Both term1 & term2 are variable (TVAR)
if (Character.isUpperCase(tpstack.peekNode(t).term1.chaarAt(c))
    && Character.isUpperCase(tpstack.peekNode(t).term1.chaarAt(c))
        term1 is a variable (TVAR) & term2 is a literal constant
if (Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
    && Character.isUpperCase(tpstack.peekNode(k).term1.chaarAt(c))
        term1 is a variable (TVAR) & term2 is a literal constant

    if (tpstack.peekNode(k).term1.equals(tpstack.peekNode(t).term1)
        && ! tpstack.peekNode(k).term1.equals(tpstack.peekNode(t).term1)
            uniqueSubst = false;
        break;
    } id
    } // if term1 is a variable (TVAR) & term2 is a literal constant

    ***term1 == term2
    if (tpstack.peekNode(k).term1.equals(tpstack.peekNode(t).term1)
        && tpstack.peekNode(t).term1 = tpstack.peekNode(k).term2;
    } ***term1 == term1
    } ***term1 == term2
else if (tpstack.peekNode(k).term1.equals(tpstack.peekNode(t).term1)
    && tpstack.peekNode(k).term1 = tpstack.peekNode(t).term2;
    } ***else term1 == term2
    } if term1 is TVAR

term1 is a literal constant & term2 is a variable (TVAR)
if (!Character.isUpperCase(charAt(0)))
    & Character.isUpperCase(tpstack.peekNode(k).term1.
    charAt(0)))
        ***term1 == term1
    if
    (tpstack.peekNode(k).term2.equals(tpstack.peekNode(t).
    term1))
        tpstack.peekNode(t).term1 =
        tpstack.peekNode(k).term1;
        ***term1 == term1
    else if (tpstack.peekNode(k).term2.equals(tpstack.
    peekNode(t).term2))
        tpstack.peekNode(t).term2 =
        tpstack.peekNode(k).term2;
        ***else term1 == term2
    if (term2 is TVAR
    )
        if (term1 & term2 are TVAR
    )
            for ********** Propagates into remaining term-pairs

    Not legal unique substitution
    if (!uniquesubst) {
        break; // Break from enumerating remaining term-pair in
        stack
    }

    bindingnode bind = new bindingnode();
    bind.bindTerm1 = tpstack.peekNode(k).term1;
    bind.bindTerm2 = tpstack.peekNode(k).term2;

    bstack.push(bind);
    }
else
    *** if term1 == term2
    *** term1 = term2 ***
else if (tpstack.peekNode(k).term1.equals(tpstack.peekNode(k).
    term2))
        continue; // proceeds with next term-pair in stack: Possible
        relevant plan
    *** else if term1 = term2
    } // Enumerate each pair of term-pair in stack ******

// Return all substitution pairs (in bstack) if unify
 successfully
if (eq1terms & & uniquesubst) {
    return bstack;
}
else {
    bstack = null;
    return bstack;
}
else
    if similar functor and arity
    else {
        bstack = null;
        return bstack;
    }
else different functor or arity
} unification

} Class
public final class AppletUnifyBaseBel {

    // Determine if ground formula
    public static boolean groundformula(literal aformula) {
        // Enumerate through terms of aformula
        boolean ground = true; // Indicator for ground formula
        for (int k=0; k < aformula.term.length; k++) {
            if (Character.isUpperCase(aformula.term[k][1].charAt(0))) {
                ground = false;
                break;
            }
        } // for each context belief's term (k)
        return ground;
    } // groundformula

    // Determine if logical consequence (for ground formula/formula without variables)
    public static boolean logconseq(literal bformula) {
        // Indicator of bformula is ground and logical consequence
        boolean grdlgcsq = false;
        boolean ground = true; // ALWAYS ground (to be explicit!)
        // Enumerate through base beliefs in beliefs set
        for (int i=1; i <= AppletBelPlanLib.thebeliefset.size(); i++) {
            beliefnode tmpbelnode = AppletBelPlanLib.thebeliefset.peekNode(i);
            // Similar functor and arity
            if (tmpbelnode.beliefAtom.predSymbol.equals(bformula.predSymbol) &&
                tmpbelnode.beliefAtom.noofTerm == bformula.noofTerm) {
                // Indicator of bformula is log. consequ. of current base belief (i)
                boolean logiconseq = true;
                for (int m=0; m < tmpbelnode.beliefAtom.term.length; m++) {
                    if (!
                        tmpbelnode.beliefAtom.term[m][1].equals(bformula.term[m][1]) &&
                        ground) {
                        logiconseq = false;
                        break; // Stop scanning through the remaining terms
                    }
                } // for (m)
                if (logiconseq) {
                    grdlgcsq = true; // bformula is ground and logical
                    consequence
                    break; // Stop scanning through the remaining base beliefs
                } // if
            } // if similar functor & arity
        } // for each base belief in beliefs set [thebeliefset] (1)
        return grdlgcsq;
    } // logconseq

    // Unifying with current base beliefs
    public static bindingstack unifybasebel(literal uformula) {
        bindingstack tmpbdstk = null; // binding stack for unifiers if unifiable
        // Indicator of uformula is unifiable and a logical consequence
        boolean unilgcsq = false;
        // Enumerate through base beliefs in beliefs set
        for (int n=1; n <= AppletBelPlanLib.thebeliefset.size(); n++) {
            beliefnode tmpbelnd = AppletBelPlanLib.thebeliefset.peekNode(n);
            // ...
literal tmpbel = new literal();
literal tmpctx = new literal();

tmpbel.predSymbol = tmpbelnd.beliefAtom.predSymbol;
tmpbel.noofTerm = tmpbelnd.beliefAtom.noofTerm;
tmpbel.term = tmpbelnd.beliefAtom.term;

tmpctx.predSymbol = uformula.predSymbol;
tmpctx.noofTerm = uformula.noofTerm;
tmpctx.term = uformula.term;

tmpbdstk = Unification.unification(tmpbel, tmpctx);

