Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Larysa Aharkava

External environment for Soar

Department of Software and Computer Science Education
Bachelor thesis supervisor: Mgr. Cyril Brom
Study field: Informatics

2007

I would like to gratefully acknowledge the supervision of Mgr. Cyril Brom during this work. I thank for the technical discussions to the Victorstar and Delyn D. Watling for his language prompting.

Prohlašuji, že jsem svou bakalářskou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 30.7.2007 Larysa Aharkava

Content

1. Introduction 6
2.Initial theory 8
2.1 What is Soar 8
2.2 How Soar works 8
2.3 Soar and external environments communication facilities 11
2.4 Related works 12
2.4.1 dTank 12
2.4.2 TacAir-Soar 14
2.4.3 Soar Eaters 16
2.4.4 TankSoar 17
2.4.5 Soar Quakebot 18
2.4.6 EES and other environments 19
3. EES architecture 21
3.1 Introduction 21
3.2 Loose coupling conception in EES 22
3.2.1 Kernel, Modules and Objects 22
3.2.2 Kernel and Modules (detailed) 25
3.2.3 Modules and Objects (detailed) 27
3.3 Subsidiary classes 28
3.4 Communication with Soar 28
3.4.1 EES and Soar 29
3.5 Implementation problems during EES creation 29
3.5.1 XML binding 29
3.5.2 XML parsing 30
3.5.3 Acquiring class fields during the runtime 31
3.5.4 Saving EES actual state or why EES has no ‘Save’ and ‘Load’ properties 31
4. EES implementation 33
4.1 Input Data 33
4.2 EES main classes 34
4.3 Main GUI window and EES features 34
4.3.1 EES features 35
4.4 Using Soar debugger with external environment 36
5. Conclusion 37
5.1 Summary 37
5.2 Future work 38
A. Programmer’s guide 39
A.1 Input Data 39
A.1.1 Requirements to the class structure 39
A.1.2 Requirements to the configuration XML file 40
A.1.3 Module and Object configuration files 42
A.2 How to create new classes 42
A.2.1 How to create new module 42
A.2.2 How to create new object 44
A.2.3 How to create new Soar agent 44
A.2.4 How to register new module or object in EES 45

A.3 How to create new environment 46
B. User's guide 47
B.1. EES Main Window 47
B.2. EES features 50
B.2.1 Configuring new environment 50
B.2.2 Changing module's properties during runtime 53
B.2.3 Changing object's properties during runtime 54
B.2.4 Module's runtime vizualization 55
B.2.5 Creating stop condition 56
B.2.6 Selected properties tracing 58
B.2.7 XML Editor 60
B.2.8 Settings 61
Bibliography 63

Thesis title: External environment for Soar
Author: Larysa Aharkava
Department: Department of Software and Computer Science Education
Thesis supervisor: Mgr. Cyril Brom
Supervisor's e-mail address: Brom@ksvi.mff.cuni.cz
Abstract: The aim of that work is to implement external environment for cognitive architecture Soar that can be used for testing Soar agents. That environment should be flexible enough to render research testing for different types of Soar agents. The program should also allow the researcher to combine new environments from already existing objects and environments without any modifications of code. An inseparable part of such an environment is communication between the environmental kernel and the Soar kernel.
The work also contains theoretical principles of Soar, description of some similar programs and comparison with them.

Keywords: artificial intelligence, Soar, external environment

Název práce: Externí prostředí pro Soar
Autor: Larysa Aharkava
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Cyril Brom
e-mail vedoucího: Brom@ksvi.mff.cuni.cz
Abstrakt: Cílem této práce je vyvinout vnější prostředí pro implementaci kognitivní architektury Soar, jenž by umožňovalo testování Soar agentů. Navržené prostředí by mělo vykazovat dostatečnou flexibilitu k testování různorodých Soar agentů. Program by měl uživateli také umožňovat tvorbu kombinací testovacích prostředí z již existujících objektů a modulů. Nedílnou součástí vyvinutého prostředí tvoří možnost jeho komunikace s jádrem Soaru.
Práce se taktéž zabývá nástinem teoretických principů Soaru, deskripcí podobných prostředí a jejich srovnáním.

Klíčová slova: umělá inteligence, Soar, vnější prostředí

Chapter 1

1. Introduction

Artificial Intelligence (AI) is increasingly being used in transportation, the armaments industry, corporative business, interactive computer games and many other branches of science and technology.

The idea of an intelligent machine is old but the thoughts were crystallized in 1950 when Alan Turing wrote his science fiction article «Computing Machinery and Intelligence» (Turing, 1950). In an almost simultaneous publication by Claude E. Shannon's (1950) discussion of how a machine might be programmed to play chess.

Since that time scientist and engineers have been writing more complex and sophisticated AI programs to solve many classes of problems that would give humans intellectual difficulty like solving championship chess problems, transforming one symbolic expression into another by given rules, proving mathematical theorems and many others. Today AI has numerous and very broad applications in such fields as space exploration, the transportation industry, and all phases of manufacture, chemistry, and mathematics. Numerous applications are becoming commonplace in aeronautics, automotive diagnostics, monitoring and automatically adjusting for important variables.

The part of AI research is creating external environments of all kinds where agents[1] can interact. Results of such environment interaction with the agent is usually

[1] Intelligent agent (IA) is a software agent that assists users and will act on their behalf, in performing non-repetitive computer-related tasks. Source: Wikipedie

modification of state of the environment - based on specified parameters, these AI agents then act to modify its certain states.

Some examples of using such environments include computer games (Unreal Tournament, [1]), in some parts of common life simulations (the ENTs project, [2]), and in the air pilot training program that is linked to a simulator (Strategic simulator for military pilots or TacAir-Soar, [3]).

The purpose of this work is the creation of external environmental framework for Soar – general cognitive architecture for developing systems that exhibit intelligent behavior – EES(from External Environment over Soar). That environment should be flexible enough to render research testing for different types of Soar agents(agents that move in 2D worlds or agents that play tic-tac-toe and similar games). The program should also allow the researcher to combine new environments from already existing objects without any modifications of code. An inseparable part of such an environment should be communication between the EES kernel and the Soar kernel.

In first chapter I introduce the problem to the reader. Second chapter describes theoty besides Soar, explains how Soar works, cites several Soar external environments as an example and compares them with my bachelor thesis environment. Ideas behind the architecture and problems I've solved during my working on the software can be found in the Chapter 3. Chapter 4 brings a shallow description of the my environment implementation and Chapter 5 sums up done work and cites as an examples of future work some possible widenings.

Chapter 2

2.Initial theory

2.1 What is Soar

Soar employs a general cognitive architecture for developing systems that exhibit intelligent behavior. Per se, Soar is a computer program for creating agents that stores all available knowledge[2] for use with every task on any given problem that the system encounters.

Traditionally Soar is connected with Allan Newell's name who in 1987 wrote in his book "**Unified Theories of Cognition**"([4]) wherein he defined intelligence as the ability to use ones knowledge to achieve ones goals. Soar was designed to enable access to all knowledge for use on any given problem or set.

It is important that we distinguish notion Unified Theory of Cognition (UTC) and Soar. Normally a UTC must explain how intelligent organisms react to stimuli from the environment, how they exhibit goal-directed behavior, acquire goals rationally, how they represent knowledge (or which symbols they use), and learning. Soar is an architecture[3] – an implementation of a UTC that was developed by Allen Newell, John Laird and Paul Rosenbloom at the University of Michigan.

2.2 How Soar works

[2] Soar architecture represents and uses appropriate forms of knowledge, such as procedural, declarative, episodic, and possibly iconic. Detailed description of the knowledge representation in Soar follows.

[3] Cognitive architecture - an implementation, as a computer program, of the fixed set of behavior-independent mechanisms that define a specific UTC. The result is a programmable system that allows a modeler to specify the knowledge or parameters required for producing behavior. Source: Wikipedie

The design of Soar (Fig. 2.2.1) is based on the hypothesis that all deliberate goal-oriented behavior can be cast as the selection and application of operators to a state. A state is a

representation of the current problem-solving situation; an operator transforms a state (makes changes to the representation); and a goal is a desired outcome of the problem-solving activity. As Soar runs, it is continually trying to apply the current operator and select the next operator, until the goal has been achieved.

Soar has two types of memory – long term memory and short term memory or so-called Working Memory (WM). Soar's working memory consists of a set of <object ^attribute value> elements and short term memories are stored here. Value may describe any sub-object, making hierarchical organization possible (examples can be found in [5]). WM also includes data from sensors like input-link and output-link[4], active goals and active operators.

Soar's long term memory specifies how to respond to different situations in working memory. The Soar agent cannot solve any problem without the addition of long-term memory. Soar represents long-term memory as productions[5]. Each production has a set of conditions and a set of actions. If the conditions of the production match working memory, the production fires and the actions are performed.

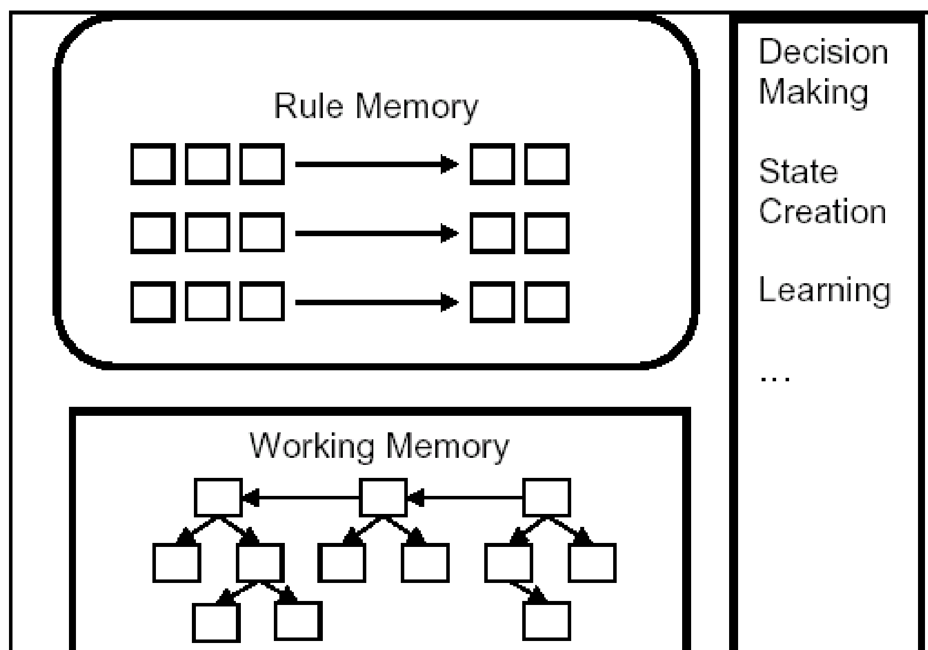


Figure 2.2.1 Soar architecture, [6]

[4] Sensors used for communication with the external environments
[5] Production rule – a representation of knowledge in the following form: `if <condition> then <action>`. The `<condition>` portion of a rule is called the "left-hand side." The `<action>` portion of the rule is called the "right-hand side."

All of Soar’s long-term knowledge is organized like functions of operator selection and application. The knowledge can be of four types:

Knowledge to select an operator:

1. *Operator Proposal*: Knowledge which operator is appropriate in current situation
2. *Operator Comparison*: Knowledge to compare candidate operators.
3. *Operator Selection*: Knowledge to select single operator, bases on the comparisons

Knowledge that applies to an operator:

4. Knowledge of how a specific operator modifies the state.

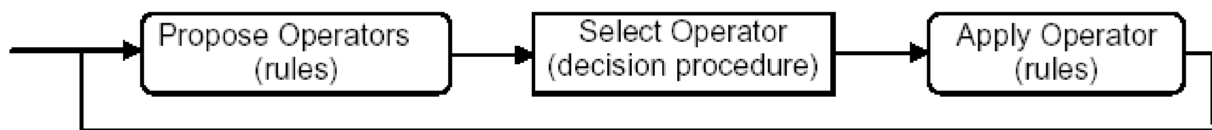


Figure 2.2.2: Soar is continually trying to select and apply operator, [6].

State elaborations indirectly affect operator selection and application by creating new descriptions of the current situation that can cue the selection and application of operators.

These problem-solving functions are the primitives for generating behavior in Soar. They require retrieving long-term knowledge that is relevant to the current situation and function by elaborating the state, proposing candidate operators, comparing the candidates, and applying the operator by modifying the state (Fig. 2.2.2). These functions are driven by the knowledge encoded in a Soar program. Soar represents such knowledge as production rules.

The “if” part of the production is called its conditions and the “then” part of the production is called its actions. When the conditions are met in the current situation as defined by working memory, the production is matched and it will fire, which means that its actions are executed, making changes to working memory. Some productions retract their actions when the conditions are no longer met.

The other function, selecting the current operator, involves making a decision once sufficient knowledge has been retrieved. This is performed by Soar’s decision procedure

that is a fixed procedure that interprets preferences that have been created by the retrieval functions. The knowledge-retrieval and decision-making functions combine to form Soar's decision cycle.

When the knowledge to perform the problem-solving functions is not directly available in productions, Soar is unable to make progress and reaches an impasse. There are three types of possible impasses in Soar:

1. An operator cannot be selected because none are proposed.
2. An operator cannot be selected because multiple operators are proposed and the comparisons are insufficient to determine which one should be selected.
3. An operator has been selected, but there is insufficient knowledge to apply it.

In response to an impasse, the Soar architecture creates a substate. The goal of the substate is to resolve the impasse. During the impasse, operators can be selected and applied to generate or deliberately retrieve the knowledge that was not directly available. For example, during a substate, a Soar program may do a look-ahead search to compare candidate operators if comparison knowledge is not directly available.

2.3 Soar and external environments communication facilities

Soar has its own mechanisms to allow agents to receive input from the environment and effect changes in that environments. These mechanisms provided in Soar are called input functions and output functions or collectively I/O functions:

Input functions add/delete elements to/from working memory in response to changes in the external environment.

Output functions attempt to effect changes in the external environment.

Every Soar's program contain so-called initial structure in its working memory, that creates automatically before agent initialization (Fig. 2.3.1).

