Development of a Scripting Language Interpreter for Acquisition of Expert Knowledge in a Regular Way

Tadevos Baghdasaryan
Laboratory of Cognitive Algorithms and Models, IPIA, Academy of Science of Armenia
tbs_@mail.ru

Arthur Grigoryan
Laboratory of Cognitive Algorithms and Models, IPIA, Academy of Science of Armenia, Yerevan Telecommunication Research Institute
grigoryan.arthur@gmail.com

Zaven Naghashyan
State Engineering University of Armenia
znaghash@googlemail.com

ABSTRACT

A software package for building knowledge base competitive software agents has been designed and developed. It uses PPIT (Personalized Planning and Integrated Testing) algorithm as a base for solving problems of SSRGT class. The PPIT algorithms elaborate moves in target positions depending on the knowledge in the agent's knowledge base, which contains formal structures of attributes, goals, strategies, plans, etc. In the current implementation of the package, the problem of the regular acquisition of the expert knowledge, was not solved. In this paper we suggest an approach to solving the problem by using JavaScript language interpreter for acquiring and managing knowledge formal structures, represented with JavaScript source code in Object Oriented manner.

1. INTRODUCTION

We developed a software package, components of which can be used as a basis for building knowledge [1, 2, 3] base competitive software agents, with the possibility to improve their knowledge in a regular way, by using experts' knowledge. The program of the agent should solve SSRGT class' problems. The SSRGT class represents problems, which **solution space** can be described as **reproducible game trees**. Combinatory games like chess and checkers can be example of such problems. Our software package should include programming implementations of algorithms which are developed for solving sub-problems of the main problem. Taking into account that each problem of the class has its common and individual aspects, software package should cover all common aspects and be as flexible as possible to be adapted for working with concrete problem's personalized aspects. In the paper we present an approach to regular acquisition of experts' knowledge by competitive agents.

2. BASICS OF THE PROGRAM PACKAGE FOR PPIT

As a base for the package, the PPIT program is chosen. It uses the general knowledge [4], as well as experts' personalized knowledge. Personalized expertise is the expertise, that human gets during his personal life experience, his world-view. Another specification of non-communicable knowledge is that it is a knowledge which has non-communicable components. The question of how that knowledge should be passed from human experts to the computer program, during their communication, is still unresolved. For the current implementation of the package as sample SSRGT class problem, the chess game was chosen.

Each node of the SSRGT class problems' tree represents some state of the system, in which the agent acts. In case of chess, it represents a state of the chess board with available pieces on it. Let's name each state (node of the tree) a **situation**. The agent has a possibility to act on the system by changing the system states. Actions are limited by the game rules. In the real world problems, as the game rules we have physical limitations of the environment.

Let's call the elementary influence of the agent on the system, during which it changes its state – an **action.** Thus, each next situation (node of the game tree) differs from the previous state by actions. The goal of the agent is to win the game, i.e. have a Mat situation on the board. For reaching the goal, it should build a chain of actions from the current and Mat situations. Each intermediate situation in the change is the intermediate goal of the agent. Let's call the composition of actions that connect some situation with the target situation, a **strategy**. Let's call the description of a strategy, a **plan**. The strategy explicitly defines the next turn from the current state of the system. To make a meaningful turn, the agent should find the strategy, which connects the target situation with the current situation. There are two ways to do that:

1. By dynamically traversing the game tree and testing different chains of turns for finding the one that reaches the target state. In other words, by searching the tree.
2. By using an already learned, stored plan, which corresponds to the current situation.

Both ways are important, but here we focus the second approach, that is by investigating possibilities of creating, storing and using plans. Two important attributes of the plans are initial and target states. Using that attributes, the agent chooses relevant plans from its knowledge base. Thus it is important to represent the system states within the agent's knowledge store. The agent also should realize the correspondence of the objects in its memory with the real objects of the tree and vice versa. According to this we can say that the agent first of all has to have possibility to differ real objects from each other. For example, it should differ different kinds of pieces, their colors, their possible relations (e.g. piece belongs to some field). Such kind of initial functionality can be prebuilt in the agent. By the usage of initial objects, agent can build higher level representation of the system attributes. As system attributes, we consider parts of it. State of the system can also be considered as an attributed built using lower level attributes.

We have used the Object Oriented (OO) programming methodology for designing the agent architecture. In accordance with it, we design the agent's knowledge base and its supporting procedures.