    //If unifyable (tmpbdstk not null)
    if (tmpbdstk != null) {
        break;       //Stop scanning through the remaining base beliefs
    }  //if
}  //for each base belief in beliefs set [thebeliefset] (n)

return tmpbdstk;      //Not unifyable if tmpbdstk == null
}  //unifybasebel

) //class AppletUnifyBaseBel
public final class Substitution {

    public static void substitution(bindingstack binds, String[][] terms) {
        for (int i = 1; i <= binds.size(); i++) {
            for (int j = 0; j < terms.length; j++) {
                // Both bindTerm1 and bindTerm2 are variables (TVAR).
                // No replacement/substitution takes place as this will NOT bring any
                // significant changes to the terms
                // AppOut.append("Substitution - binds.peekNode("-i-").bindTerm1 : "
                // binds.peekNode(i).bindTerm1);
                // AppOut.append("Substitution - binds.peekNode("-i-").bindTerm2 : "
                // binds.peekNode(i).bindTerm2);
                System.out.println("binds.peekNode("-i-").bindTerm1 :
                " + binds.peekNode(i).bindTerm1);
                System.out.println("binds.peekNode("-i-").bindTerm2 :
                " + binds.peekNode(i).bindTerm2);
                System.out.println("terms[-j-][0] : " + terms[j][0]);
                System.out.println("binds.peekNode("-i-").charAt(0) : " + "binds.peekNode(i).charAt(0));
                if (Character.isUpperCase(binds.peekNode(i).bindTerm1.charAt(0))
                        && Character.isUpperCase(binds.peekNode(i).bindTerm2.charAt(0))) {
                    // if
                    // bindTerm1 is variables (TVAR) and bindTerm2 is literal constant
                    // (LTRL)
                    if (Character.isUpperCase(binds.peekNode(i).bindTerm1.charAt(0))
                            && ! Character.isUpperCase(binds.peekNode(i).bindTerm2.charAt(0))) {
                        if (binds.peekNode(i).bindTerm1.equals(terms[i][0])) {
                            terms[i][0] = binds.peekNode(i).bindTerm2;
                            // AppOut.append("bindTerm1 is variables (TVAR) & terms[-j-][0] : 
                            // " + "binds.peekNode(i).bindTerm1 = " + "binds.peekNode(i).bindTerm2");
                            System.out.println("bindTerm1 is variables (TVAR) & terms[-j-][0] : 
                            " + "binds.peekNode(i).bindTerm1 = " + "binds.peekNode(i).bindTerm2");
                        } else {
                            System.out.println("bindTerm2 is variables (TVAR) & terms[-j-][0] : 
                            " + "binds.peekNode(i).bindTerm2");
                        }
                    }
                }
            }
        }
    }
}
Substitution.java

*/       
    }    //if
    }    //for terms.length
}    //for binds.size()
}    //substitution
}    //GenApplPlan thread
import java.awt.*;
import java.io.*;
import java.util.StringTokenizer;

public class AppletTokenizer {

    public String checkpp(String tokenstr) {
        String tokenstring, returnstr = "";
        boolean parenthe = false, period = false;
        int tokenlen = 0, leftparencnt = 0, rightparencnt = 0;

        tokenstring = tokenstr;
        tokenlen = tokenstring.length();

        for (int i = 0; i <= (tokenlen - 1); i++) {
            switch (tokenstring.charAt(i)) {
                case '(':
                    if (tokenlen == 1) {
                        returnstr = " + tokenstring.charAt(i);
                        break;
                    } //if
                    else {
                        leftparencnt = leftparencnt + 1;
                        if (i == 0) {
                            returnstr = " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //if
                        else {
                            returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //else
                    } //else
                    case ')':
                        if (tokenlen == 1) {
                            returnstr = " + tokenstring.charAt(i);
                        } //if
                        else {
                            rightparencnt = rightparencnt + 1;
                            returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //else
                        break;
                } //case '
                case '.':
                    if (tokenlen == 1) {
                        returnstr = " + tokenstring.charAt(i);
                    } //if
                    else {
                        returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                    } //else
                    break;
                } //case '
                case '&':
                    if (tokenlen == 1) {
                        returnstr = " + tokenstring.charAt(i);
                    } //if
                    else {
                        if (i == 0) {
                            returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //if
                    } //else
                    break;
                } //case '
            } //switch
        } //for
        return returnstr;
    } //checkpp

    public String checkpp(String tokenstr) {
        String tokenstring, returnstr = "";
        boolean parenthe = false, period = false;
        int tokenlen = 0, leftparencnt = 0, rightparencnt = 0;

        tokenstring = tokenstr;
        tokenlen = tokenstring.length();

        for (int i = 0; i <= (tokenlen - 1); i++) {
            switch (tokenstring.charAt(i)) {
                case '(':
                    if (tokenlen == 1) {
                        returnstr = " + tokenstring.charAt(i);
                        break;
                    } //if
                    else {
                        leftparencnt = leftparencnt + 1;
                        if (i == 0) {
                            returnstr = " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //if
                        else {
                            returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //else
                    } //else
                    case ')':
                        if (tokenlen == 1) {
                            returnstr = " + tokenstring.charAt(i);
                        } //if
                        else {
                            rightparencnt = rightparencnt + 1;
                            returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //else
                        break;
                } //case '
                case '.':
                    if (tokenlen == 1) {
                        returnstr = " + tokenstring.charAt(i);
                    } //if
                    else {
                        returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                    } //else
                    break;
                } //case '
                case '&':
                    if (tokenlen == 1) {
                        returnstr = " + tokenstring.charAt(i);
                    } //if
                    else {
                        if (i == 0) {
                            returnstr = returnstr + " + tokenstring.charAt(i) + " + tokenstring.charAt(i) + " ";
                        } //if
                    } //else
                    break;
                } //case '
            } //switch
        } //for
        return returnstr;
    } //checkpp

    public static void main(String[] args) {
        String input = "Hello, World!
This is a test.
Sample text.
";
        String output = checkpp(input);
        System.out.println(output);
    }
} //public class AppletTokenizer
else {
    if ((i+1) < tokenlen) {
        returnstr = returnstr + "  " + tokenstring.charAt(i) + " ";
        break;
    } //if
    else {
        returnstr = returnstr + "  " + tokenstring.charAt(i);
        break;
    } //else
} //else