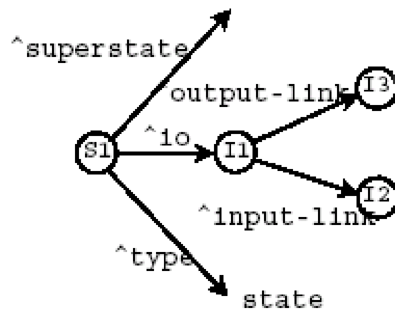


Fig. 2.3.1: Soar's initial structure, [6]

There are two attributes of I/O: *input-link* and *output-link*. The input-link is where an agent's sensory information is available in working memory. The output-link is where action commands must be created for the agent to move in its world.

Also Soar provides special APIs for binding together Soar and some widely distributed languages like Java, C++ or Tcl.

2.4 Related works

2.4.1 dTank

dTank ([7]) is a Java-based simulation game, where a user's tank wages battle with one to many agent-controlled tanks or other users, that was inspired by TankSoar and developed for architectural comparisons of competitive agents and human and agent behavior. dTank offers examination of agents that are built in different architectures as well as humans in the same environment (Fig. 2.4.1). It uses socket communication to provide uniform connections to all models.

Another advantage of dTank is to provide a lightweight alternative to Modular Semi-Automated Forces (ModSAF[6]).

For implementation a client-server architecture and a socket-based interface was chosen. A server is started up using a Java based program. The server displays everything, and runs the simulation. A human user can choose to enter the environment from the server menu, and then the user interface displays only the limited view parallel to agent vision. Agents, models, or humans that connect to the server are all given a tank on the board, and can then send commands to move the tank. The clients are given

[6] The goal of ModSAF is to replicate the outward behavior of simulated units and their component vehicles and weapons systems to a level of realism sufficient training and combat behavior. In essence, ModSAF is a set of software modules and applications used to construct specialized applications.

updates every 2 seconds of what is visible to them on the board. Models receive information and pass commands in the form of text-strings that are converted into state representations conforming to the needs of the architecture.

Models can use this state-representation differently, either acting directly on information as it is made available, or building internal state-representations. Models use this state information, and then decide upon an action that affects the environment. This action is sent to dTank in a formatted string and held in a buffer for time-synchronization purposes. Aside from state information, which is sent at regular intervals, software agents are also informed of important events, such as being hit, 'dying', and 'health-low'. This is to allow modelers some flexibility in the behaviors of models in difficult situations.

To compare different agents, dTank models in several different architectures were created ([8]):

1. *JavaTank*

   Java has no implicit theory of cognition, perception, or action. However, Java agent, JavaTank, takes a small step towards cognitive plausibility by waiting 50 ms before it will consider the environment or implement an action. JavaTank is capable of moving forward, turning, rotating its turret, aiming at a target, and firing.

2. *JessTank*

   Jess (Java Expert Systems Shell) is a Java-based rule engine derived from the CLIPS[7] expert systems shell created and distributed by NASA. The Jess agent, JessTank, also requires 50 ms before it will consider the environment or implement an action. It is only capable of moving forward, turning, rotating its turret in the cardinal directions, and firing. Unlike JavaTank or SoarTank, JessTank does not perform the required trigonometric equations to aim the turret at any enemy that can be seen.

3. *SoarTank*

   The SoarTank model included in the dTank distribution was
   the first model developed. It is capable of moving forward, turning, rotating its turret, aiming at a target, and firing. SoarTank relies on a directed form of

[7] The software CLIPS is a computer program that contains some of the subject-specific knowledge of one or more human experts. CLIPS is an acronym for "C Language Integrated Production System".

wandering that prevents it from merely rotating in place. After it has spotted an enemy, it begins attacking.

Perhaps its greatest is a training environment for cognitive modelers. The environment is simple enough to work with, yet complex enough to provide some challenges. More importantly, dTank provides tools to support the analysis of information gathered in its environment.

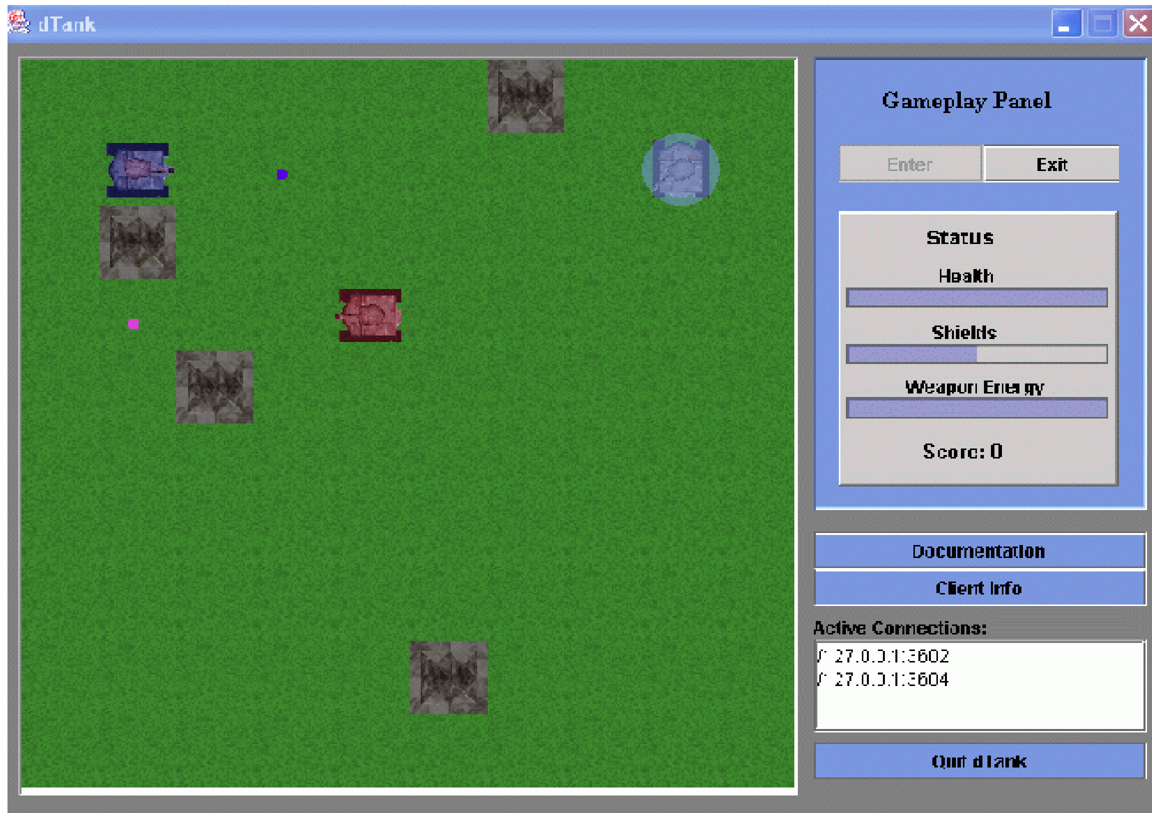


Figure 2.4.1. dTank 2D environment screenshot, [8]

2.4.2 TacAir-Soar

TacAir-Soar (Jones et al., 1999) was developed from Soar using C to provide virtual partners in military training simulations using the Soar architecture with added knowledge about the tactical air-combat domain and interfaces that allow TacAir-Soar pilots to fly in the US military simulation environment and reducing the need for expert human role-players during training exercises.

TacAir Soar is a dynamic real-time simulation of a battlefield. Time in the simulation corresponds to time in the real world, so both humans and computer forces must react in real time. Some aspects of the terrain are also dynamic, such as destructible buildings and bridges. Further, weather servers simulated meteorological features such as

clouds, rain, tides, and daylight, sometimes using live feeds from the area of the world in which the simulation took place.

Communication between TacAir-Soar agents takes the form of message templates. These are fixed messages that resemble the 70 most commonly used conversations between pilots. Future development aims at introducing natural language into the communication.

TacAir-Soar participates in the simulation environment by interfacing with ModSAF. ModSAF includes all the necessary software for interacting over the network with other entities in the simulation as well as simulations of the vehicle dynamics, weapons, and sensors.

TacAir-Soar connects to ModSAF with the Soar-ModSAF interface (SMI). SMI translates simulated sensor and vehicle information into the symbolic representation used by Soar and translates Soar's action representation into ModSAF function calls. One example would be the control and manipulation of a vehicle’s information(speed, direction, weapons, sensors, and radios).

The SMI attempts to organize data so that TacAir-Soar has available the same information a pilot would have, including data from radar and vision sensors as well as data on the status of the aircraft and messages received on its radios. This information is added directly to the working memory of each entity and is updated each cycle of the simulation. For a single entity, there are over 200 input and over 30 different types of output for controlling the plane. Each entity’s access is limited to information from its own sensors, thereby eliminating the possibility of cheating.

The whole environment (i.e. human and automated agents) is bound by distributed interactive simulation (DIS) network as shown on Fig. 2.4.2.

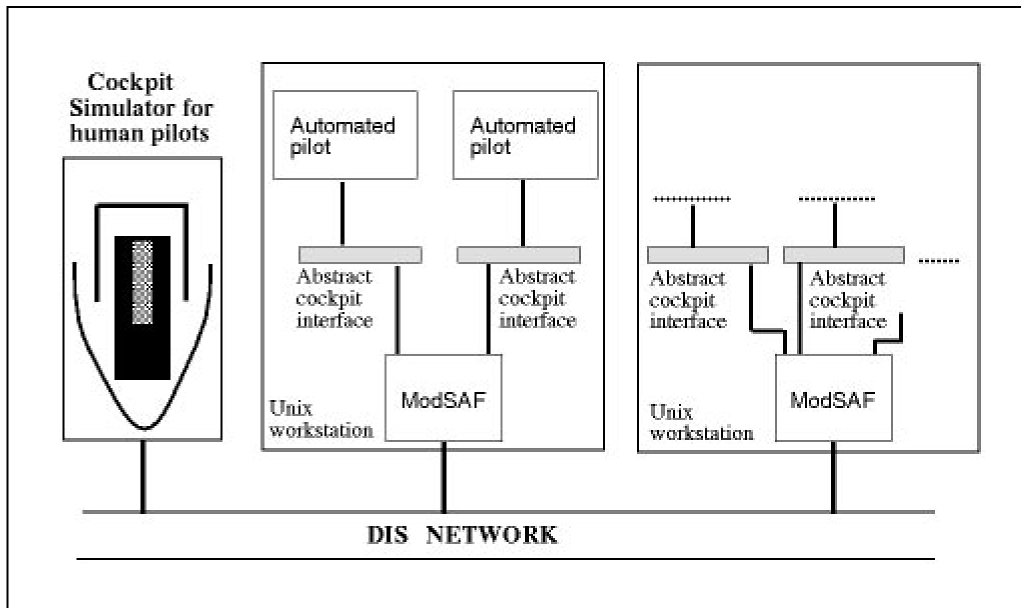


Figure 2.4.2. Human and automated pilots interact with the DIS environment, [10]

2.4.3 Soar Eaters

Eaters is a Pacman-like eaters compete to consume food in a simple grid world. The Eaters world consists of a rectangular grid, 15 squares wide by 15 squares high (Fig. 2.4.3). Walls bound all four sides. Interior wall sections are randomly generated for each new game. No two walls will touch, so there are no corners, except for exterior walls and no "dead ends" anywhere on the board. Each eater starts at a random location. Food pellets are in all other squares of the grid. There are two kinds of food: normal food (blue circles and worth 5 points) and bonus food (red squares and worth 10 points). An eater consumes food by moving into a square. When an eater moves out of a square it will be empty (unless another eater moves into it).

An eater can sense the contents of cells (eater, normal food, bonus food, and empty) up to 2 squares away in all directions. On each turn, an eater can move one square north, south, east, or west. An eater can also jump two squares north, south, east, or west. An eater can jump over a wall or another eater. An eater does not consume any food in the space it jumps over. A jump costs the eater 5 points. Whenever two eaters try to occupy the same cell at the same time, they collide. As a result of the collision, their scores are averaged and they are teleported to new, random locations on the board.

The GUI of the environment was realized in Tcl/Tk, connection between agents and the environment uses Soar I/O links.

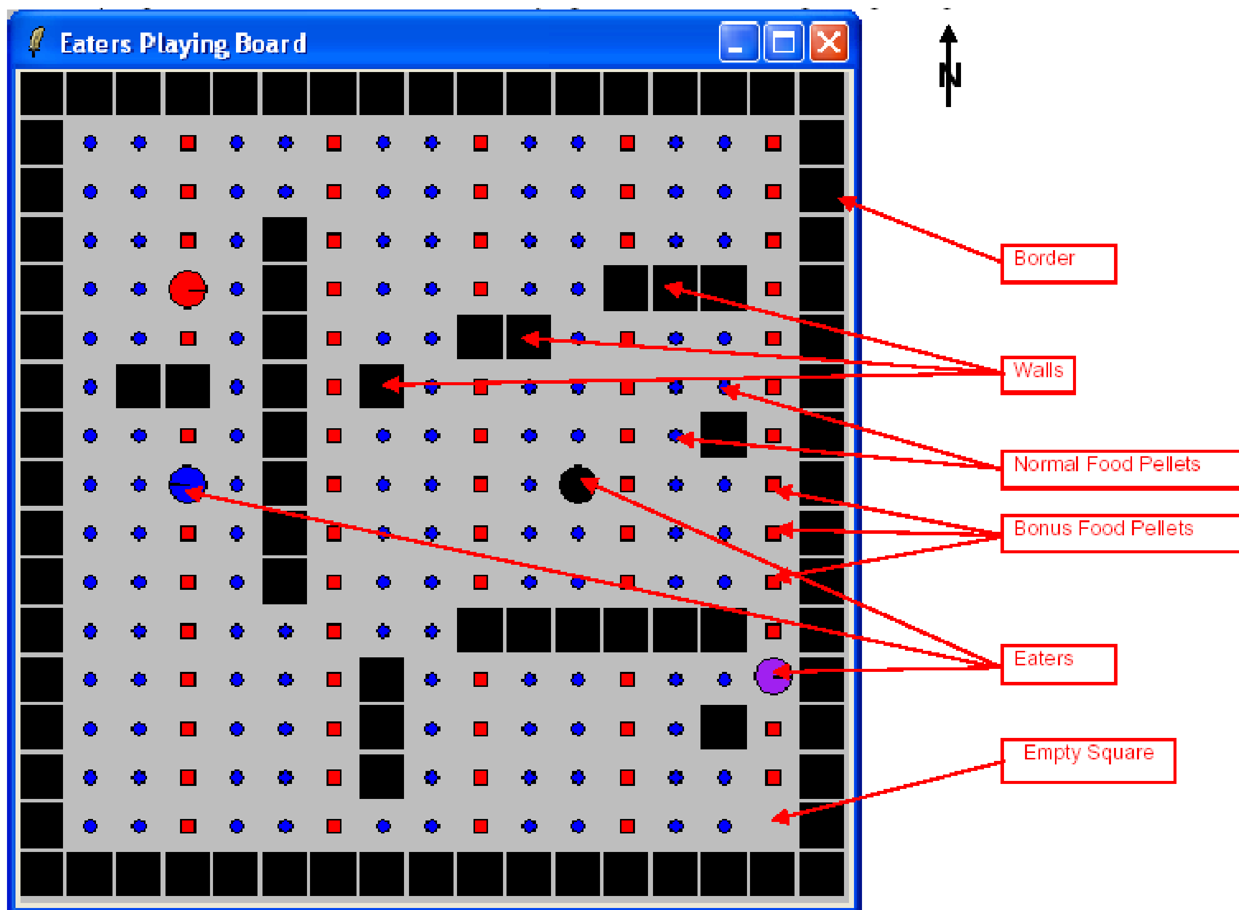


Figure 2.4.3. Eaters Environment, [6]

Futher information can be found in [6].

