An Agent Based Formal Specification Language Processor

Vinitha H. Subburaj 1, Joseph E. Urban 2,

1 Department of Mathematics and Computer Science, Baldwin Wallace University,
275 Eastland Road, Berea, OH 44017, USA
2 Department of Industrial Engineering, Texas Tech University,
805 Boston Avenue, Lubbock, TX 79409, USA

{vsubbura@bw.edu, joseph.urban@ttu.edu}

ABSTRACT

In the software development life cycle, requirements elicitation and specification is the most important phase to be considered to avoid maintenance cost after the software development. Specifications written in natural language are often ambiguous, incorrect, and unreliable. Specifications serve a vital role in any software development and needs to validate for correctness before entering into design and implementation phases. One of the critical areas currently growing is agent systems. The agent domain was used in this paper to analyze the importance of formal specifications. The natural language specifications for agent systems were written as formal specifications for execution. In our approach, a formal specification language for agent systems, the Agent - Descartes specification language was taken into study. The agent specification language which was based on a functional model and is described by defining the input and output data, was used in the development of a language processor. In this paper, the steps and detailed analysis of this specification language processor developed for agent systems is described.

KEYWORDS

agent systems; language processor; specification language; software development.

1 INTRODUCTION

The Descartes specification language is an executable formal specification language that was developed that uses Hoare logic [1]. Existing work on the extensions available to the Descartes specification language had the following agent limits [9]: did not provide a conceptual framework for specifying agent properties; did not address high-level abstraction details of agent systems; and did not provide tool support to execute the agent specifications.
The main objective of this project was to implement a language processor that retains all its functionality with improvement in design and user friendliness. This project aimed at developing the language processor for the Descartes - Agent specification language using Java as the programming language. A challenging issue in this project was re-implementing a legacy system with recent technology. The current Descartes specification language processor was designed and implemented using a lexical analyzer written for Java (JLex), ANTLR parser generator which uses the LL(*) technique, and Java source code. The need of such a language and also development details can be found in this paper.

The remainder of this paper is structured as follows. Related work along with state of the art is discussed in Section 2. Section 3 discusses the design and implementation details of the Agent Descartes language processor. Section 4 discusses the future work from this research effort and Section 5 concluded this paper with a summary.

2 BACKGROUND

Taibi [4] used JLex and the Constructor of Useful Parsers (CUP) to generate a highly optimized Java-based lexical analyzer and parser for BPSL. JLex takes a specification file and creates a Java source file for the corresponding lexical analyzer being able to tokenize an input file and pass those tokens to the parser that checks them against the grammar of BPSL defined using a CUP grammar specification. CUP is a system for generating a Look Ahead Left Right parser, which uses a specification including embedded Java code and produced parsers. The possibility of embedding Java code into the CUP specification of a BPSL grammar allows for the checking of all possible lexical, syntactic, and semantic errors in BPSL specifications. In addition to syntactic checking, Taibi [4] has added action code to check for lexical and semantic errors in a BPSL specification and action code to convert the behavioral aspect specification into FSP notation, which enables use of a "Labelled Transition System Analyzer" (LTSA) to automatically check safety, liveliness, and reachability of the behavioral aspect of a pattern.

Berk [2] developed a lexical analyzer that decomposes an input stream of characters into tokens. The job of writing a lexical analyzer manually can be a tedious job, so Berk [2] developed the JLex software tool to ease the task. The JLex utility is based upon the Lex lexical analyzer generator model. Lex is a computer program that generates lexical analyzers based on the C programming language for the UNIX operating system. JLex is a lexical analyzer generator, written for Java, in Java. JLex can generate a Java source file from the specification file for the corresponding lexical analyzer. Hudson [3] developed CUP, which offers most of the features of "Yet Another Compiler Compiler" (YACC) parser generator developed by Johnson for the Unix operating system. CUP is written in and operates entirely with Java code.
Lee and Bryant [5] introduced the Contextual Natural Language Processing to handle problems in natural language and the DARPA Agent Markup Language (DAML). In this paper, the requirements were converted to the Extensible Markup Language (XML). The XML requirements were parsed using Contextual Natural Language Processing (CNLP) to build the knowledge base. The knowledge base is then converted to a Two Level Grammar (TLG). The knowledge base is used to convert the NL to formal VDM++ formal representation using the TLG. The domain knowledge base describes the relationship between components and constraints related to the system. This project applied the principles of transforming NL to formal specification language automatically using a parsing technique. A built TLG is used to bridge the gap between the informal and formal specification language.