//case ':'
if (tokenlen == 1) {
    returnstr = "" + tokenstring.charAt(i);
    break;
} //if
else {
    if (i == 0) {
        returnstr = "" + tokenstring.charAt(i) + " ";
        break;
    } else {
        if ((i+1) < tokenlen) {
            returnstr = returnstr + "  " + tokenstring.charAt(i) + " ";
            break;
        } //if
        else {
            returnstr = returnstr + "  " + tokenstring.charAt(i);
            break;
        } //else
    } //else
} //else

//case '+':
if (tokenlen == 1) {
    returnstr = "" + tokenstring.charAt(i);
    break;
} //if
else {
    if (i == 0) {
        //returnstr = returnstr + "  " + tokenstring.charAt(i);
        returnstr = "" + tokenstring.charAt(i) + " ";
        break;
    } else {
        if ((i+1) < tokenlen) {
            returnstr = returnstr + "  " + tokenstring.charAt(i);
            //returnstr = returnstr + "  " + tokenstring.charAt(i) + " ";
            break;
        } //if
        else {
            returnstr = returnstr + "  " + tokenstring.charAt(i);
            break;
        } //else
    } //else
} //else

//case '-':
if (tokenlen == 1) {
    returnstr = "" + tokenstring.charAt(i);
    break;
} //if
else {
    if (i == 0) {
        //returnstr = returnstr + "  " + tokenstring.charAt(i);
        returnstr = "" + tokenstring.charAt(i) + " ";
        break;
    } else {
        if ((i+1) < tokenlen) {
            returnstr = returnstr + "  " + tokenstring.charAt(i):
            returnstr = returnstr + "  " + tokenstring.charAt(i) + " ";
        } //if
        else {
            returnstr = returnstr + "  " + tokenstring.charAt(i)
            break;
        } //else
    } //else
} //else
case '!':
    if (tokenlen == 1) {
        returnstr = " + tokenstring.charAt(i);
        break;
    } //if
    else {
        if (i == 0) {
            returnstr = " + tokenstring.charAt(i) + "
            break;
        }
    } //else
} //else

case '?':
    if (tokenlen == 1) {
        returnstr = " + tokenstring.charAt(i);
        break;
    } //if
    else {
        if (i == 0) {
            returnstr = " + tokenstring.charAt(i) + ";
            break;
        }
    } //else
} //else

case '<':
    //Must be exactly '<=' (2 chars)
    //if (tokenlen == 1) {
    //    returnstr = " + tokenstring.charAt(i);
    //    break;
    //} //if
    if (tokenstring.charAt(i+1) == '=') {
        returnstr = " + tokenstring.charAt(i);
        break;
    } //if
    else {
        if (i == 0) {
            if (tokenstring.charAt(i+1) == '-') {
                returnstr = " + tokenstring.charAt(i) + tokenstring.charAt(i+1) + "
                break;
            }
        }
    } //else
} //else
else {
  if (tokenstring.charAt(i+1) == '_') {
    if ((i+2) < tokenlen) {
      returnstr = returnstr + " " + tokenstring.charAt(i) +
      tokenstring.charAt(i+1) + " ";
      i++;
      break;
    } //if
  } else {
    returnstr = returnstr + " " + tokenstring.charAt(i) +
      tokenstring.charAt(i+1);
    i++;
    break;
  } //else
} //if
else {
  if ((i+1) < tokenlen) {
    returnstr = returnstr + " " + tokenstring.charAt(i) +
      " ";
    break;
  } //if
else {
    returnstr = returnstr + " " + tokenstring.charAt(i);
    break;
  } //else
} //else
}

default:
  returnstr = returnstr + tokenstring.charAt(i);
  break;
} //switch
} //for i

return returnstr;
} //checkpp

/***Count the number of token within a token string

public int returntoken(String lexiconstr) {
  int lexiconcnt = 0;
  StringTokenizer st = new StringTokenizer(lexiconstr, " ");
  lexiconcnt = st.countTokens();
  return lexiconcnt;
} //returntoken

****** MAIN PROGRAM ******

public void tokenizer (TextArea textsrc, TextArea tkntext) {
  AppletTokenizer tokentext = new AppletTokenizer();
  StringBuffer tknbuffer = new StringBuffer();
  StringBufferInputStream ps = new StringBufferInputStream(textsrc.getText());

  try {
    StreamTokenizer stok = new StreamTokenizer(new DataInputStream(ps));
    stok.wordChars('a', 'z');
    //stok.wordChars('0', '9');
    stok.wordChars('!', '-');
    stok.wordChars('<', ' ');
    stok.wordChars('?', ' ');
    stok.wordChars('(', ')');
    stok.wordChars('+', '+');
    stok.wordChars('!', '!');
    stok.wordChars('@', '@');
    stok.wordChars('_', '_');
    stok.wordChars('-', '-');
    stok.wordChars('#', '#');
    stok.wordChars('>', '>');
```java
import java.util.Scanner;

public class AppletTokenizer {
    public static void main(String[] args) {
        Scanner stdin = new Scanner(System.in);
        String line = stdin.nextLine();
        System.out.println(line);
    }
}
```
```java
String tmpkn = "";

while (tmpkn retorna) case 1 { //
    StringTokenizer tmpkn = new StringTokenizer tmpstring, 
    }
    tmpkn = tmpkn.nextElement ;
}
else (token.nextToken tmpstring = 1
else {
    tmpkn = tmpstring;
}

Initialize if not continuation to next line indicates by 'g'

if (tmpkn.charAt 1) == 'g' { //
    if the last token is a numeric constant, append '1' to the
    token to indicate it is
    if (Character.isDigit tmpkn.charAt ;
        tmpkn.charAt ; == ' ' && Character.isDigit tmpkn,
        charAt = ' ';
        linestring = linestring '1';
    }

    linestring = linestring 'n';
    String.append linestring;

    *** Reinitialize variables
    token = 1;
    linestring = "";
    finalToken = 1;
    sumnum = sumnum - 2;

    if token == dot { //
        System.out.println 'Line: ' - lineum ;
    if
    if 'g' from end of statement string
    else if tmpkn.length == 0 && tmpkn.charAt 1 == 'g' &&
    tmpkn.charAt 1 == ' ';

    StringTokenizer tmpkn = new StringTokenizer linestring,
    ;
    String implinestr = "";

    for string i=implinestr, nextToken : tmpkn.hasMoreTokens
        if i' == 'g' { //
            if implinestr == "" { //
                implinestr = "";
            }
            else //
                implinestr = implinestr '1';
            else
                implinestr = implinestr '1';
    } else
        linestring = implinestr;
    else
```

```
catch (Exception = / System.err.println e ;
```
return;  //catch