2.4.4 TankSoar

TankSoar is similar to Eaters in that the Soar program controls a tank in a grid-based world with walls and the same as Eaters can be found in Soar's installation package. However, in the TankSoar game, a tank has many more sensors and many more actions than in the SoarEaters game. The tanks also shoot at one another. These additional actions mean that the programming for TankSoar if much more complicated than for the Eaters game.

The TankSoar game consists of a rectangular grid, 14 squares wide by 14 squares high. All four sides are bounded by walls made of rock. Interior walls are made of trees. There are a variety of maps that can be used, with different layouts of walls (Fig. 2.4.4).

Each TankSoar agent controls one tank. Each tank occupies one square in the playing field. A tank has actions it can take, resources it carries, and sensors with which it observes its environment.

Figure 2.4.4. TankSoar Enviroment, [6]

Futher information can be found in [6].

2.4.5 Soar Quakebot

Soar has also been interfaced to Quake II game by J.E.Laird[8].

In that project Soar bots were located on the separated computer and use socket communication with the game engine through C-code in a DLL. Bots were given the same information to that available to human players (limited vision, no seeing through obstacles, limited hearing, no initial map, etc.) and were meant to behave themselves in similarly way like human players do (e.g., ambush opponents coming out of a room where they reloaded).

The main idea was to enable Quake bot to predict enemies behavior and to take advantage of that. That's why Soar bot used three additional (besides opponents evaluation finctions rules) sets of rules:

[8] Quake II is not the only one popular game that Soar AI architecture was built to – the other one is Descent 3.

1) For deciding when to start predicting
2) For generating expected opponent behavior
3) For proposing how to take advantage of the prediction (e.g. ambush)

Figure 2.4.5. Soar Quakebot, [11]

2.4.6 EES and other environments

All environments listed above are purposed for specific intents (tank field for TankSoar and dTank, air battle field for TacAir-Soar, food field for Eaters). That environments allow the user to achieve more detailed developmental work for the concrete environment (better design, for example). A drawback to the the environments is that modification of any environment is limited by type of the environment. For example, in Eaters one is limited to the creation of a rectangular field and placing here some predefined elements (for Eaters - borders, walls, bonus and normal food pellets, eaters and empty squares).

In contrast to other environments, EES is a framework-like environment. It means that it is not concrete environment but set of tools for creating new environment. Modules[9] can be independent part of the EES. Every module can be stored as xml-file and loaded anytime.

[9] Here module may have two meanings

EES allows to load several modules at the same time and switch between them. EES also allows the user to configure by dint of xml files new modules from already created objects and environmental templates by entering new file properties of the module and it's objects. The results of such configuration could be two versions of the same environment but with absolutely different objects and properties.

Also, unlike other environments, the user can directly participate in EES development by adding new functionality – writing new components of different kinds(modules and objects) that can be used by other users.

---

1) template of the environment, i.e. we may say "module binary tree" or "module 2D plane", but we dont care about objects that it may content or even concrete properties of that module
2) concrete module with defined properties and objects

Chapter 3

3. EES architecture

3.1 Introduction

EES architecture design was suggested under the influence of the program requirements – form followed function.

The most important requirement that distinguish EES from other environments is its flexibility. The main requirement was enable the user configuring new environments in a 'Lego-like' manner – creating new ones from already existing parts. The other requirements are:

1) extensibility – user must be able to extend EES functionality by adding new standardized code
2) universality – user must be able to configure differen types of environments, from 2D environments to binary tree envitonments
3) usability – EES must give the user possibilities for analizyng runtime, vizualizing environments etc.

Using EES the user can configure new environments from existing parts with no change of the code and can add new modules and objects. Simply stated, where the overriding factor in program selection is the need to replace independent component parts of the program, EES is the architecture of choice because it can do this very easily.

Soar gives an opportunity for programmer to use one of three languages for making external tools (like external environment) – Java, C++ and Tcl. Java was chosen as the most corresponding to the problem:

- Java is a cross-platform language and gives more flexibility
- Java naturally supports any Object Oriented Paradigm (that is foundation of EES) and is more intuitive for the programmer
- Java has well-engineered tools like Reflection (that enables dynamic retrieval of classes and data structures by name) or Spring Framework (that provides a lot of functionality including working with XML format and XML data binding[10]) as external project that were extremely useful while creating EES

3.2 Loose coupling conception in EES

3.2.1 Kernel, Modules and Objects

EES holds three levels of abstraction – kernel, modules and objects(Fig. 3.1)

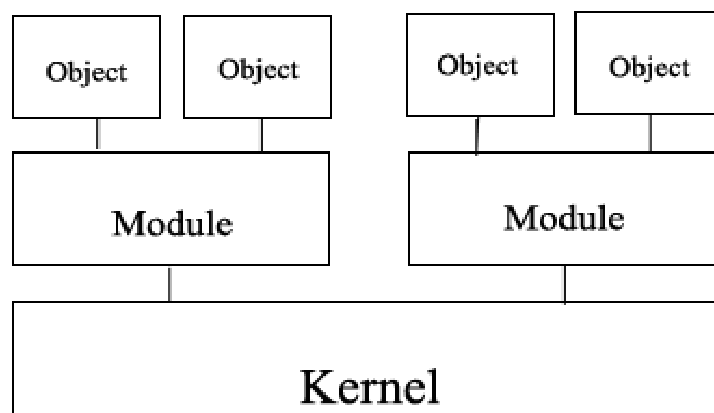


Figure 3.1Three levels of EES

Kernel is responsible for the whole runtime – it registers new modules, reacts on changes(deleting modules from the kernel, adding modules to the kernel, acting as a conduit between modules and the GUI for posting messages to the user and works as 'go-

[10] XML data binding is the binding of XML documents to objects designed especially for the data in those documents.

between' GUI settings and modules). Kernel also houses the dynamic map of modules (where modules can be added or deleted during runtime).

Module is an environmental template that contains properties and methods that must be supported by the objects of that module. Module also holds the dynamic map of registered objects.

Example: Module Plane2D
Properties – gravitation constant g
Methods required from object – move up, move down, move left, move right

Module is just a template until it is registered by kernel. While registering module is initialized with name, properties values and objects.

Example: Module Plane2D can be initialized as module "Earth" (by setting value of g = 9.81) or as module "Moon" (e.g., by setting value of g = 1.62 in XML file). We can register in that modules object "Animal" that support all the methods move up, move down, move left, move right but can't register object "Binary tree searcher" that normally contains search methods but none of the mentioned methods.

Kernel has only initialized modules[11] in its map. Module must have default values of it's parameters and some of them can be changed by users request. One module can be registered several times with different name and parameters – for the Kernel that modules will be different.

Objects – objects are created for concrete modules (like objects "Animal" for module "Earth") and must support properties that are compatible with the 'parent' module. One object can support several modules.

There are two types of objects. 1) Dynamic (runnable, that can 'decide' and modify their state by itself) objects[12] and 2) Static (unrunnable, that cannot decide for itself and

[11] Inicialized module is instanted module class, that contains instanted objects in it's map.
[12] Soar agent is an example of the dynamic object, but dynamic object not necessarily must be Soar – Java object can be dynamic too.

modify the module state[13]. Static module's state can be changed only by module or by a runnable object) objects.

Soar indication can be used to differentiate whether or not an object is Soar. Soar object is recognizable by module only by some boolean flag value (set as *true* if the object is Soar agent and *false* if not a Soar agent).

Example: Lets we have object(as class) "Animal", we may derive from it as object "Cat" (by setting properties 'predator' and 'meow' values as true) as object "Camel"(by setting property 'hump' as true).

Every object must have default values included in its parameters and some of them can be changed by initiation of user's request.

GUI - for greater flexibility of the EES program GUI is also produced as an external class that supports some interface and grants functionality(like vizualization. settings etc.). Isolation of GUI components as separated subsystem allows it's changing without effecting the rest parts of the program. The format of the GUI is not so important – it could be Windows Forms, Web Application or utility for Dos. At the current version (Java Swing windows) GUI interface consists of functions that write out some runtime routine (like soar agent, debug information or some events from modules) and represents current EES state as drawing objects. EES GUI also provides some additional functionality(see later).

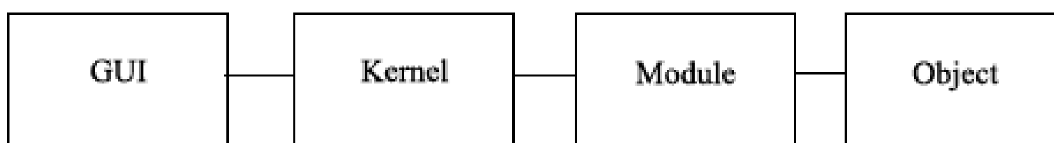


Figure 3.2 Connection between EES components

[13] Soar object can also be static in spite of the fact that there is no point in that.

3.2.2 Kernel and Modules (detailed)

As I mentioned above, Kernel is the 'heart' of the environment. Nevertheless it contains particulat code for manipulating modules only – considerable proportion of the functionality is implemented on the Module-Object level.

For the kernel all modules are the same – it doesn't matter if it is module Plane2D or Binary Tree. For that 'similarity' purpose it is important for modules to support some universal interface that contains all the functions necessary for cooperation of the module and kernel. The most important of functions are:

- Every module must have a pointer to the kernel – for example for using if necessary functions that are rendered by a kernel for working with a GUI. Interactions between the kernel and module are implemented as an Inversion of the Control pattern[14] . Kernel initializes(creates) module and immediately sets that module's 'parent' kernel pointer[15] on itself.
  The EES kernel doesn't work with the module directly, but it takes that module as some universal interface. That module must support all the methods that are listed in that interfaces to interact with the kernel.
  Implementations of like sort help avoid strong binding together between modules and kernel and makes EES more light-weight.

[14] Inversion of Control, also known as IOC, is an important object-oriented programming principle that can be used to reduce coupling inherent in computer programs. More in Wikipedie, [12]
[15] Every module conteins pointer on 'parent' kernel

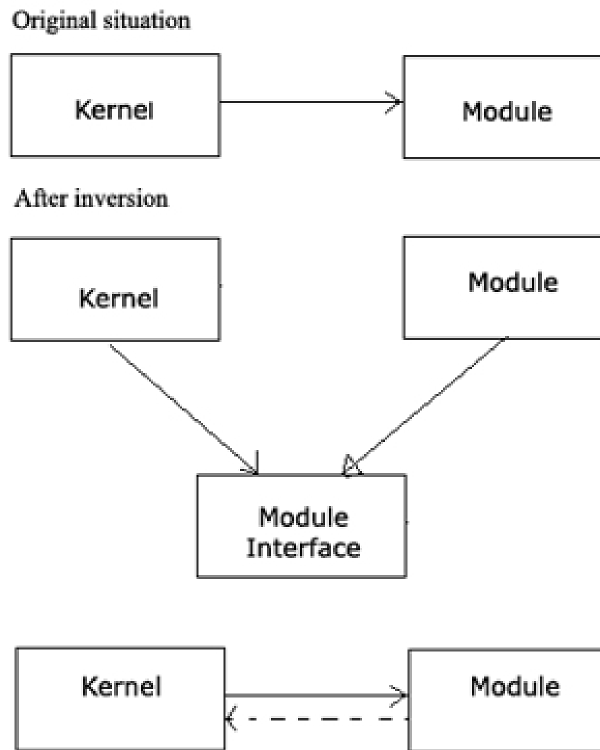


Fig 3.3 Schema of IoC application

- Every module should register objects that are sent to it from the kernel

- Every module should delete objects from it's dynamic map on kernels request[16]

- Every module should print its actual state for users purposes (may be in several forms)

- Every module should commit iterations when in every iteration module allow runnable objects to make one decision.

  Example: We have two objects in module Earth – Bone and Dog. Bone is un-runnable and has set static properties (as coordinates). Dog is runnable and may accomplish a decision cycle. For example, in every step Dog objects changes its

[16] This property is important for situations where several modules are cooperating and can affect each other (like deleting objects from other module). Then, all the communications are housed in kernel. In the current version (ver 1.0) this feature is not implemented but will appear in some of future versions.

coordinates[17] to move closer to the Bone. So iteration in such module is doing one decision step for the Dog (or for more runnable objects if present).

An advanced description of the module must be prepared by the user in special-formatted xml file, where the user indicates module's class, module's name, module's properties (if some) and objects map file (more in Chapter 4 and documentation).

3.2.3 Modules and Objects (detailed)

Module-object level is on a similar level to the kernel-module. In the kernel, modules do not differentiate between objects and the only requirement for any object is to support some pre-defined interface. Also, in the kernel, module consists of a dynamic map of it's objects where objects can be added and deleted during runtime (by the module or by the user).

The module automatically registers objects according to the objects map (also as xml file) where must be indicated every object name, every object class and objects properties (if some).

The module creates all objects from the map[18], examines every object for specific interface support, and uses Inversion of Control (IoC) patterns to register every object for support of the "specific" interface.