2.1. Implementation of algorithm in the form of dynamic loaded library PPIT_ROOTS dll

We design shells of PPIT programs as a composition of the following basic units:

- Reducing Hopeless Plans (RHP)
- Choosing Plans with Max Utility (CPMU)
- Generating Moves by a Plan (GMP) [5]

In present C++ implementation of PPIT [5] program unit of knowledge are realized as OO classes with the specialized interface for each type of knowledge and uniform for the program as a whole. In program operations the following procedures are realized:

- ✓ Selecting plans most suitable to a current position from sets of all plans (it is fulfilled for each situation).
- ✓ Reducing those plans which cannot be realized (for example if the plan suggests to use such resource which is not accessible) (it is fulfilled for each position).
- ✓ Construction of all trajectories, the fixed length (it is fulfilled for each position).
- ✓ Checking an accessibility of a purpose, for each trajectory (round each trajectory allowed bands are under construction, they are estimated on a tag of realizability of the selected purpose), and according to the received outcomes of assignment of some values or priorities (it is fulfilled for each trajectory)
- ✓ Calculation of the value or priority of the selected plan by assigning to it of a maximum estimation or priority of available trajectories (it is fulfilled for each schedule).
- ✓ Selecting plans with maximum value or priority, by a choice of the plan with a maximum estimation or priority from all available already estimated planes.
- ✓ Choosing operations according to already selected suitable plans.

3. AN APPROACH TO THE REGULAR ACQUISITION OF EXPERT KNOWLEDGE FOR TRADING AGENTS

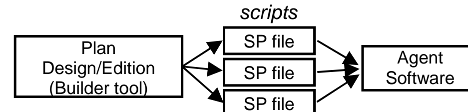
In the Management Strategy Provision (MSP) problem a company is competing in oligopoly market for some success criteria (max cumulative profit, max return on investment, etc.) and is going to make decisions in market situations that are consistent with the best strategy at least for defined periods of the competition.

In order to find the best performance strategy for the acting agent, various strategy plans need to be dynamically simulated (the process of making "on-the-job" performance assessment). This simulation supposes running the particular strategy plan (SP) for the agent throughout the game flow against the other competing parties, applying real (quantitative) values to the SP`s qualitative moves and thereby estimating them and further selecting the most acceptable one [6].

Let's suppose there is a mechanism for strategy plan assessment and selection. What about the development of the strategy plans?

The proposed approach is that a plan is described in some easy-readable high-level language (scripting language) as sequences of statements and stored in a separate data unit (some text file). In order to facilitate the process of writing a plan script another application – the Strategy Builder Tool is created. It is used for visual development/changing and further translation of the strategy plan into terms of the mentioned language. In statements there may be points with some parameters which would be subjects for the agent to perform variations and quantifications on a game tree in order to find the most appropriate values in particular situation [6]. Doing this way it would facilitate the creation of Strategy Plans libraries which further could be used by the agent as the knowledge base.

At first some format representing a strategy plan should be specified for a script that could be stored in a separate data unit - in a file (or multiple such scripts could be collected in one file).



The agent program itself should be able to read such script and act accordingly.

Another software part should be created for the strategy plans (scripts) definition and edition, so one can describe the agent behavior and store it in a separate file.

The key advantages of this approach are the following:

there is no need to change the agent-part source code making it to understand and run another strategy plan. More, the actual SP in use can be swapped with another without interrupting the simulation process.

The process of plan description becomes much easier because the user operates with limited set of high level scripting terms and operators that make the statements closer to natural languages. High-level means there is no matter which term or operator is applied to which type or kind of data or process evaluation. If there is a case when some term is not applicable anyway – it would be intuitively clear for the user (since by perceptive point of view the statement construction and terms themselves are close to natural language) and moreover, he would be notified about by the builder tool)

the plan creation or changing does not require the user to be a programmer; a user just needs to know about the game (model) rules and types of data/terms which are described in corresponding documentation.

Thus, we need some kind of specialized high-level language qualitatively describing the agent's behavior. It should be used by both parties (by the agent and the editor software units) when creating (editor) or reading (agent) a strategy plan. The language should operate with all task-specific entities in a qualitative (or behavioral) manner and define the points where the quantification must be done when dynamically tested by the agent part.