String tknoutput = tknbuffer.toString();
tkntext.setText(tknoutput);
tokentext = null;
}  //main
}  //class
/** Perform Syntactical Analysis based on the following syntax symbol table.
  Receive input from tokenized textarea and append result to parsed textarea.
  Statements begin with '#' indicate variables-domain declaration.
  Statements begin with '%' indicate constraint axiom declaration.
  Parser modified on 6 Apr 1999 for inequilities constraint expressions in real constraint domain (partial).
  Parser modified on 15 Aug 1999 to cater for constraint expression in Finite Domain (FD) constraint.
*/

import java.awt.*;
import java.io.*;
import java.util.StringTokenizer;

class AppletParser {
  static final int LTRL=1, LPRN=2, TVAR=3, RPRN=4, SSUG=5, CONJ=6, STOP=7;
  static final int GOAL=8, OPER=9, CTXT=10, PRED=11, TERM=12, ACTN=13,
  CNXL=14;
  static final int CONS=15, ROPR=16;

  //***Parsing for the INDIVIDUAL TOKEN STRING
  //***Return the type of token based on symbol table
  public int tokenparser(String inputtoken) {
    int tokentype = 0, tokenlen = inputtoken.length();

    if (inputtoken.length() == 0 && !Character.isLetterOrDigit(inputtoken.
    charAt(0))) {
      switch (inputtoken.charAt(0)) {
        case '(': tokentype = LPRN;
                    return tokentype;
        case ')': tokentype = RPRN;
                    return tokentype;
        case '&': tokentype = SSUG;
                    return tokentype;
        case ':': tokentype = CONJ;
                    return tokentype;
        case ':': tokentype = STOP;
                    return tokentype;
        case ':': tokentype = CTXT;
                    return tokentype;
        case '!': tokentype = GOAL;
                    return tokentype;
        case '?':
tokentype = GOAL;
return tokentype;
case '+':
tokentype = OPER;
return tokentype;
case '-':
tokentype = OPER;
return tokentype;
case '*':
tokentype = OPER;
return tokentype;
case '/':
tokentype = OPER;
return tokentype;
case '=':
tokentype = ROPR;
return tokentype;
default:
System.out.println("Unrecognised Character !!!" + inputtoken);
return tokentype; //Return tokentype = 0
}  //switch

}  //if
else if (Character.isUpperCase(inputtoken.charAt(0))) {  //Identify TVAR
if (inputtoken.length() == 1) {
tokentype = TVAR;
return tokentype;
}  //else
else {
if (this.tvarpaser(inputtoken)) {
tokentype = TVAR;
return tokentype;
}  //if
else {
System.out.println("Unrecognised Term Variable !!!" + inputtoken);
}  //else
}  //else

} //if
else if (inputtoken.charAt(0) == '<' || inputtoken.charAt(0) == '>') {  //Identify SSUG
if (this.ssugparser(inputtoken)) {  //Identify SSUG
tokentype = SSUG;
return tokentype;
}  //if
else if (this.roprparser(inputtoken)) {  //Identify ROPR
tokentype = ROPR;
return tokentype;
}  //if
else {
System.out.println("Unrecognised Syntactic Sugar or Relational Operator !!!" + inputtoken);
}  //else
} //if
else if (Character.isDigit(inputtoken.charAt(0)) || (inputtoken.charAt(0) == '-' && Character.isDigit(inputtoken.charAt(1)))) {  //Identify CONS
if (Character.isLowerCase(inputtoken.charAt(0)) && this.ltrilparser(inputtoken)) {  //Identify LTRL
return tokentype;
}  //if
else {
System.out.println("Unrecognised Literal Type or Numberic String !!!" + inputtoken);
}  //else
}  //else

} //if
else if (inputtoken.charAt(0) == '<') { //Identify SSUG
    if (!this.ssugparser(inputtoken)) {
        tokentype = SSUG;
        return tokentype;
    } //if
    else {
        System.out.println("Unrecognised Syntactic Sugar !!!" + inputtoken);
    } //else
} //if
*/

return tokentype; //Return tokentype = 0
} //tokenparser

//***Parsing for TERM VARIABLE (TVAR)
//***3 automaton states : initial state=0, accepting state=1, error state=2

private boolean tvarparser(String tvarstr) {
    boolean trmvar = false;
    int tstate = 0;

    //***process begin from 2nd char
    for (int i = 1; i <= (tvarstr.length() - 1); i++) {
        if (Character.isLetterOrDigit(tvarstr.charAt(i))) {
            tstate = 1; //Accepting State
        } //if
        else {
            tstate = 2; //Error State
            break;
        } //else
    } //for

    if (tstate == 1) {
        trmvar = true;
    } //if

    return trmvar;
} //tvarparser

//***Parsing for PREDICATE/TERM STRING (LTRL)
//***3 automaton states : initial state=0, accepting state=1, error state=2

private boolean ltrlparser(String ltrlstr) {
    boolean predterm = false;
    int lstate = 0;

    //***process begin from 2nd char
    if (ltrlstr.length() == 1
        && Character.isLetter(ltrlstr.charAt(0))
        && Character.isLowerCase(ltrlstr.charAt(0))) {
        lstate = 1;
    } else {
        for (int i = 1; i <= (ltrlstr.length() - 1); i++) {
            if (Character.isLowerCase(ltrlstr.charAt(i))) {
                lstate = 1; //Accepting State
            } //if
            else if (Character.isDigit(ltrlstr.charAt(i))) {
                lstate = 1; //Accepting State
            } else {
                lstate = 2; //Error State
                break;
            } //else
        } //for
    } //else

    if (lstate == 1) {
        predterm = true;
    } //if

    return predterm;
} //ltrlparser
//***Parsing for SYNTACTIC SUGAR (SSUG)
//***4 automaton states : initial state=0, accepting state=2, error state=3
//***State 1 is intermediate state
private boolean ssugparser(String ssugstr) {
    boolean synsug = false;
    int sstate = 0;