Every object residing in the module has to:

- Be able to report to the module if it's runnable or not
- If runnable be able to make a decision cycle by steps (one step – one decision).
- Be able to report to the module if it's Soar agent or not
- Print it's state

The advantages of the above are:

[17] More precisely, an object cannot automatically change its 'external' properties (like coordinates for Earth module). An object may, however, submit a request to change its external properties and the module, if allowable, will then allow object to change its property.
[18] Map is stored as XML file with the objects description

1) independency of single parts from one another – user can change logics of any module or GUI and it will not change the kernel and other parts of the program
2) the possibility of configuring a new module without any source code changes

3.3 Subsidiary classes

EES holds three types of classes:

1) classes that are responsible for the runtime, for example kernel class, GUI class, common module abstract class(that provides realization of most of the module functionality), common object abstract class(that provides realization of some of the object functionality) - so called working classes
2) classes that implement environmental templates (specific modules are created during the runtime)
3) classes that implement objects templates

Expansibility of EES by writing code is realized by adding classes of types 2 and 3.

As runtime classes EES houses three important subsidiary classes that implements basic functionality. These classes are class for common module, class for common object and class for common Soar object. Supposed that many case interfaces will be implemented in some standard way (for example registering object or deleting object in module, connecting to the Soar kernel, creating agent and loading productions in Soar object agent) and that the user wants to implement a unique module or object. The user can use these classes as templates and derive the desired modules/objects from these classes. If desired, further refinements can then be implemented for more specific purpose. More information can be found in tutorial.

3.4 Communication with Soar

In last version of Soar, creators provided a new way for communication between Soar and other languages (like Java). It's called SML (Soar Markup Language) and it was directly used for creation of EES.

In SML, sending and receiving commands are packaged as XML packets. If the user desires to use SML, then the library sml.jar and Java_sml_ClientInterface.dll[19], must be copied to the desired project directory.

Using SML, users can perform all necessary functions from a position external of Soar. These functions include:

- create a Soar kernel and allows for communication directly with kernel
- create a Soar agent and associate that agent with soar productions file
- send information to the input-link
- receive information from the output-link
- run agent for decision, run agent till output, run agent forever, run all agents
- etc

Soar does not make calls to the environment; it simply responds to commands from it. The environment does all the pushing and pulling of io-link WME[20]s.

3.4.1 EES and Soar

EES doesn't need Soar to be installed because it works directly with the Soar kernel (added as .dll and .jar libraries). So EES Soar agents are per se wrappers for the Soar Kernel that send some information to the agent input-link and grab and interpret the information from the output-link.

3.5 Implementation problems during EES creation

3.5.1 XML binding

As written earlier, all the modules and objects are initialized during the runtime and without any need to change code. The main objective was to create objects during the runtime from some text (xml in EES case) file and then have the ability to manipulate the newly created objects. This problem is called data binding. In Java there are many data binding solutions such as JAXB, Castor, Digester and Spring Bean Factory. All of the tools are for restoring objects from the xml to objects in code.

As in EES:

[19] In Java
[20] Working memory elements

1) objects are not created before initializing (the kernel cannot predict the number and quality of the objects)
2) reconfiguration of the xml parser for every input configuration file is inadmissible – the user must not be obliged to change the code manually before
3) initializationcreating new files in like classes and interfaces by a data binding tool is extremely undesirable.

In contrast to other data binding tools, Spring Bean Factory (the part of Spring Framework) is a tool that meets all of the above criteria. In order to use Spring Bean Factory, every module and object must have set and get methods for all the properties (that can be set by beans) that are corresponding to the property names[21].

3.5.2 XML parsing

As previously written, initialization of objects and modules is the result of data binding from a xml file. This data binding is performed by the Spring Framework XML Bean Factory.

In EES, the complete process requires unique objects IDs[22] that are written in xml binding file. IDs are then collected for the creation of objects in XML and Bean Factory must locate the specific ID for any object in the file. There are two ways that Bean Factory uses to locate an ID from the file – SAX[23] parsing and DOM[24] parsing.

SAX and DOM were both designed to allow programmers to access information stored as xml without having to write a parser. Java was designed with and uses SAX and DOM via its JAXP library to implement these two methods.

SAX parser is event based. SAX works like a scanner in that it takes fields that the user wants. It's quicker then DOM parsing but it stores nothing in memory.

DOM parser creates a tree structure of the document and stores this information in memory. It is not as fast as SAX but allows the user to work with the whole structure at once.

Because of its speed, SAX was chosen as the data binding parser.

[21] For example for property 'name' class must have methods setName and getName
[22] If using EES tool, unique IDs are gerenated automatically
[23] Simple API for XML
[24] Document Object Model

3.5.3 Acquiring class fields during the runtime

Existing EES GUI has the possibility to represent a current module's state as an object tree that contains the object's names, the objects properties and these properties values. While this is all good, there still was the need for a universal code that could represent a module's state without the need to be re-written with the addition of a new class. This need was met by Reflection.

Reflection is a process by which a computer program can be modified in the process of being executed - thus dynamic modification. Restated another way, reflection allows an object to have information about the structure of the project (classes, fields and the field's values that project contains) and this information is accessible during its runtime. Java has Reflection API[25] as a part of it's programming.

3.5.4 Saving EES actual state or why EES has no 'Save' and 'Load' properties

Initially I planned to implement 'Save' and 'Load' options for the EES that would be implemented as making copy of the saved module and saving that copy on disk with the possibility of following loading .

Java provides two ways of how to copy(for saving purposes) its objects – shallow and deep copying(or cloning in Java terms). Shallow clone of the object is easy to implement but it provides copying of the objects' shell only – so when one of the objects(parent or clone) manipulates with its inner state, the inner state of the another object changes too.

Deep cloning can be implemented by cloning module's objects recursively(idea similar to DFS) and the problem occurs when I try to copy Soar object(Agent, Kernel, Identifier etc.). Java has no native mechanism for copying foreign classes like these ones(i.e., they are not serializable) so the other possibility is to clone that objects using reflection by copying actual fields values manually. Nevertheless that method fails too. As I found out, Soar's Java classes are wrappers around underlying C++ objects so saving and recreating the Java objects (or cloning them) wouldn't trigger the C++ objects to be copied or recreated. Soar isn't directly implemented in Java, it's actually all in C++ with a Java interface.

[25] When program source code is compiled, information about the structure of the program is normally lost when lower level code (typically assembly language code) is produced. If a system supports Reflection, the structure is preserved as metadata with the emitted code.

Something like saving were implemented in TacAir-Soar using special methods, but Soar's authors didn't come with a general solution that can be used.

Chapter 4

4. EES implementation

4.1 Input Data

EES uses configuration XML files to input data that describes modules and objects. This process is carried out during runtime when all objects are created using these files and then the Kernel processes them.

Configuration XML files can be divided into two types:

1) module configuration file
2) module map or objects configuration file

Module configuration files have their own peculiar properties. Every module configuration file must contain:

1) only one element(co called 'bean') with ID value "module"
2) a property 'objectMap' where the user sets the path in the XML configuration file with objects
3) a property 'name' that is necessary for the kernel to use when processing modules (sending to the modules map and receiving from the modules map).

Module map configuration file contains list of objects for the current module. Every object initialization must contain:

1) unique ID
2) object's instantiation class (including package, e.g., com.larrr.project.objects.soarwrappers.SoarRandomMover)
3) a property 'name' that is necessary for the module to use when processing objects

Names of modules and objects (in the context of the same module) must be unique.

4.2 EES main classes

EES contains several classes essential for the environment[26]:

1) Kernel class (EESKernel.java class) – class that implements environmental kernel
2) Common module template (CommonModulee.java class) – class that implements most of the module routines. The goal of that class is to minimize programmer's work while adding new functionality.
3) Common object and Common Soar templates(CommonObject.java and CommonSoar.java classes) – classes that implement most of the object(or Soar object) rutines.
4) GUI class (GUI.java class) – responds for the communication between user and EES.

More information about these classes can be found in the Programmer's guide (Appendix A).

4.3 Main GUI window and EES features

As was mentioned in chapter three, EES GUI is not a direct part of the program – it is implemented as an external module. As an external module, it can be easily be interchanged for another GUI.

Current EES GUI is implemented as Java Swing GUI application (Fig 4.3.1).

[26] All these classes can be found directly in com.larrr.project package.

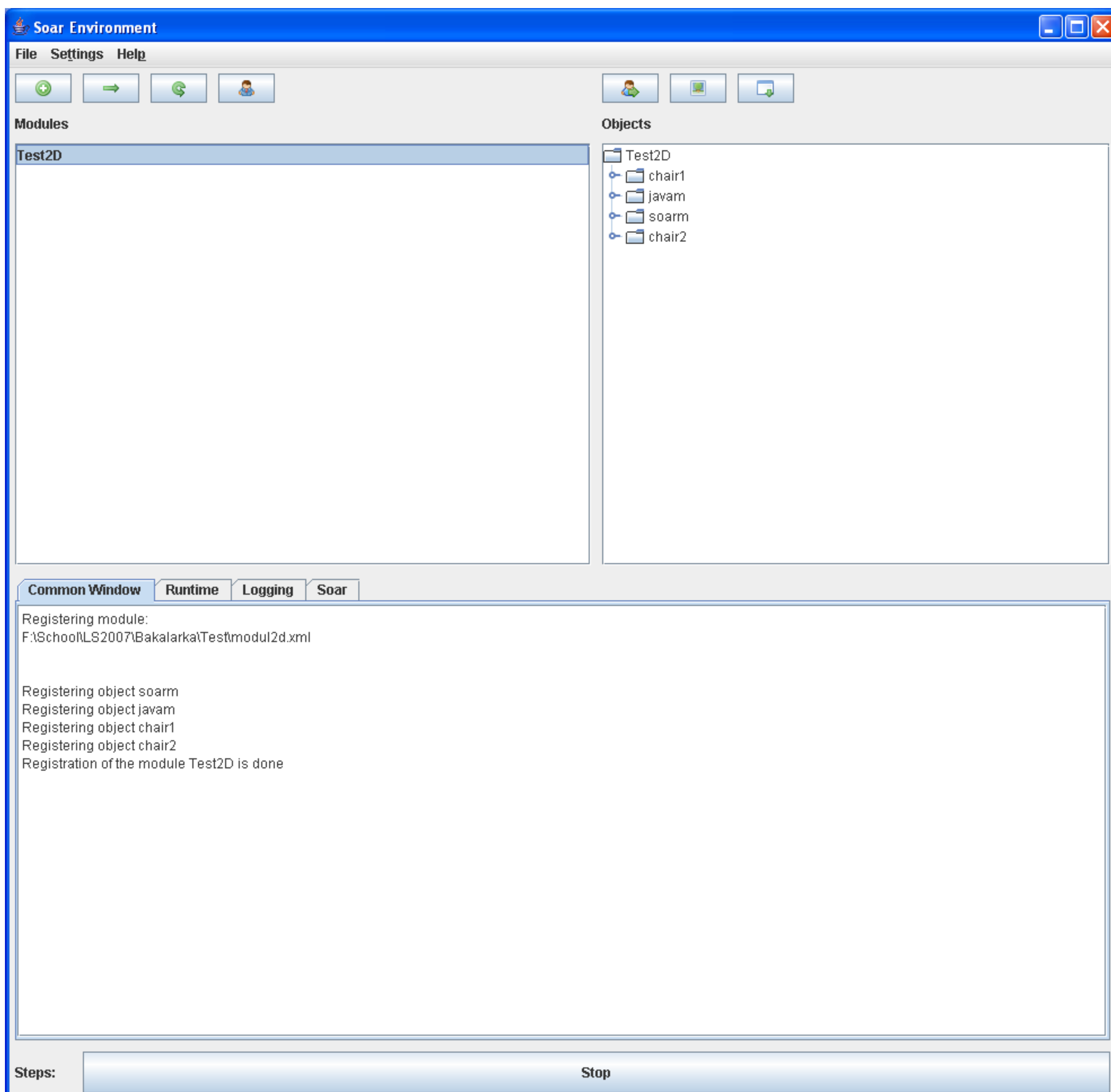


Figure 4.3.1. EES GUI

EES GUI allows the user to manipulate the program by adding new modules, switching between already added modules, selecting modules, perform iteration within the chosen module, manipulating modules , creating stop conditions, generating new modules and using some other settings.

4.3.1 EES features

EES allows user to[27]:

[27] See user documentation for details

1) Load environments from the XML configure files. EES allows simultaneous working with several modules and renders possible to switch

2) Visualize module objects map as tree

3) Delete module from the kernel during runtime

4) Delete objects from the module during runtime

5) Edit objects values during runtime

6) Configure new environments from the already registered modules and objects

7) Do iterations(one or several) in selected module

8) Visualize progress in selected module

9) Track selected module and its inner objects parameters

10) Edit XML configures (using Xerlin XML editor, [13])

11) Create debug conditions – so called 'Stop conditions'

More information about the features can be found in User's guide (Appendix B).

4.4 Using Soar debugger with external environment

The new version of Soar Debugger (SDB) is implemented in Java. It has control over breakpoints and allows the user wide functionality - in SDB user can debug agents, examine their working memory step by step, and observe which operators were proposed, selected and applied in several modes. The interacting window(part of SDB GUI) allows user to trace the execution of Soar at many different levels such as decisions, phases, production firings, working memory changes. Debugger also has additional windows that display common structures, such as the current state, the current operator, etc.

SDB is very useful for debugging the interaction of a Soar agent with an external environment. SBD uses special tools for connecting the remote Soar to port 12121. For this purpose, the user can trace his Java environment (program) in debug mode (that also uses port 12121) and after creating a Soar kernel program in Java, SDB connects with that kernel and executes the necessary changes for debugging.

SBD can be used as external tool from the Eclipse IDE and being delivered with the Soar software.

Chapter 5

5. Conclusion

5.1 Summary

The aim of this research paper was to investigate how to create an external environment for cognitive architecture Soar that in terms of configuring new modules is easily distensible and that does not require the writing of new code.

Upon investigating the EES architecture it was found that EES is based on loose coupling architecture. This loose coupling allows the user to separate different parts of the program, and to manipulate any it without affecting other parts of the program.

The results of this investigation also show that it is possible to divide EES in two levels of abstraction: kernel-module and module-object.

Another important requirement was that the programs commands would be easy and intuitive for the user. Upon investigating this aspect it was found that EES allows the user to configure modules while observing a short list of simple rules.