Obviously, every kind of competing problem (game) may be put in accordance with its own specific version of such a language, describing the objects, situations, states, events and actions (moves) that are specific for the given problem.

The description of such scripting language for dealing particularly with the TAC problem will be given below in "Operators and Functions" section.

3.1. Distinguishing TAC Events

The flow of the Trading Agent Competition day, in general, is described with the following row of events:

- Customer RFQs are received
- Customer orders are received
- Supplier offers are received
- Simulation status notification

Each event causes execution of its corresponding procedure, so for each of these events we should have an assigned "sub-strategy plan" (SSP). Described by terms of the language, such SSP consists of one or several statements with included actions. We can construct the statement so that the actions may occur, reoccur or be absent based on rules or/and conditions. Description of some kind of actions may require fixed position in a statement, following or preceding some other actions or conditions.

For different events some of the language terms and data objects may be not applicable and therefore might be inaccessible. So, for every event there is a set of objects and data that are active and operable.

Along with basic conditional operators, the actions and operations here are defined as high-level.

3.2. High Level Regulation Concepts

Saying "high-level" means independence from the type of operating data and the ability of dealing with some level of abstraction. This abstraction level will operate with comparative and absolute conceptions, for example, terms (that are operators, in fact) like "better", "worse", "similar", "soonest", "later", "more", "least", etc. And as defined by term "abstraction", these terms would be applied to any type of data that are used within the occurred event, i.e. during the procedure being executed.

3.3. Types of Data

For each event there is an active set of data objects and basic-type data. "Active" means that during the particular event only these data are accessible and may be operated with. There are some types of basic data. Objects are represented as sets of complex structured data (dynamic data vectors and tables) that may be changed in size and containing data as the simulation is running with encapsulated methods for handling these data. An example – "Deal history" object. It should collect and store information about deals made by the agent, i.e. storage of statistical data of deals that have ever been performed by the agent from the very start of the simulation. "Production history", "Customer orders history", "Supplier orders history", "Profit history", "Delivery history", "Factory utilization", "CPU Component history", etc. However, any data object should be "smart enough" for using by the high-level operators and functions (described below).

The offered scripting language includes the following types of operators: basic (conditional) and high-level. These are of the following types: date-time and period processing (terms like "today", "yesterday", "before", "… days ago", "… days before", "ever", "during", "when", etc.), comparative (terms like "better than", "like (equal)", "more than", "faster than"), evaluative (terms like "good", "better", "fast", "slow", "efficiency", etc.), calculative (terms like "more", "less", "count of", etc.), process workflow control (like cyclic operators with implicit number of loops, etc), critical values (like "max of …", "average of …", "min of …", etc.)

Functions describe particular actions agent may perform and there may be some functions that are available within particular event. Functions may be used as the final actions after conditional operators, as parts of rules or as independent (fixed) actions. Examples of functions are "keep count of ...", "order from supplier ...", "do nothing", "do like ...", etc.

There are two stages in processing of statements. On the first stage we may run any kind of routine tasks related to filtering, sorting, etc. of incoming data of event. Then, on the second stage we perform other actions (selection, etc), which may be described by rules or (as more complicated) may depend on some condition(s). In conditional statement several conditions may be logically grouped by using 'OR' and 'AND' operators. Also, there may be more than one statement defining the agent's behavior during the particular event.

In any statement there may be a point where the agent may do some variations on the game tree in order to find the most appropriate quantified value of some parameter for particular situation.

Below are examples of complete statement with fixed actions for "Customer RFQs are received" event:

'do_filtering_by_price'; 'do_filtering_by_date'; 'do_sort_by due date acceding'; 'select_first N requests'; 'select_requests_from_index M to K'

3.4. The Tool

Strategy Plans Builder tool is used for construction and edition of strategy plans visually, in an easy way, like drag-and-dropping elements within a CAD environment. When the design or change of an SP is done the plan description is translated into the text representation according to specified format and is saved in some structured format (for example, as XML file). Automatic translation means the final SP description could be constructed correctly, with no syntax errors and without corruption of statement format.

Usage of this tool facilitates development of new strategy plans as there is no need to create or change a plan manually. Instead, the user just creates the plan visually, and then the application translates it into the script using described scripting language and saves it in a file. The advantage of this is that it doesn't require any development skills from the person running the agent.