    //***process begin from 2nd char
    for (int i = 1; i <= (ssugstr.length() - 1); i++) {
        if (ssugstr.charAt(i) == '{') {
            sstate = 2; //Accepting State
        } else {
            sstate = 3; //Error State
            break;
        }
    } //for

    if (sstate == 2) {
        synsug = true;
    } //if

    return synsug;
} //ssugparser

//***Parsing for RELATIONAL OPERATOR (ROPR : <=, >=)
//***4 automaton states : initial state=0, accepting state=2, error state=3
//***State 1 is intermediate state
private boolean roprparser(String roprstr) {
    boolean relopr = false;
    int sstate = 0;

    //***process begin from 2nd char
    for (int i = 1; i <= (roprstr.length() - 1); i++) {
        if (roprstr.charAt(i) == '=') {
            sstate = 2; //Accepting State
        } else {
            sstate = 3; //Error State
            break;
        } //else
    } //for

    if (sstate == 2) {
        relopr = true;
    } //if

    return relopr;
} //roprparser

//***Automaton for VARIABLE-DOMAIN PARSING
//***6 automaton states : initial state=0, accepting state=4, error state=5
//***States 1, 2, 3 are intermediate states
private int varautomaton(String variablestr) {
    int toktype = 0, varstate = 0;

    StringTokenizer varstr = new StringTokenizer(variablestr, " ");

    while (varstr.hasMoreTokens()) {
        String nextstr = varstr.nextToken();
        toktype = tokenparser(nextstr);
        //toktype = tokenparser(varstr.nextToken());
        System.out.println("TOKEN - "+nextstr);
        System.out.println("TOKTYPE - "+toktype);
        if (toktype < 1 || toktype > 16) {
```java
AppletParser.java

System.out.println("SYNTAX ERROR: Unrecognised token(s) in variable-domain Statement !!!");
break;
}

if (varstate == 0) {
System.out.println("varstate : " + varstate);
if (toktype == TVAR) {
    varstate = 1;
} else {
    varstate = 5;
    break;
}
} //if varstate==0
else if (varstate == 1) {
System.out.println("varstate : " + varstate);
if (toktype == CTXT) {
    varstate = 2;
} else {
    varstate = 5;
    break;
}
} //if varstate==1
else if (varstate == 2) {
System.out.println("varstate : " + varstate);
if (toktype == CONS || toktype == LTRL) {
    varstate = 3;
} else {
    varstate = 5;
    break;
}
} //if varstate==2
else if (varstate == 3) {
System.out.println("varstate : " + varstate);
if (toktype == STOP) {
    if (varstr.hasMoreTokens()) {
        varstate = 5;
        break;
    } else {
        varstate = 4;
    }
} else if (toktype == CONS || toktype == LTRL) {
    varstate = 3;
} else {
    varstate = 5;
    break;
}
} //if varstate==3
else if (varstate == 4) {
System.out.println("varstate : " + varstate);
varstate = 5;
//FOOL-PROOF: no more token after state 4
break;
} //if varstate==4
} //while
return varstate;
} //varautomaton

/*** Automaton for CONSTRAINT AXIOM PARSING
*** 8 automaton states: initial state=0, accepting state=6, error state=7
*** States 1, 2, 3, 4, 5 are intermediate states

private int axmautomaton(String variablestr) {
    int toktype = 0, axmstate = 0;
    StringTokenizer varstr = new StringTokenizer(variablestr, " ");
    ..
```

while (varstr.hasMoreTokens()) {
    String nextstr = varstr.nextToken();
    toktype = tokenparser(nextstr);
    //toktype = tokenparser(varstr.nextToken());

    System.out.println("TOKEN = "+nextstr);
    System.out.println("TOKTYPE = "+toktype);

    if (toktype < 1 || toktype > 16) {
        System.out.println("SYNTAX ERROR: Unrecognised token(s) in constraint axiom statement!!!");
        break;
    }

    if (axmstate == 0) {
        if (toktype == TVAR) {
            axmstate = 1;
        } else {
            axmstate = 7;
            break;
        }
    } else if (axmstate == 1) {
        if (toktype == CTXT) {
            axmstate = 2;
        } else if (toktype == TVAR) {
            axmstate = 1;
        } else {
            axmstate = 7;
            break;
        }
    } else if (axmstate == 2) {
        if (toktype == LPRN) {
            axmstate = 3;
        } else {
            axmstate = 7;
            break;
        }
    } else if (axmstate == 3) {
        if (toktype == CONS | toktype == LTRL) {
            axmstate = 4;
        } else {
            axmstate = 7;
            break;
        }
    } else if (axmstate == 4) {
        if (toktype == CONS | toktype == LTRL) {
            axmstate = 4;
        } else if (toktype == RPRN) {
            axmstate = 5;
        } else {
            axmstate = 7;
            break;
        }
    } else if (axmstate == 5) {
        if (toktype == STOP) {
            axmstate = 6;
        } else if (toktype == LPRN) {
            axmstate = 3;
        } else {
            axmstate = 7;
        }
    }
private int beliefautomaton(String beliefstr) {
    boolean ruleflag = false; //belief rule indicator (<=)
    int toktype = 0, belstate = 0;
    StringTokenizer belstr = new StringTokenizer(beliefstr, " ");
    while (belstr.hasMoreTokens()) {
        toktype = tokenparser(belstr.nextToken()) ;
        if (toktype < 1 || toktype > 16) {
            System.out.println("SYNTAX ERROR: Unrecognised token(s) in Belief Statement !!!");
            break;
        }
        if (belstate == 0) {
            if (toktype == LTRL) {
                belstate = 1;
                System.out.println("TOKTYPE 01 -"+toktype);
            } else if (toktype == LPRN && ruleflag) {
                belstate = 2;
                System.out.println("TOKTYPE 02 -"+toktype);
            } else {
                belstate = 6;
                System.out.println("TOKTYPE 06 -"+toktype);
                break;
            }
        } else if (belstate == 1) {
            if (toktype == LPRN) {
                belstate = 2;
                System.out.println("TOKTYPE 12 -"+toktype);
            } else {
                belstate = 6;
                System.out.println("TOKTYPE 16 -"+toktype);
                break;
            }
        } else if (belstate == 2) {
            if (toktype == TVAR || toktype == LTRL || toktype == CONS) {
                belstate = 3;
                System.out.println("TOKTYPE 23 -"+toktype);
            } else {
                belstate = 6;
                System.out.println("TOKTYPE 26 -"+toktype);
                break;
            }
        } else if (belstate == 3) {
            if (toktype == TVAR || toktype == LTRL || toktype == CONS) {
                belstate = 3;
                System.out.println("TOKTYPE 33 -"+toktype);
            }
else if (toktype == RPRN) {
    belstate = 4;
    System.out.println("TOKTYPE 34 -"+toktype);
}  
else {
    belstate = 6;
    System.out.println("TOKTYPE 36 -"+toktype);
    break;
}
    //if belstate==3
else if (belstate == 4) {
    if (toktype == STOP) {
        if (belstr.hasMoreTokens()) {
            belstate = 6;
        System.out.println("TOKTYPE 46 -"+toktype);
        break;
        }  
    else {
        belstate = 5;
        System.out.println("TOKTYPE 45 -"+toktype);
    }
    else if (toktype == SSUG) {
        if (ruleflag) {  //if more than one SSUG
            belstate = 6;
        System.out.println("TOKTYPE 46 -"+toktype);
        break;
        }  
    else {
        belstate = 0;
        System.out.println("TOKTYPE 40 SSUG -"+toktype);
        ruleflag = true;
    }
    else if (toktype == CONJ) {
        if (! ruleflag) {  //no CONJ before a SSUG
            belstate = 6;
        System.out.println("TOKTYPE 46 -"+toktype);
        break;
        }  
    else {
        belstate = 0;
        System.out.println("TOKTYPE 40 CONJ -"+toktype);
    }
    }  
else if (belstate == 5) {
    belstate = 6;  //FOOL-PROOF: no more token after state 5
    System.out.println("TOKTYPE 56 -"+toktype);
    break;
    }  
    //while
if (! ruleflag) {
    return belstate;
}
else {
    return belstate = belstate - 5;  //-5 to indicate a belief rule!
}
}  
//beliefautomaton
//beliefautomaton