A very important finding was that it is possible to manipulate with EES during runtime easily using a GUI interface.

To sum up the results of this investigation are that the EES program matches the requirement of a universal source code that requires modification only for cases were new functions are added.

5.2 Future work

Current version of EES can be widen in several meanings: functional(adding new functionality to the EES) and user(making GUI more user friendly).

Functional widening:

- Adding modules cooperation: in current version every module loaded to kernel is a separated unit that has no possibility co communicate and with other modules. That widening needs modification of the EES kernel code.
- Adding possibility of the simultaneous running of the several modules for the purpose of comparing them.
- Adding statistics that will allow to summarize module’s work or compare several modules after making iterations
- Adding more modules and objects templates

Appendix A

A. Programmer's guide

A.1 Input Data

EES uses configuration XML files to input data that describes modules and objects. This process is carried out during runtime when all objects are created using these files and then the Kernel processes them.

As previously noted, EES uses the part of Spring Framework for data binding – XML Bean Factory. This factory creates objects during the runtime from the predefined XML-file, and stores it in the so-called Bean Container that instigates, configures, and manages a number of created objects[28]. Bean Container has some requirements like

- Requirements to the class structure from which object is created
- Requirement to the configuration of a XML file

Data binding with XML Bean Factory involves two steps: 1) creating all beans from the configuration file and storing these files in the container 2) getting objects from the container by ID and processing them.

For the user's comfort EES contains utility that ease creation of the input XML files[29].

A.1.1 Requirements to the class structure

Class must be an actual implementation of bean that is described in the definition. That means that class must contain all the properties that are described in the XML descriptions. Unlike XML Bean Factory, EES does not use complex pointing for creating sub-objects from other objects. When creating any new object or module class it is enough for the writer to obey the so-called bean class structure. In bean class structure,

[28] Objects created with XML Bean Factory from configuration XML file are called 'beans'
[29] Details can be found in user documentation

every class that is instigated from the bean must be setter-based. This means that every property in class must have a set method with a name that corresponds to that property name[30]. Although not required by XML Bean Factory, when data binding, the 'intelligent programmer' should implement to every set method a corresponding get method. In bean, the writer is not required to describe all the properties but any property may be required by the kernel or module. This is why it is very important to set default property values that may be reset while bean is created.

Example: Assume EES contains class Chair. Every object that is instigated from that class has properties 'height' and 'typeOfWood'. Then the class Chair must look like that [31]:

```
Class Chair
{

  //default values are important!
 private int height = 70;
 private String typeOfWood = "Red Wood";
 private String name = "some chair";

/* draw attention on corresponding between property name and
method name */
public void setHeigth(int height){…}
public int getHeigth(){…}
public void setTypeOfWood(String wood){…}
public String getTypeOfWood(){…}
public void setName(int name){…}
public String getName(){…}

//other methods
}
```

A.1.2 Requirements to the configuration XML file

Configuration XML file must have a special format[32]. For EES purposes that format has rather stringent restrictions but its scope is more than ample.

---

[30] Uppercase first letter of the priperty name and add set or get to the beginning

[31] That example is very rough – of course user may use his own units (like meters,kilometers for int value). But module and object must use the same units, of course(if you has object's height in meters , don't forget about that when you are programming module and don't use kilometers or centimeters).

[32] Viz http://www.springframework.org/dtd/spring-beans.dtd

1. For validation purposes, user must indicate where the Data Type Definition is placed. This is usually at the top of the file.

For users that are connected to the internet it is enough to validate their configuration file online. For this the user must add at the beginning of the file following lines**:**

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
```

Otherwise user needs to download that file from the internet, save it on the local disc and then load validator from that file:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"file:///X:\somepath\spring-beans.dtd">
```

2. Every configuration file must start and end with <beans> tag. Every bean that is described must start and end with <bean> tag.

3. Every bean has one or more IDs (also called identifiers, or names). These IDs must be unique within the XML Bean Factory where the bean is hosted. In EES a bean will always have only one ID.

Every bean in the configuration file must have the following fields:

- At bean: id and class as attributes
- At property: property name (the same name as in class) as attribute, value as subitem

4. Every configuration file must start and end with <beans> tag. Every bean that is described in must start and end with <bean> tag.

Example: For the aforesaid class Chair, bean configuration may looks like:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "file:///D:\spring-
beans.dtd">

<beans>
  <bean id="chair" class="com.larrr.project.modules.ObjectChair">
  <property name="height">
       <value>100</value>
  </property>
  <property name="typeOfWood">
       <value>oak</value>
  </property>
  </bean>
</beans>
```

During the process of creating bean, types of properties are recognized by XML Bean. This cancels the need for writing these properties in the configuration file. But in EES user will need “beans”, “bean”, “property” and “value” tags and it’s attributes only.

As noted earlier, XML Bean Factory has a broad range of uses.

A.1.3 Module and Object configuration files

Module configuration files have their own peculiar properties. Every module configuration file must contain:

4) only one bean with ID value “module”
5) a property ‘objectMap’ where the user sets the path in the XML configuration file with the objects
6) a property ‘name’ that is necessary for the kernel to use when processing modules (sending to the modules map and receiving from the modules map).

Example: Typical module configuration file looks like that:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "file:///D:\spring-
beans.dtd">

<beans>
  <bean id="module" class="com.larrr.project.modules.ModuleBinaryTree">
  <property name="objectMap">
       <value>D:\Rocnikac_add\Config\ObjectsBinaryTree.xml</value>
  </property>
  <property name="name">
       <value>BinaryTreeTest</value>
  </property>
  </bean>
</beans>
```

Object configuration file must also contain a name field.

Names of modules and objects (in the context of the same module) must be unique.

A.2 How to create new classes

A.2.1 How to create new module

1. EES consists interface IKernelModule[33] that is important for communication between the Kernel and every module. For communications with its objects, every module uses its own unique interface. Most methods of the interface communication (like setting and getting name and registering objects are usually implemented in the same way. EES uses the abstract class 'CommonModulee' that implements the 'IKerneModule' interface and executes all that routines.

2. 'CommonModulee' class consists of the abstract method 'printMyself'. Common Module must be implemented in a unique way for every module. For example the print state for module 'Plane2D' and the print state for module 'BinaryTree' are absolutely different. Print state for Plane2D must have similar to co-ordinates and prints objects as they are found in respect to these coordinates. Binary Tree is different and requires something similar to a tree-structure and prints dependencies (parent-child scenario) between objects.

3. For loose coupling, every module has a unique interface that it implements. For example, interface IModuleBinaryTree for BinaryTree). When calling its module, objects do not interact directly with the module class, but with the interface of that module. This configuration is not obligatory, but is very desirable because writing the interface of the module before the module is created delineates clear requirements to the module while maintaining the loose coupling conception of EES.

Noting the requirements above, the procedure of creating any new module is as follows:

1) Creation of the interface of the module – that interface must consist of all the methods that are significant to any future objects of that module.

2) Creation the class of the module. That class must implement the already created interface for the level module-object (module, that specify functions for working with objects but never for working with the kernel) – that's 'the job' of the IKernelModule interface) and interface IKernelModule. For simplification, it is recommended to extend new module class from CommonModule class. CommonModulee class implements all the methods of the IKernelModule

[33] For more details see the programmer documentation

interface except ‘printMyself’ which that must be implemented by the user. User can redefine some of the CommonModule methods.

A.2.2 How to create new object

Creating new object is similar to creating new module:

1. EES includes the interface IModuleObject that every object must implement.

2. EES includes the abstract class ‘CommonObject’ that implements IModuleObject interface for execution of all routines. That class is not as functional as ‘CommonModule’ class – it names and retrieves the name of the objects and sets some other important parameters such as soar and runnable flag.

3. ‘CommonObject’ class consists of the abstract methods ‘printMyState’ and ‘runMe’. The ‘printMyState’ method is identical to that found in the module, is specific for every object and must be implemented by the user while taking into consideration that object’s particular qualities. The ‘runMe’ method must be implemented in objects that are ‘runnable’. The function of the ‘runMe’ method is making one decisional iteration. For example with the runnable object “Dog” on the “Plane2D” module it can be the decisional iteration on where to move (up, down, left or right).

4. For loose coupling, every object has an interface that it implements (for example interface IObjectBinaryTreeNode for BinaryTreeNode). While retrieving objects from its object map, module works with that interface and all objects of that type look the same. This point is also not obligatory, but **very** desirable.

A.2.3 How to create new Soar agent

EES doesn’t differentiate between Soar object and normal object written in Java. The only way how to determine the difference is examination by the getSoar() method. The results of getSoar will return a ‘true’ value for Soar object and a ‘false’ value for otherwise.

For Soar objects, EES consists of the class ‘CommonSoar’ that extends the class ‘CommonObject’ (for more information see programmer’s documentation). User may extend from that class or use the ‘CommonObject’ class for extension.

Information concerning the communicating with Soar kernel is found in “SML Quick Start Guide” or in examples that are the part of EES and programmers documentation.

A.2.4 How to register new module or object in EES

After creating new EES item(module or object), the one has to register it for the purpose of subsequent usage.

New module must be registered in modules.xml configuration file, that can be found in ./config folder. In order to register the module, programmer must add to the configuration module description, i.e.,:

1) module class
2) module parameters

in special format.

Example: Module 2D can be registered in the following way

```
<module class="com.larrr.project.modules.Module2D">

  <param name="name"></param>
  <param name="objectMap"></param>
  <param name="maxX"></param>
  <param name="maxY"></param>

</module>
```

Draw attention on the fact that properties are given as attributes, not values.

Object must be registered in the same way in objects.xml configuration file.

Example:

```
<object class="com.larrr.project.objects.object2d.StaticObject">

      <param name="name"></param>
      <param name="type"></param>
      <param name="x"></param>
      <param name="y"></param>
      <param name="icon"></param>

</object>
```

A.3 How to create new environment

After writing or adding new code for module and objects for the module the last step is to configuration – writing XML configuration files. A detailed explanation on how to write XML configuration files is found in 1.1. For the reader's convenience, a quick check list of several of the most important rules for successful configuring are listed below.

Quick check list of important rules

- Every object's must have default 'runnable' and 'soar' values set as false. Don't forget to change them if there is another type of object. Take into account that soar objects are ***usually*** also runnable.
- Every object (in the context of the module) and every module (in the context of the kernel) must have a unique name.
- In the configuration file every object must have a unique ID and that ID must be the same as found in the "module"
- Every module must contain absolute path to the XML file with objects description in property "objectMap"

Concrete example of the EES widening can be found in tutorial.

Appendix B

B. User's guide

B.1. EES Main Window

Current EES GUI is implemented as Java Swing GUI application (Fig. B.1).

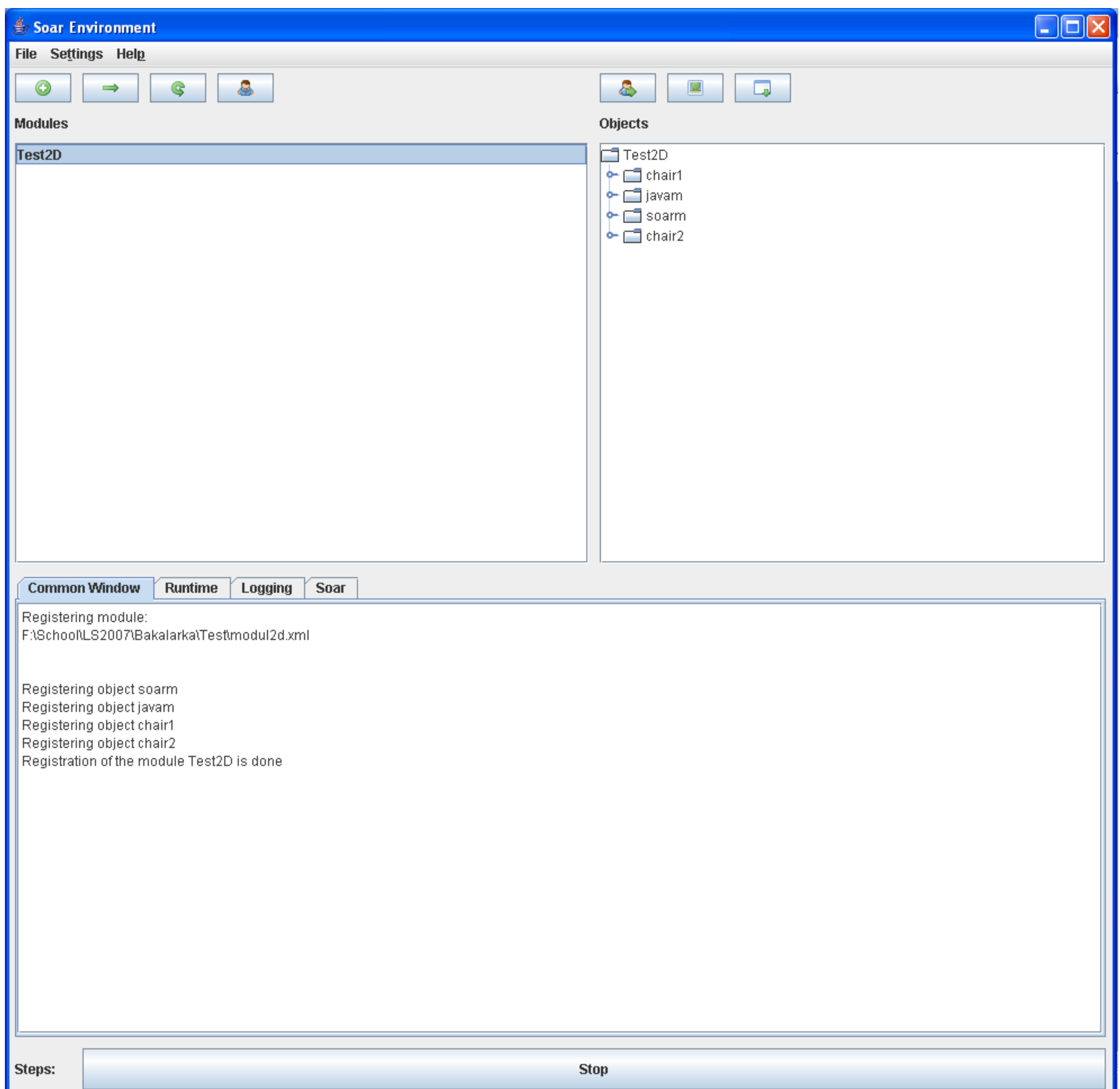
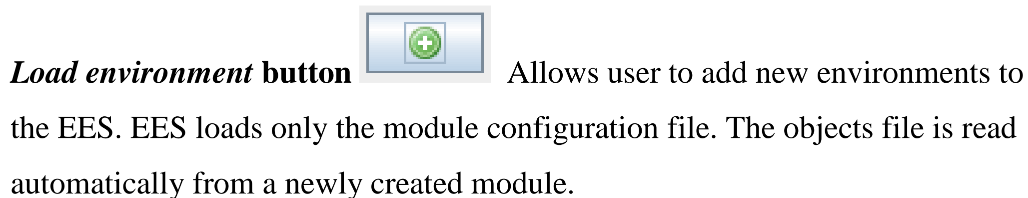


Figure B.1. EES GUI

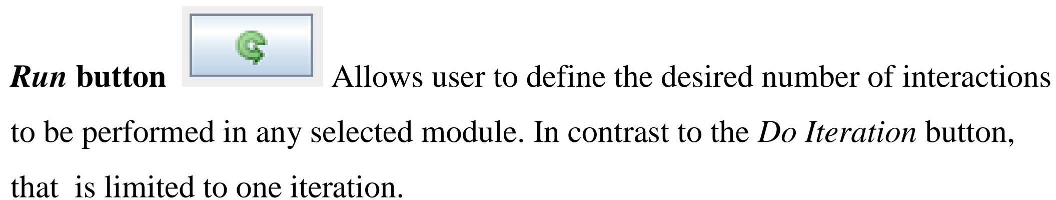
EES GUI allows the user to manipulate the program by adding new modules, switching between already added modules, selecting modules, perform iteration within the chosen module, manipulating modules , creating stop conditions, generating new modules and using some other settings.