4. JAVASCRIPT INTERPRETER

PPIT_ROOTS library is the initial implementation of the PPIT algorithm. Its goal is in proving an ability of PPIT to successfully solve SSRGT class problems.

Historically, PPIT_ROOTS was designed to solve chess related problems. Thus, its implementation is strongly interconnected with chess concepts. Experiments with usage of expert knowledge for solving Reti and Nodareishvili etudes were proven viability of approach used in PPIT algorithm. However, PPIT_ROOTS has the following limitations:

1. It is impossible to add new concepts to the library knowledge-base or edit existing ones without the recompilation of its source code.
2. Library modules are designed for solving only chess related problems.
3. User, who uses the package, and tries to add new concepts, should be familiar with programming in C++ .

The new version of library – PPIT_BASE_V2, is a generalized version of PPIT_BASE with the following additional characteristics.

Concepts are more dynamically structured and have possibility to be loaded dynamically into the agent's knowledge-base, without compilation of the agent's source code. Having a possibility to dynamically learn new concepts or to improve the existing knowledge the system will regularly gain a new knowledge, as new concepts or new plans. Concepts descriptions are separated from the source code what is achieved by describing the concepts by scripting language and the interpreter of that language. This approach allows insertion of new concepts into the system, although the representation of already loaded concepts and concepts' management functionality is not considered yet. We consider the scripting language interpreter integrated with other parts of the package and having a functional interface for working with the concepts store. JavaScript is chosen as a scripting language. Its interpreters are widely available and its syntax is rich enough for describing object-oriented concepts.

The process of making knowledge base architecture enough dynamic for regular acquisition of the knowledge blocks requires the following reconstructions. Using input devises, sensors, agents can distinguish world realities from each other. Instances of the RealObject class in the system memory (mind) are reflections of world realities. RealObject class can have as much properties as system sensors can distinguish. In case of chess game, agent's sensors distinguish 4 properties:

- **Shape –** the shapes of world realities
- **PositionX** – position of the object on the X axis

- **PositionY** – position of the object on the Y axis
- **Color** – color of the object

Values of the properties are limited by the game rules and possible values are accepted in a certain way.

In case of the chess game, world is represented as a chess board. States of the chess board (the world), i.e. situations, are configurations of pieces (objects) on the chess board. This means, that situations can be described by describing the states (values) of the pieces' attributes at the given time interval.

Positions	
PositionX	[1..8]
PositionY	[1..8]

Agent can change board states by changing the states of the pieces. States of the pieces are combinations of the values of its attributes. Let's call each atomic change of the piece's state as an action. Action is associated with the piece, whose attributes are being changed. Executing the piece's action means an atomic change of the piece state.

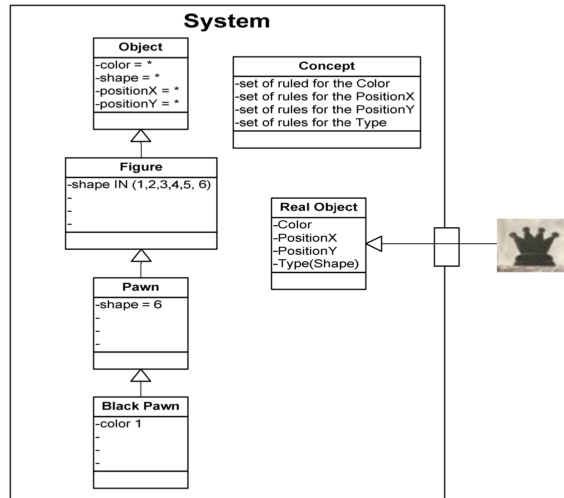
Experts' knowledge can be represented as a library of plans for achieving some goal state from the certain state. This means that for learning expert knowledge agent should have possibility to store plans within its knowledge base. Compositions of the units of the plans are descriptions of system states and actions of the system properties. Thus, agent should have possibility to learn descriptions of system states and objects' actions. Let's name these descriptions as concepts. We define concept learning as a possibility to store some information about the concepts within the agent's memory as a data structures and a possibility to associate that information with the corresponding realities of the world – real objects and the actions of real objects. For achieving that agent should implement 2 procedures **FIND** and **EXECUTE**. The **FIND** procedure gets the knowledge structure (a concept) as a parameter and finds the real object corresponding to the concept. After the object was found it creates an instance of RealObject class, and fills the attributes' values with corresponding attributes values of the found Object. Attribute values are being loaded into the agent's memory via its sensors.