//****Automaton for plan parsing
//****9 automaton states : initial state=0, accepting state=7, error state=8
//****States 1, 2, 3, 4, 5, 6 are intermediate states
private int planautomaton(String planstr) {

boolean ctxtflag = false;  //context indicator (:)
boolean bodyflag = false;  //body indicator (<--)
boolean boolflag = false;  //boolean indicator for context
int ptoktype = 0, planstate = 0, tokenctr = 0;

StringTokenizer plnstr = new StringTokenizer(planstr, " ");

while (plnstr.hasMoreTokens()) {
    String tokenstr = plnstr.nextToken().toString();
    ptoktype = tokenparser(tokenstr);
    ++tokenctr;
    //System.out.println(tokenctr) ;
    //System.out.println("PTOKTYPE "+ptoktype);
    if (ptoktype < 1 || ptoktype > 16) {
        System.out.println("SYNTAX ERROR: Unrecognised token(s) in Plan Statement !!!");
        break;
    }
    if (planstate == 0) {
        //System.out.println("before planstate : "  + planstate);
        if (ptoktype == OPER && (! ctxtflag || bodyflag))  {
            planstate = 1;
            //System.out.println("PTOKTYPE 1 -"+ptoktype);
        } else if (ptoktype == GOAL && (! ctxtflag || bodyflag))  {
            planstate = 2;
            //System.out.println("PTOKTYPE 2 -"+ptoktype);
        } else if (ptoktype == LTRL && ctxtflag && tokenstr.equals("true") && ! boolflag) {
            planstate = 0;
            boolflag = true;
            else if (ptoktype == LTRL && (ctxtflag || bodyflag))  {
                planstate = 3;
                //System.out.println("PTOKTYPE 3 -"+ptoktype);
            } else {
                planstate = 8;
                //System.out.println("PTOKTYPE 8 -"+ptoktype);
                break;
            }
        }  //if planstate==0
    else if (planstate == 1) {
        //System.out.println("before planstate : "  + planstate);
        if (ptoktype == GOAL) {
            planstate = 2;
            //System.out.println("PTOKTYPE 2 -"+ptoktype);
        } else if (ptoktype == LTRL) {
            planstate = 3;
            //System.out.println("PTOKTYPE 3 -"+ptoktype);
        } else {
            planstate = 8;
            //System.out.println("PTOKTYPE 8 -"+ptoktype);
            break;
        }
    }  //if planstate==1
    else if (planstate == 2) {
        //System.out.println("before planstate : "  + planstate);
        if (ptoktype == LTRL) {
            planstate = 3;
            //System.out.println("PTOKTYPE 3 -"+ptoktype);
        } else {
            planstate = 8;
            //System.out.println("PTOKTYPE 8 -"+ptoktype);
            break;
        }
    }  //if planstate==2
    else if (planstate == 3) {
        //System.out.println("before planstate : "  + planstate);
    }
if (pcktype == LPRN) {
    planstate = 4;
    System.out.println("PCKTYPE 4 -'pcktype':
    
    } else {
    planstate = 5;
    System.out.println("PCKTYPE 5 -'pcktype':
    
    } else if (planstate == 4) {
    System.out.println("before planstate : " - planstate);
    if (pcktype == VAR pcktype == LRL pcktype == CONS) {
        planstate = 5;
        System.out.println("PCKTYPE 5 -'pcktype':
    
    } else if (pcktype == LPRN) {
        planstate = 6;
        System.out.println("PCKTYPE 6 -'pcktype':
    
    } else {
        planstate = 5;
        System.out.println("PCKTYPE 5 -'pcktype':
        
        } else if (planstate == 5) {
    System.out.println("before planstate : " - planstate);
    if (pcktype == STOP) {
        if (pinst.hasMoreTokens) {
            planstate = 6;
            System.out.println("PCKTYPE 6 -'pcktype':
            
            } else {
                planstate = 5;
                System.out.println("PCKTYPE 5 -'pcktype":
                
                } else if (pcktype == TXT) {
                    if (ctxtflag) {
                        planstate = 1;
                        System.out.println("PCKTYPE 1 -'pcktype":
                        
                        } else {
                            planstate = 5;
                            System.out.println("PCKTYPE 5 -'pcktype":
                            
                            } else if (pcktype == CONS) {
                    if (ctxtflag) {
                        planstate = 1;
                        System.out.println("PCKTYPE 1 -'pcktype":
                        