For these manipulations GUI contains the following elements:

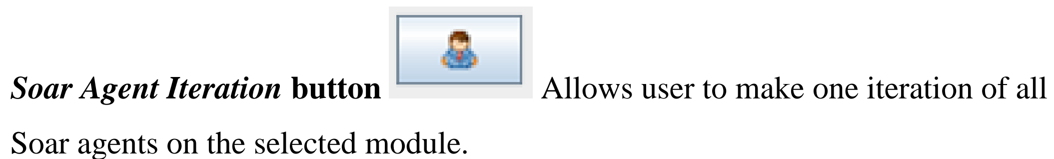


***Load environment* button** Allows user to add new environments to the EES. EES loads only the module configuration file. The objects file is read automatically from a newly created module.

***Do Iteration* button** Allows user to make one iteration on the selected module.



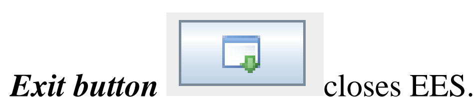
***Run* button** Allows user to define the desired number of interactions to be performed in any selected module. In contrast to the *Do Iteration* button, that is limited to one iteration.



***Soar Agent Iteration* button** Allows user to make one iteration of all Soar agents on the selected module.

***Run Soar* button** Similar to Run button, but iterations are made with Soar objects only.

***Visualize module* button** Prints selected module in some graphic mode.



Exit button closes EES.

Common window Prints all information and errors that are relative to loading new modules and objects.

The state of every running objects iteration is printed to the ***Runtime window***.

Modules window Shows all the modules that are loaded to the kernel and allows to the user to delete modules during the runtime.

Objects window When a module is selected it appears as a tree-representation on the module map. User can also delete objects from the selected module map during runtime (Fig. B.2).

Menu gains access to some other EES features.

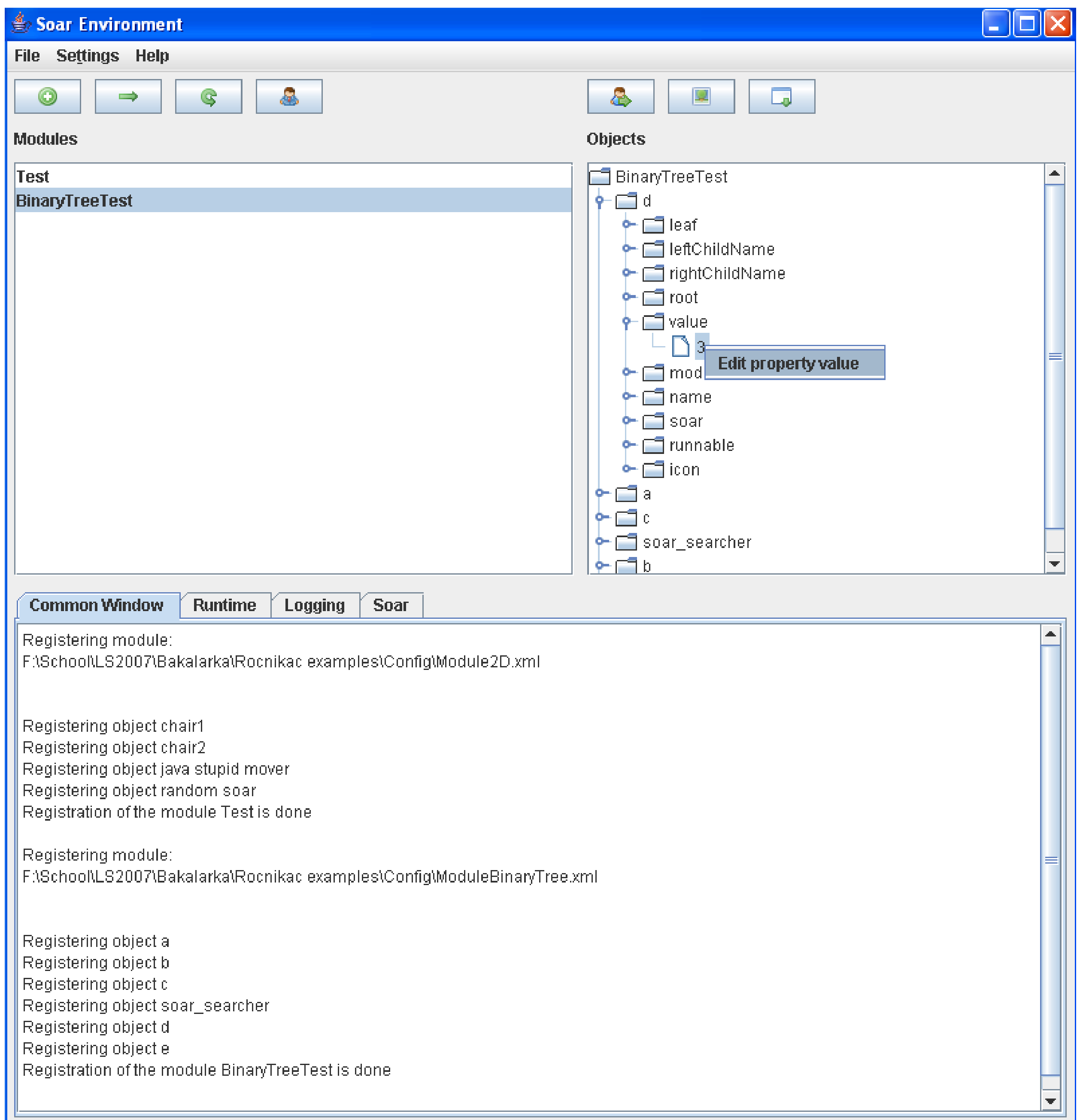


Figure B.2. EES runtime.

B.2. EES features

B.2.1 Configuring new environment

For configuring new environment user need to define two environmental constituents:

1) module – initialization of module properties, i.e., module's name, module's objects map(as absolute path to XML file)

2) module objects map – initializing of module objects and it's properties

There are two ways how to create that configure: manually or using EES configure tool. Manual method is rather labour-intensive and fraught with lots of errors, so I'd advise usage of the tool.

Step 1. Call configuring tool from the menu (Fig. B.2.1.1)

Figure B.2.1.1

Step 2. Select module type you want to configure (Fig. B.2.1.2)

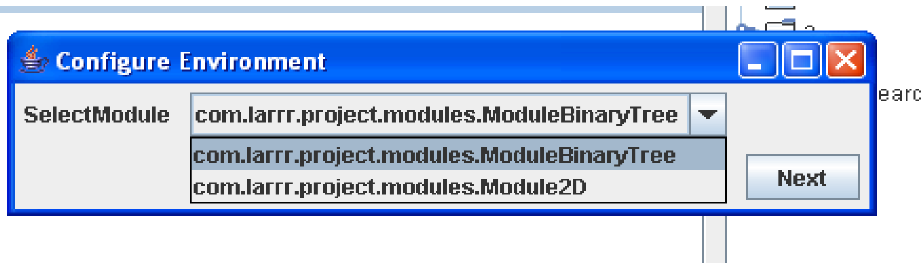


Figure B.2.1.2

Step 3. Enter module properties. 'Name' and 'Object Map' are required fields (Fig. B.2.1.3).

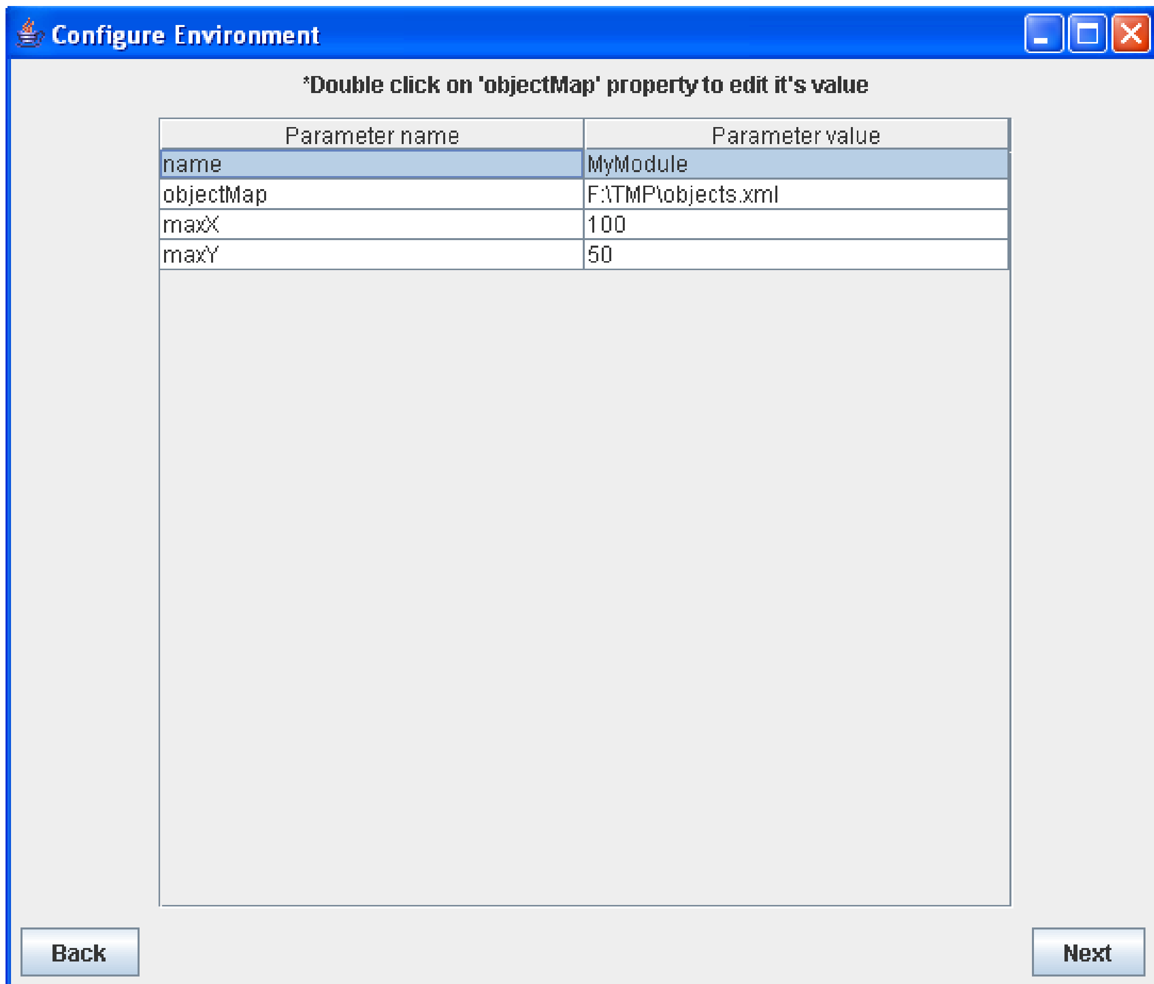


Figure B.2.1.3

After clicking 'Next' button you will be asked to save module file.

Step 4. Select objects you want to configure for that module. Remember that objects must support certain interface, defined for your module (Fig. B.2.1.4). Every object may be used several times.

Example: Every object added to Module2D module must support interface IObject2D.

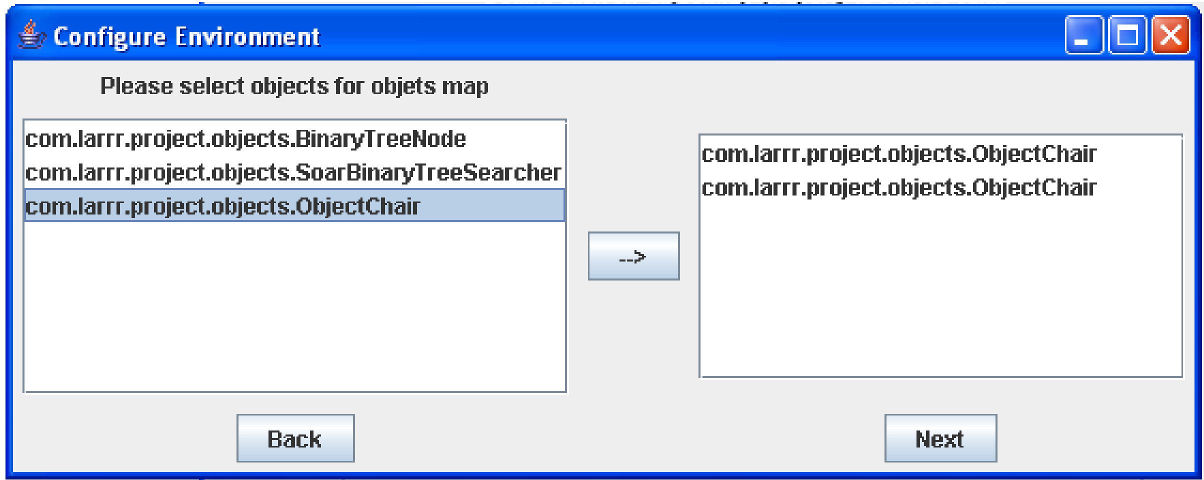


Fig. B.2.1.4

Step 5. Set objects values. 'Name' field is required and must be unique (Fig. B.2.1.5).