EXECUTE procedure gets a concept and an action corresponding to the concept as parameters and first FINDs the object corresponding to the concept, then executes the action over the loaded instance of the RealObject class.

Using these procedures agent can operate with stored plans and act depending on them. It is possible to call the EXECUTE procedure's internal version, EXECUTE_I, which changes only attributes of the loaded instance of RealObject class and doesn't changes the real object's attributes (the state of the chess board). This function can be used for planning future events.

In the knowledge-base concepts are represented as instances of the Concept class. Concept class contains sets for storing matching rules for the each property of the RealObject class. Matching rules are pairs of matching functions and possible values, i.e. arguments for the matching function. As matching rules for the templates properties, mathematical functions can be used. The FIND procedure uses attribute matching rules for finding (matching) real objects corresponding to the properties. Using these structures it is possible to dynamically add new concepts by creating new objects of the Concept class and adding attribute matching rules to it. An instance of the Concept class can generalize another instance of the Concept. Generalization means that child concept also uses properties' rules of the parent concept for matching real objects. System can automatically check the rule overlapping and will use only more concrete rules.

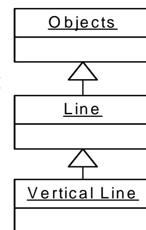
The most abstract concept is the Object, which corresponds to any found real object. The image bellow shows a mechanism of creating the "Black Pawn" concept:



Templates can have methods which are calculating values of some properties of templates depending on other template properties. Using these methods it is possible to do assumptions for some properties values depending on other properties' values.

Actions are represented as a set of rules of how real object's attributes can be changed. Within concepts actions are included as associative arrays in the following format: {actionName => [(rules for positionX, rules for positionY, rules for Shape, rules for Color), (….), ….], …}. Each action name represents a class of actions. For example action **Move** is a representation of several possible moves depending on the figure type. If all child-concepts of some concept have an action with the same name, then the parent concept also can have an action with that name, but without description. This approach is similar to the abstract virtual functions in terms of OOP. This means that the EXECUTE procedure, in case it tries to execute Move action over the Figure concept, should find the child-concept which corresponds to the found real object and then execute the action, i.e. change the real objects' attributes depending on the rules.

Another type of knowledge is sets. They are instances of the Set class. Sets represent all instances of the real objects that match to some Concept, which correspond to the set. Here concepts act as templates, for matching objects that correspond to the set. We say, that set is a parent of another set, if concept, that corresponds to the parent set is the parent of the concept of the child set. Sets can be used as arguments for the **IN** attributes matching function. The diagram on the right shows examples of the sets.



In many cases concepts are being defined depending on other concepts. An example of such relational concepts is a "Neighbor Field". It is a field which is near by another field. "Near by" means that coordinates of field A and B are matching the following rules: ($|Xa – Xb| = 1$ and $Ya = Yb$) OR ($|Ya – Yb| = 1$ and $Xa = Xb$). For supporting relative concepts in the PPIT_ROOTS_V2, it is possible to use other concepts' attributes in the attributes' matching rules of a concept. This means, that the FIND procedure first should find the inner concept, and then use the attribute values of found instance of the RealObject for matching rules of the searched concept. It is also possible to use concepts' actions within the matching rules of other concepts as functions in combination with attributes of other concepts. Such functions can modify the state of the found real object and then apply the matching

rules over the changed values. In such cases EXECUTE_I procedure is used.

To show how PPIT_ROOTS_V2 agent can learn non-trivial concepts, let's step by step organize learning of the "Check" concept. Our assumptions will not completely correspond to the chess concepts, but they will be quite similar. Check can mean that on the board is a King, which is under attack of another Piece. For describing the «King under attack of another Piece» concept, we can describe «A Piece is under attack of another Piece» concept and then extend from it, by setting the Shape attribute matching rule to the «KING» value. «Piece is under attack of another piece» means that Piece's coordinates are EQUEAL to the Field's coordinates, which is IN the «Attacking Fields of another Piece» set. For an «Attacking Fields» set, as a matching concept «Attacking Field» concept can be used. «Attacking Field» of the piece is a field, which coordinates are equal to the Piece's coordinates after executing Move action over them. So «Attacking Field» can extend from the «Field» concept and have additional attribute matching rules.