                        } else {
                            planstate = 5;
                            System.out.println("PCKTYPE 5 -'pcktype":
                            
                            } else {  
                            System.out.println("PCKTYPE " -'pcktype":
                            
                            } else if (pinst.hasMoreTokens) {  
                    planstate = 6;
                    System.out.println("PCKTYPE 6 -'pcktype":
                    
                    } else if (pcktype == TXT) {
                        if (ctxtflag) {
                            planstate = 1;
                            System.out.println("PCKTYPE 1 -'pcktype":
                            
                            } else {  
                            planstate = 5;
                            System.out.println("PCKTYPE 5 -'pcktype":
                            
                            } else {
                                System.out.println("PCKTYPE " -'pcktype":
                                
                                } else if (pinst.hasMoreTokens) {  
                        planstate = 6;
                        System.out.println("PCKTYPE 6 -'pcktype":
                        
                        } else if (pcktype == TXT) {
                            System.out.println("PCKTYPE " -'pcktype":
                            
                            } else if (pinst.hasMoreTokens) {  
                        planstate = 6;
                        System.out.println("PCKTYPE 6 -'pcktype":
                        
                        } else if (pcktype == TXT) {
                            System.out.println("PCKTYPE " -'pcktype":
                            
                            } else if (pinst.hasMoreTokens) {  
                        planstate = 6;
                        System.out.println("PCKTYPE 6 -'pcktype":
                        
                        } else if (pcktype == TXT) {
                            System.out.println("PCKTYPE " -'pcktype":
                            
                            } else if (pinst.hasMoreTokens) {  
                        planstate = 6;
                        System.out.println("PCKTYPE 6 -'pcktype":
                        
                        } else if (pcktype == TXT) {
                            System.out.println("PCKTYPE " -'pcktype":
                            
                            } else if (pinst.hasMoreTokens) {  
                        planstate = 6;
                        System.out.println("PCKTYPE 6 -'pcktype":
                        
                        } else if (pcktype == TXT) {
if (! bodyflag) {
    planstate = 0;
    // System.out.println("PTOKTYPE 0 - " + ptoktype);
    bodyflag = true;
} else {
    planstate = 8;
    // System.out.println("PTOKTYPE 8 - " + ptoktype);
    break;
}
else {
    planstate = 6;
    // System.out.println("PTOKTYPE 6 - " + ptoktype);
    break;
}
else if (planstate == 7) {
    // System.out.println("before planstate : " + planstate);
    planstate = 8; // POOL-PCCF: no more token after state 7
    // System.out.println("PTOKTYPE 9 - " + ptoktype);
    break;
}  //while
return planstate;
}  //planautomaton

//***Parsing for the ENTIRE STATEMENT LINE
private int lineparser(String linestr) {
    // final int vardeclare = 3; varrations-domain statement
    final int beliefatom = 1; // belief atom statement
    final int beliefrule = 2; // belief rule statement
    final int plan = 3; // plan statement
    final int stmcerror = 4; // unrecognized erroneous statement
    final int vardomain = 5; // Variable-domains statement
    final int consaxiom = 6; // Constraint axioms
    boolean vardomainflag = false; // variable-domain declaration
    boolean conaxmflag = false; // constraint axioms (firsttoktype == TVAR & secondtoktype == CTXT at state=0)
    boolean beliefflag = false; // belief indicator (LTRL at state=0)
    boolean planflag = false; // plan indicator (OPR at state=0)
    int ststate = 0, firsttoktype = 0, secondtoktype = 0;
    StringTokenizer Intvpe = new StringTokenizer(linestr, " ");
    firsttoktype = tokenparser(Intvpe.nextToken()); // determine type of 1st token
    secondtoktype = tokenparser(Intvpe.nextToken()); // determine type of 2nd token
    System.out.println("1st " + firsttoktype);
    System.out.println("2nd " + secondtoktype);
    // System.out.println("ststate " + ststate);
    if (firsttoktype == TVAR & secondtoktype == CTXT & ststate == 0) {
        vardomainflag = true; // statement is variable-domain declaration
        ststate = varautomaton(linestr);
        System.out.println("VAR-DOM STATE " + ststate);
    } else if (firsttoktype == TVAR & secondtoktype == TVAR & ststate == 0) {
        conaxmflag = true; // statement is constraint axiom
        ststate = axiautomaton(linestr);
        System.out.println("CONSTRAINT AXIOM " + ststate);
    } else if (firsttoktype == LTRL & ststate == 0) {
        beliefflag = true; // statement is a belief
        ststate = beliefautomaton(linestr);
        System.out.println("BELIEF STATE " + ststate);
else if (firsttoktype == OPER && ststate == 0) {
    planflag = true;
    ststate = planautomaton(linestr);
    System.out.println("PLAN STATE "+ststate);
} else {
    return stmterror;
}

if (!beliefflag && !vardomflag && !conaxmflag && !planflag) {
    if (ststate == 4) {
        return vardomain;
    } else {
        return stmterror;
    }
} else if (!beliefflag && !vardomflag && conaxmflag && !planflag) {
    if (ststate == 6) {
        return consaxiom;
    } else {
        return stmterror;
    }
} else if (beliefflag && !vardomflag && conaxmflag && !planflag) {
    if (ststate == 5) {
        return beliefatom;
    } else if (ststate > 5) {
        if ((ststate - 5) == 3) {  //belief rule : correct accepting state
            return beliefrule;
        } else {
            return stmterror;
        }
    } else {
        return stmterror;
    }
} else if (!beliefflag && !vardomflag && !conaxmflag && planflag) {
    if (ststate == 7) {
        return olan;
    } else {
        return stmterror;
    }
} else {
    return stmterror;
}

} //lineparser

//***Splitting tokens in a correct statement into a string array

public String[] tokenarray(String arraystr) {
    StringTokenizer arrstr = new StringTokenizer(arraystr, " ");
    String[] stmttoken = new String[arrstr.countTokens()];
    for (int i = 0; i <= arrstr.countTokens(); i++) {
        stmttoken[i] = arrstr.nextToken();
    }
    return stmttoken;
}