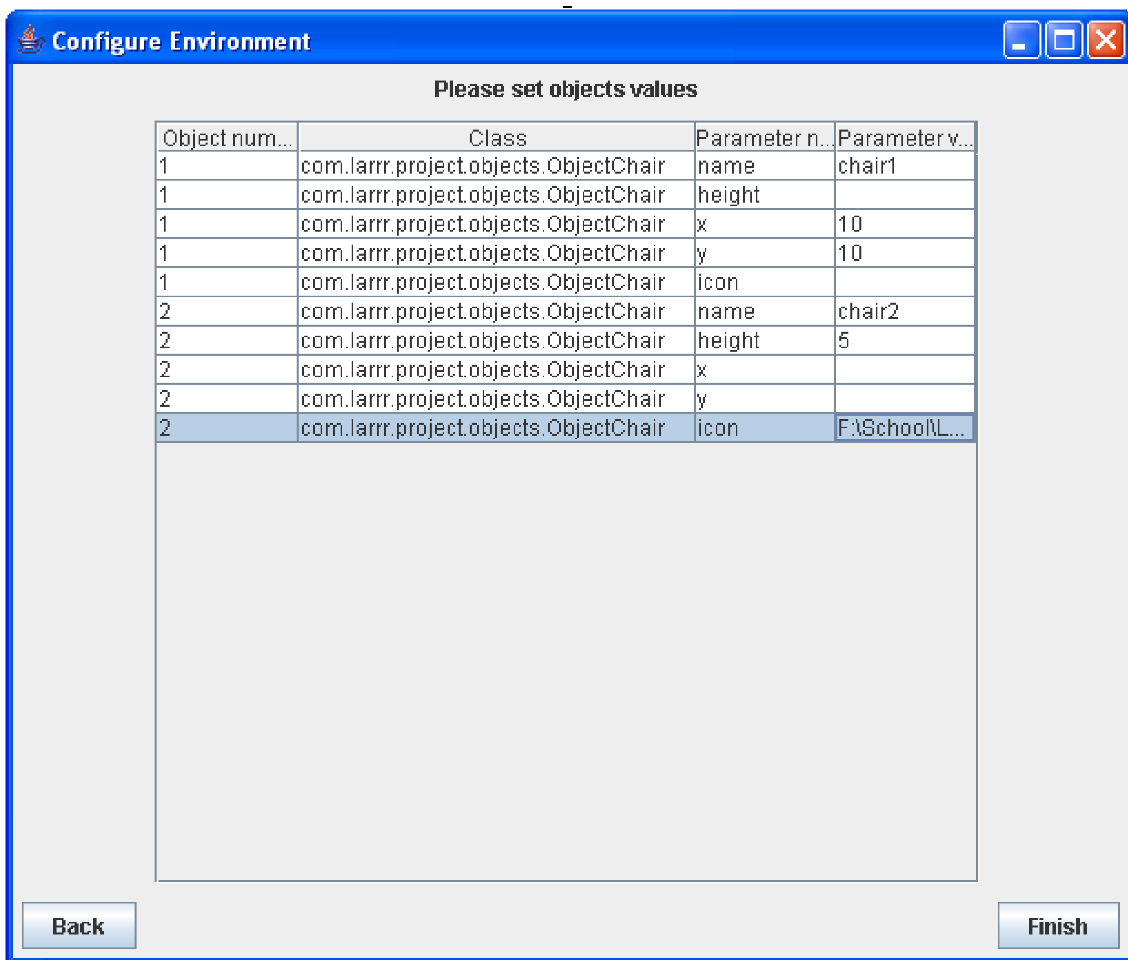


Fig. B.2.1.5

After that step you can load your module.

NB[34]: Before configuring new environments make sure you have DTD file(spring-beans.dtd) set. That file is used for configuring and further evaluation of the XML files and need to be set in order to configure new environments. DTD file can be set using EES menu: **Menu -> Settings -> DTDSettings**.

B.2.2 Changing module's properties during runtime

Module properties can't be changed after creation, but user can change objects map by deleting objects during runtime (Fig. B.2.2.1).

[34] Nota Bene is a Latin phrase meaning "Note Well".

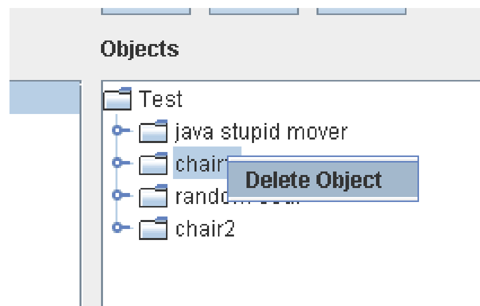


Figure B.2.2.1

Also user can delete modules from the kernel during runtime (Fig. B.2.2.2).

Figure B.2.2.2

B.2.3 Changing object's properties during runtime

You can change objects values during runtime by right mouse click and entering new value (Fig. B.2.3.1). Changed object must be:

1) one of the primitive type (String, Boolean, Character, Long, Integer, Short. Byte, Double, Float)

2) non-Soar

3) must have setProperty and getProperty to the changed property (because only that kind of properties can be restored from the XML definition of the environment)

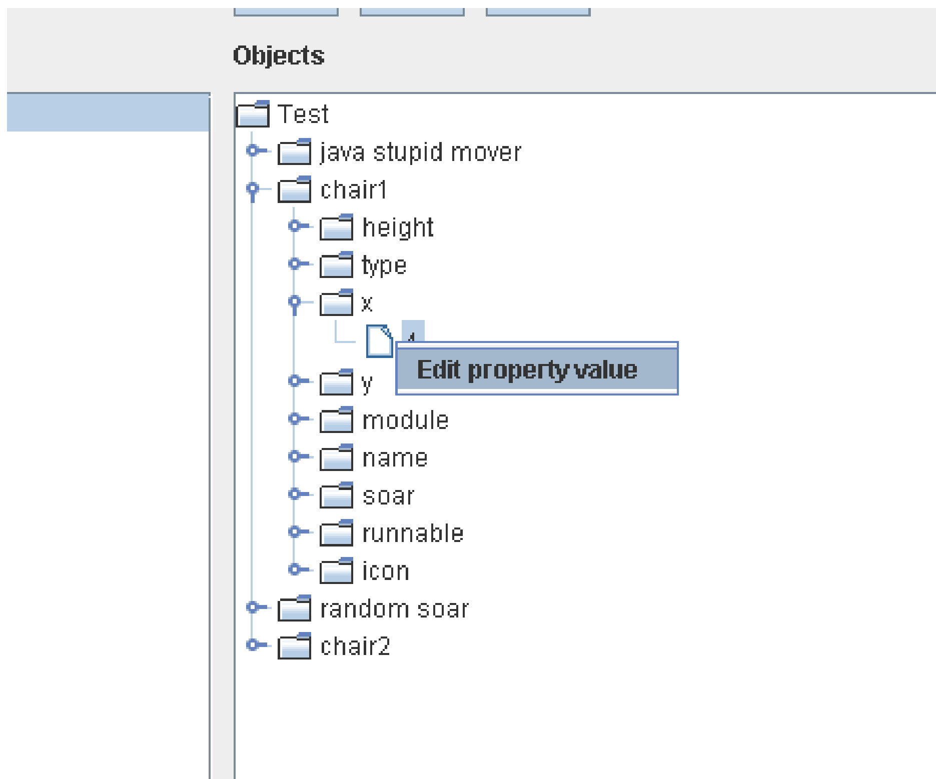


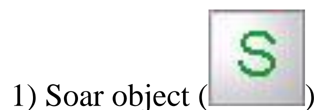
Figure. B.2.3.1

B.2.4 Module's runtime visualization

Because EES makes possible to configure a lot of module types(e.g., binary tree, plane 2D etc.), there is hardly possible to make common runtime visualizer and every module must implement its own visualization tool. Nevertheless EES has it's own preprepared 2D visualizer (Fig. B.2.4.1).

2D visualizer

Visualizer defines three pictures by default:

1) Soar object ()

2) Non-Soar object ( )

3) Empty field ( )

User can change default picture by redefinition of the “icon” property in the XML configure objects file.

Visualizer is called by pressing *Visualize module* button**.**

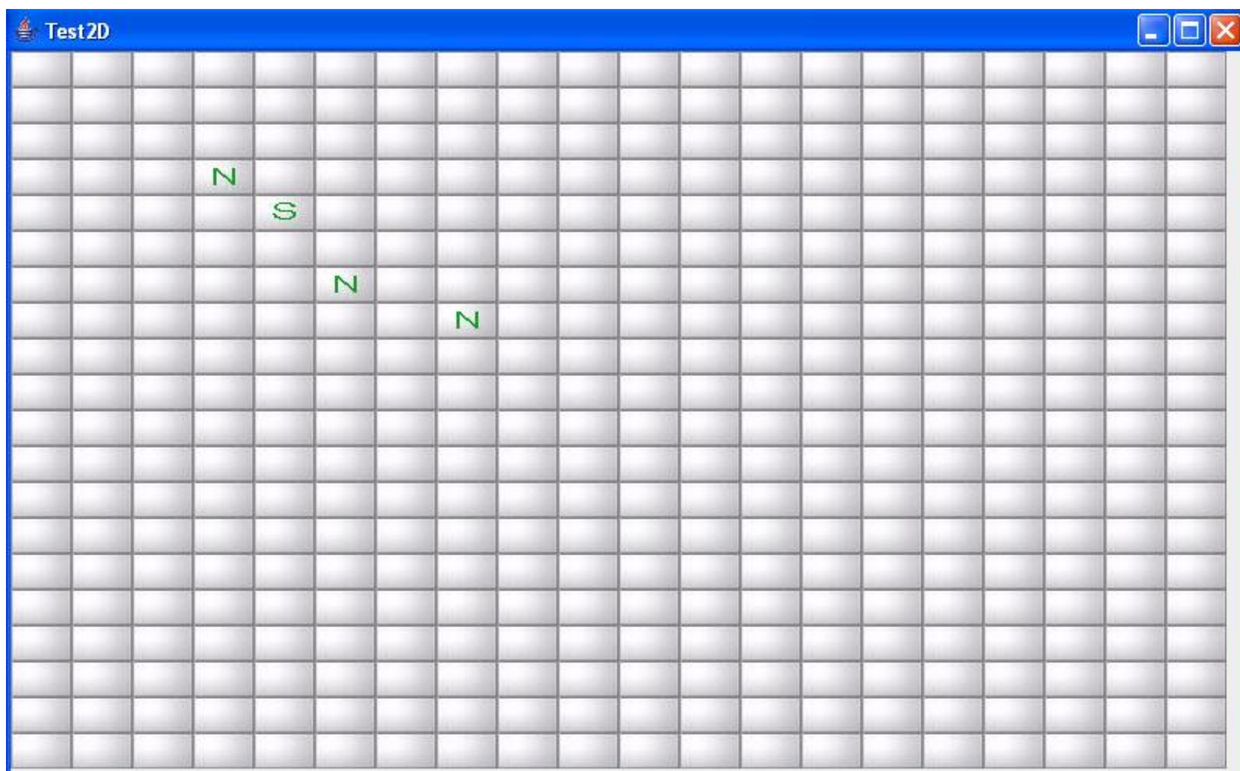


Figure B.2.4.1. Example of the default 2d EES visualization.

B.2.5 Creating stop condition

EES allows user to define so-called stop condition – set of properties to stop iterations when achieved.

Step 1. Select module and select “Create/Edit stop condition” popup menu item (Fig. B.2.5.1)