This means, that by using the concepts' structures and by executing several FIND and EXECUTE_I procedures in a chain, it is possible to find if there is a Check situation on the chessboard, and depending on that, decide the next action of the agent.

Scripts written using JavaScript are plain text files that can be loaded by the interpreter at runtime. Already loaded JavaScript objects can be edited during the program lifetime (runtime), not only by changing existing attributes values, but also by adding new attributes and methods. This means, that concepts which are already in the knowledge-base can be developed and adopted according to the new needs. In case if the knowledge-base has to be transmitted between different agents, concepts can be again serialized to the text file or be transferred via web. The JSON (JavaScript Object Notation) lightweight data-interchange format is designed for sending JavaScript object via web. Therefore, usage of the JavaScript as a language for representing knowledge concepts will easily allow agents in the future to interchange and synchronize their knowledge.

There is a vocabulary of chess concepts which contains about 400 concepts sorted from the simplest ones to the most complicated. The successful learning of those concepts will prove the adequacy of the method we used.

For simplifying the learning process of concepts a graphical editor is planned to be developed. The editor will automatically generate JavaScript files corresponding to the administrator commands.

Because of the slower work of the interpreting programs in comparison with the compiled ones, we consider the speed of the knowledge management system as a potential problem. For making the system faster, it is reasonable to compile the most useful JavaScript objects into the binary format on the runtime and improve the program speed this way.

As future steps, it is planned development of procedures for improving the existing knowledge, for example, by reorganizing associations between existing concepts. The system is being designed according to this possible improvement. This means, that dynamic architecture of the knowledge management subsystem will allow us to use several algorithms from the machine learning fields. Automated learning is considered as the next improvement of the package, and it mainly depends on the existence of dynamically manageable knowledge structures.

5. CONCLUSION

We have presented a software package PPIT_ROOTS that has been designed and developed for solving SSRGT class problems. It utilizes PPIT algorithm and proves its ability to successfully solve chess problems that are subclass of SSRGT. We also have listed the main limitations of PPIT_ROOTS package:

1. Impossibility to add new concepts to the library knowledge-base or edit existing ones without recompilation of its source code.
2. Impossibility to solve non-chess related problems.
3. Impossibility to add new concepts by the users who aren't familiar with programming and with C++ programming language.

These are limitations for organization of the regular learning (knowledge acquiring) process. According to that a new version of the package is being designed and developed, which uses JavaScript interpreter for representing the concepts within agents knowledge base. We have shown how new approach allows adding new concepts without recompilation of the package's source code and by that supports the learning regularity. As a next step we consider a graphical editor implementation that will allow an easier mechanism for adding new concepts.

ACKNOWLEDGEMENTS

We would like to express our gratitude to Professor Edward Pogossian for supervision of the work and to Emma Danielyan, Arpine Grigoryan, Artem Harutyunyan, Aram Antonyan for their constructive comments and support.

REFERENCES

[1] J. Frnkranz, "Machine Learning in Games", *A Survey Austrian Research Institute for Artificial Intelligence*, Wien, Austria ,Technical Report OEFAI-TR-2000-31 page 4

[2] S. Kosslyn, "Image and Mind". *Cambridge, MA Harvard University Press 1980*

[3] Z. Pylyshyn, "Seeing and Visualizing, It's Not What You Think", *An Essay On Vision And Visual Imagination, http://ruccs.rutgers.edu/faculty/pylyshyn.html*

[4] E. Pogossian. "Adaptation of Combinatorial Algorithms" (Russian), Yerevan.,1983, 293 pp.

[5] E. Pogossian, V. Vahradyan, A. Grigoryan,_"On Competing Agents Consistent with Expert Knowledge", *Lecture Notes in Computer Science, AIS-ADM-07: The Intern. Workshop on Autonomous Intelligent Systems - Agents and Data Mining*, June 5 -7, 2007, St. Petersburg, 11pp.

[6] T. Baghdasaryan, E. Danielyan, E. Pogossian "Supply Chain Management Strategy Provision by Game Tree Dynamic Analysis", *Fifth International Conference SBM2006*, Sept. 4-8, Sevastopol, 2p.

[7] T. Baghdasaryan "Scripting Language and Design Tool for Trading Agents Strategy Plans", *International Conference CSIT2007*, Yerevan, 5p