//***Perform Syntax Parsing on statements from .tkn file
//***Generate parsed program file with .psr extension
//***public void parsing(String[] args) {

public void parser (TextArea textsrc, TextArea tkntext) {
    AppletParser syntaxparse = new AppletParser();
StringBuffer tknbuffer = new StringBuffer();
StringBufferInputStream ps = new StringBufferInputStream(textsrc.getText());

try {
    StreamTokenizer filestok = new StreamTokenizer(new DataInputStream(ps));
    filestok.ordinaryChar('.'); // modify char '.' into ord char
    filestok.wordChars('a', 'z');
    filestok.wordChars('0', '9');
    filestok.wordChars('-', '-');
    filestok.wordChars('<', '<');
    filestok.wordChars('&', '&');
    filestok.wordChars('(', ')
    filestok.wordChars(':', ':');
    filestok.wordChars('@', '@');
    filestok.wordChars('_', '_');
    filestok.wordChars('=', '=');
    filestok.wordChars('>', '>');
    filestok.wordChars('[', ']');
    filestok.wordChars('*', '*');
    filestok.whitespaceChars(' ', ' ');
    filestok.whitespaceChars(',', '');
    int token, linenum, tokencnt;
    String linestring = "";
    String [] linearray;
    double tmpnum;
    token = filestok.nextToken();
    tokencnt = 1;
    linenum = filestok.lineno();
    while (token != filestok.TT_EOF) {
        if (tokencnt == 1) {
            switch (token) {
            case filestok.TT_NUMBER:
                //System.out.println("Number: " + filestok.nval);
                tmpnum = filestok.nval;
                linestring = Double.toString(filestok.nval);
                break;
            case filestok.TT_WORD:
                //System.out.println("Word: " + filestok.sval);
                linestring = filestok.sval;
                break;
            }
        } else {
            tmpnum = filestok.nval;
            linestring = linestring + " " + Double.toString(filestok.nval);
        }
    }
}/*1*/

while (token != filestok.TT_EOF) {
    while (filestok.lineno() == linenum && token != filestok.TT_EOF) {
        //System.out.println(token);
        switch (token) {
        case filestok.TT_NUMBER:
            //System.out.println("Number: " + filestok.nval);
            if (tokencnt == 1) {
                tmpnum = filestok.nval;
                linestring = Double.toString(filestok.nval);
            } else {
                tmpnum = filestok.nval;
                linestring = linestring + " " + Double.toString(filestok.nval);
            }
            break;
        case filestok.TT_WORD:
            //System.out.println("Word: " + filestok.sval);
            linestring = filestok.sval;
            break;
        }
    }
} /*2*/
case '-':
    //System.out.println("Default: "+ tokencnt +":"+(char)
    token);
    if (tokencnt == 1) {
        linestring = ""+(char)token;
    } //if
else {
    linestring = linestring+ " "+(char)token;
} //else
break;

default:
    break;
} //switch

token = filestok.nextToken();
tokencnt = tokencnt + 1; //original token count
/*2*/
    } //while 2nd

****** START TO PERFORM SYNTAX PARSING HERE ******
****** CALL METHODS FROM SYNTAXPARSER CLASS ******
int rtnlineparser = syntaxparse.lineparser(linestring);

    //
    //
    //
    //

    if (rtnlineparser == 0) {
        linestring = "0 " + linestring + '
';
tknbuffer.append(linestring);
    }

if (rtnlineparser == 1) {
    linestring = "1 " + linestring + '
';
tknbuffer.append(linestring);
}
else if (rtnlineparser == 2) {
    linestring = "2 " + linestring + '
';
tknbuffer.append(linestring);
}
else if (rtnlineparser == 3) {
    linestring = "3 " + linestring + '
';
tknbuffer.append(linestring);
}
else if (rtnlineparser == 5) {
    linestring = "5 " + linestring + '
';
tknbuffer.append(linestring);
}
else if (rtnlineparser == 6) {
    linestring = "6 " + linestring + '
';
tknbuffer.append(linestring);
}
else {
    tknbuffer.append("Stmt type : "+rtnlineparser + '
');
tknbuffer.append("Line:"+linenum+""+
"SYNTAX ERROR:Unrecognised statement !!!" + '
');
tknbuffer.append(linestring + '
');
    //return;
}

/*+++++++++++++++++++++++++++++++*/
    //** Reinitialise variables
    tokencnt = 1;
    linestring = "";
    linenum = filestok.lineno();
/*1*/
    } //while 1st
} //try
    catch (IOException e) {
        System.err.println(e);
        return;
    } //catch

String tknoutput = tknbuffer.toString();
tkntext.setText(tknoutput);
syntaxparse = null;
    //parser
} //AppletParser
References

[Agha, 1986]  

[Agha, 1988]  

[Anderson, 1996]  

[Arkin, 1990]  

[Ben Ari, 1993]  

[Bratman, 1987a]  

[Bratman, 1987b]  

[Bratman, 1988]  
[Brooks, 1986]

[Brooks, 1991]

[Bruchez, 1996]

[Burmeister, 1992]

[Cohen, 1990]

[Collins, 1997]

[d’Inverno, 1998]

[d’Inverno, 1997]

[Emerson, 1989]

[Epp, 1995]
[Finin, 1994]

[Firby, 1992]

[Georgeff, 1986]

[Georgeff, 1987]

[Georgeff, 1989]

[Heintze, 1992]

[Hewitt, 1977]

[Ingrand, 1992]

[Jaffar, 1986]

[Jaffar, 1992]
[Jaffar, 1994a]

[Jaffar, 1994b]

[Jennings, 1998]

[Kaelbling, 1990]

[Kumar, 1992]

[Lyons, 1992]

[Maes, 1990]

[Marriott, 1998]

[Marriott, 1998a]

[Marriott, 1998b]
[Morley, 1996]

[Müller, 1996]

[Müller, 1996a]

[Muslinder, 1996]

[Ooi, 1999]

[Paull, 1988]

[Pountain, 1995]

[Rao, 1991]

[Rao, 1995]
[Rao, 1996]

[Rossi, 1989]

[Sahota, 1994]

[Sen, 1998]

[Shoham, 1993]

[Thayse, 1988]

[Thomas, 1993]

[Thomas, 1995]

[Tsvetovaty, 1997]

[Woolridge, 1995]
[Wooldridge, 1996]

[Zhang, 1994]