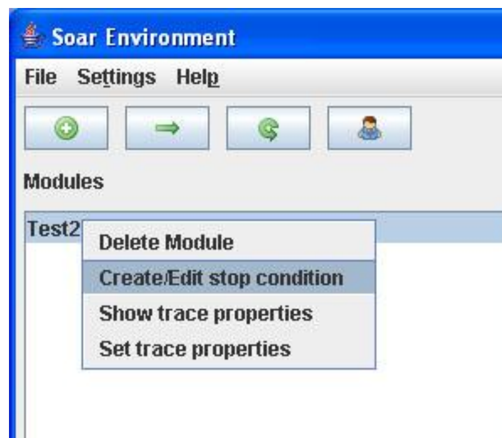
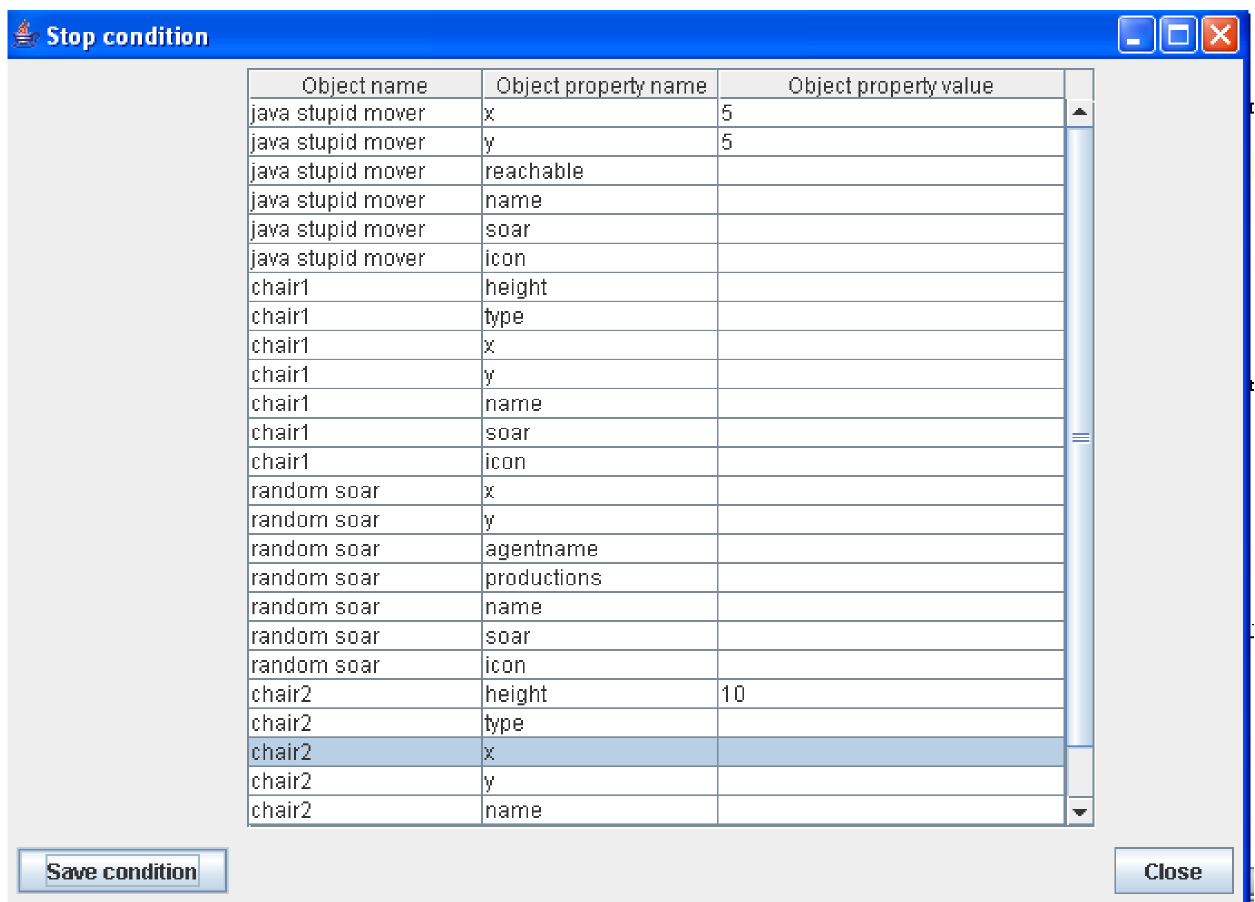


Figure. B.2.5.1

Step 2. Set stop condition values and press "Save condition" button (Fig. B.2.5.2).



Object name	Object property name	Object property value
java stupid mover	x	5
java stupid mover	y	5
java stupid mover	reachable	
java stupid mover	name	
java stupid mover	soar	
java stupid mover	icon	
chair1	height	
chair1	type	
chair1	x	
chair1	y	
chair1	name	
chair1	soar	
chair1	icon	
random soar	x	
random soar	y	
random soar	agentname	
random soar	productions	
random soar	name	
random soar	soar	
random soar	icon	
chair2	height	10
chair2	type	
chair2	x	
chair2	y	
chair2	name	

Figure B.2.5.2

When stop condition is achieved, EES informs the user by showing simple Message box:

Figure B.2.5.3

B.2.6 Selected properties tracing

EES allows tracing objects properties changes in the Objects window (Fig. B.2.6.1), but sometimes that way is not comfortable enough(e.g. when module has large objects tree). If that's the case the user can use EES tracing tool.

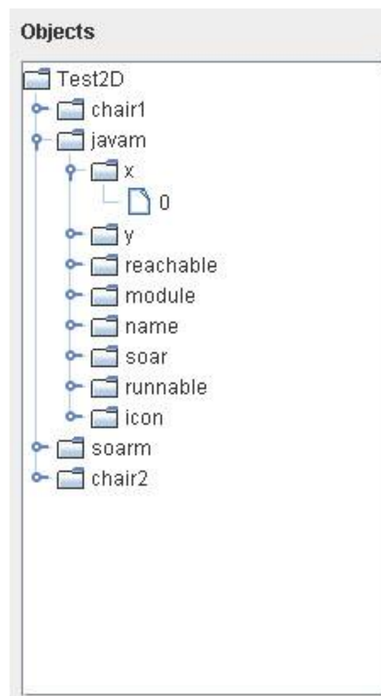


Figure 2.6.1. Objects window

Step 1. Select the module you want to create tracing for and then select “Set trace properties” popup menu item (Fig. B.2.6.2).

Figure B.2.6.2

Step 2. Select properties you want to trace and press "Save" button (Fig. B.2.6.3).

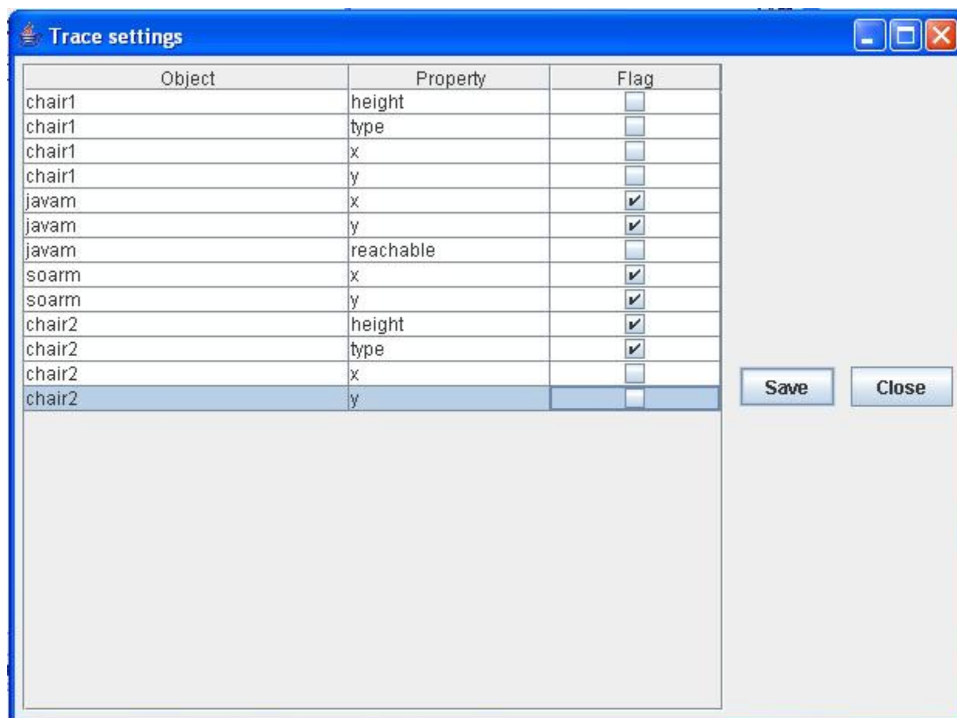


Figure 2.6.3

Step 3. Call tracing window by selecting "Show trace properties" popup menu item (Fig. B.2.6.4).

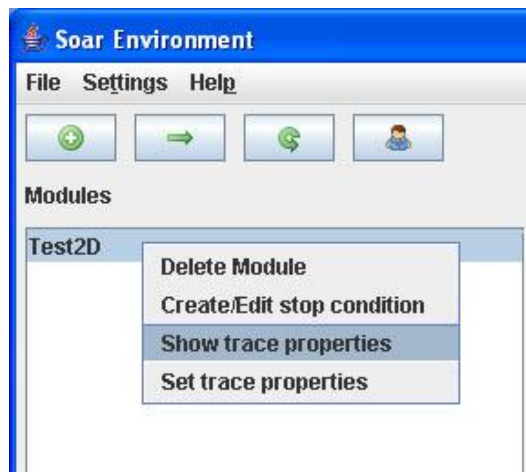


Figure B.2.6.4

Trace

Object	Property	Value
javam	x	0
javam	y	0
soarm	x	7
soarm	y	3
chair2	height	1
chair2	type	none

Close

Figure B.2.6.5

When trace window is opened you can follow changes of the objects properties in every step.

B.2.7 XML Editor

EES contains its own built-in XML editor that makes editing XML configure files more handy. For editing your file call the editor by calling **Menu -> Settings -> Edit XML file** menu item, select file you want to edit and after editing it press "Save" button (Fig. B.2.7.1).

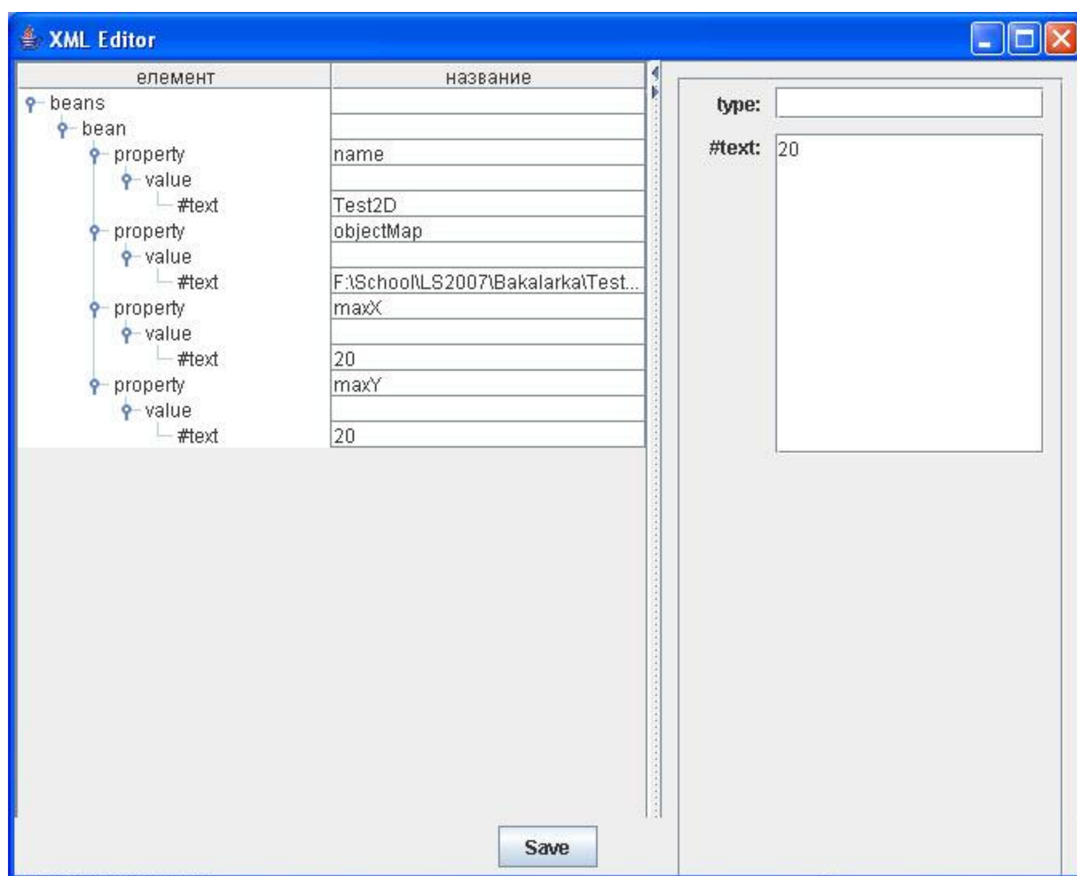


Figure B.2.7.1

B.2.8 Settings

EES contains additional settings that allow save runtime EES information to the file (Fig. B.2.8.1):

1) Saving logs
2) Saving traces (runtime)

The last one setting makes visualization and avaliable tracing during multi-iteration process, i.e. that user can see that process changes on the visualizer and in the trace window.

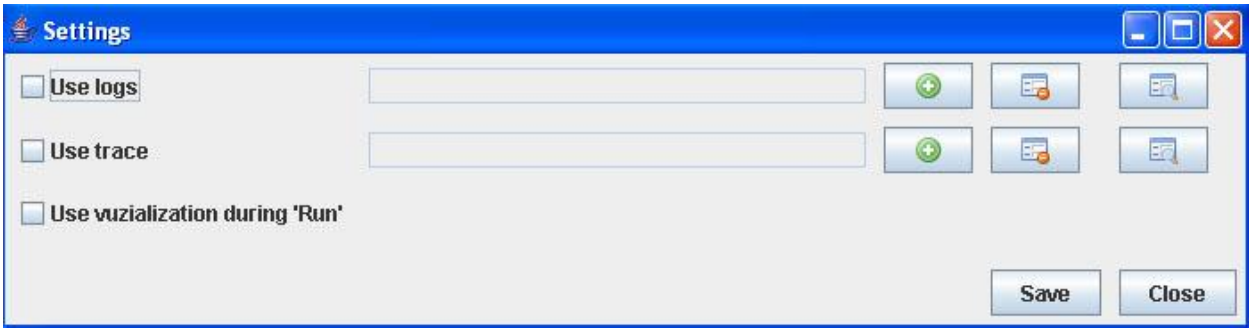


Figure B.2.8.1

Bibliography

[1] Unofficial Guide to: Unreal Tournament, http://guidesarchive.ign.com

[2] MFF UK, Prague: *Project Ents*, http://ufal.mff.cuni.cz/~bojar/enti/

[3] Laird, J. E., Coulter, K. J., Jones, R. M., Kenny, P. G., Koss, F. V., Nielsen, P. E.: *"Integrating intelligent computer generated forces in distributed simulation: TacAir-Soar in STOW-97."*, Proceedings of the 1998 Simulation Interoperability Workshop, 1998

[4] Newell, A.: *Unified Theories of Cognition*, Harvard University Press, 1990

[5] Soar homepage, http://sitemaker.umich.edu/soar/home

[6] Laird, J. E.: *The Soar 8 Tutorial*, University of Michigan, 2006

[7] dTank homepage: http://ritter.ist.psu.edu/dTank/

[8] Morgan, G. P., Ritter F. E., Stevenson W. E., Schenck I. N.: *dTank: An Environment for Architectural Comparisons of Competitive Agents,* The Pennsylvania State University, 2005

[9] CLIPS: A Tool for Building Expert Systems, http://www.ghg.net/clips/CLIPS.html

[10] Wallace, S. A., Laird J. E.: *Behavior Bounding: Toward Effective Comparisons of Agents & Humans,* University of Michigan

[11] International game developers association: http://www.igda.org/

[12] Wikipedie: http://en.wikipedia.org/wiki/Main_Page

[13] Xerlin homepage: http://www.xerlin.org/

[14] Java homepage: http://java.sun.com/