Quesada [6] discussed a model based parser generator named ModelCC. Metadata annotations were used to achieve a mapping from an abstract syntax model (ASM) to a concrete syntax model (CSM). The ASM has a tree-like structure and can be compared to a traditional grammar based parser. The ModelCC parser generator parses trees and abstract syntax graphs. The connection between language design and language processor in terms of traditional methods and model-based methods was discussed in this paper.

Gomez-Sanz, et al. [7] introduced the INGENIAS development kit for multi agent system development. In this paper, agent specifications are produced with a visual editor. The visual editor allows the user to browse through the instance of the meta model entities. The tool was developed using a model driven development approach. This tool allows automatic conversion of specifications to functional multi-agent systems. When the agent specifications are written, the INGENIAS tool converts the specifications into Java code. The complete functionality of the tool was not completed expecting the developer to specify each step of task flow to generate the multi agent system models.

3. THE DESCARTES SPECIFICATION LANGUAGE AND THE EXTENSIONS

The Descartes specification language [1] was designed for use throughout the software life cycle. The relationship between the input and the output of a system is functionally specified when using this specification language. Descartes defines the input data and output data and then relates them in such a way that output data becomes a function of input data. The data structuring methods used with this language are known as Hoare trees. These Hoare trees use three structuring methods, namely direct product, discriminated union, and sequence.

During analysis or synthesis of a Hoare tree, it is not that every node will acquire a value. Urban [1] in his dissertation has explained the matching of input strings with the Hoare tree as follows. “Matching of direct products and discriminated unions is performed as in a SNOBOL pattern match for concatenation and alternation respectively.”

Analysis of input data results in values that control the shape of a synthesis tree. Thus, discriminated union selects the first subtree having the name that defines the input data. In the case of a sequence node, if there are a number of elements matched then they are referenced by the node name capitalized and immediately followed by a sharp, “#”. In the case of a sequence node, if there is an individual element matched, then it is referenced by the node name capitalized immediately followed by the element reference enclosed in parentheses.

.

Extensions were made to the Descartes specification language for specifying intelligent software agent architectures [11]. The Wooldridge abstract architecture was used to define agent properties and to introduce new constructs to the Descartes specification language for agent specification. Specifying an agent system with the lack of concrete architecture support and tool support were drawbacks of the extensions made to the Descartes specification language by Subburaj and Urban [11]. Agent state, environment state, decision function made up of action and perception, and cycle information were the newly added constructs to the extended Descartes specification language in order to specify the intelligent software agent behavior. Medina and Urban [12] provided extensions to the Descartes specification language for formally specifying reactive agents.

4. DESIGN AND IMPLEMENTATION OF THE LANGUAGE PROCESSOR

This section describes the design methodology adopted and design approaches that were used in this research effort to develop a language processor for executing agent specifications. The different stages in the development of the interpreter can found in Figure 1. Initially, the agent grammar was written in ANTLR format. The input specifications along with the inputs to the specifications are the two input files read by the language processor. The ANTLR grammars consists of the lexer and parser rules to syntactically and semantically validate an input specification. The next step is to identify the Descartes agent modules in the specification. A Descartes agent module is analyzed and synthesized separately. A Descartes agent module interacts with the knowledge/belief base to read the agent context rules. Based on the initial set of agent rules and the input parameters, corresponding events are triggered. The last step is to synthesize output based on the decisions taken by the Descartes agent module.

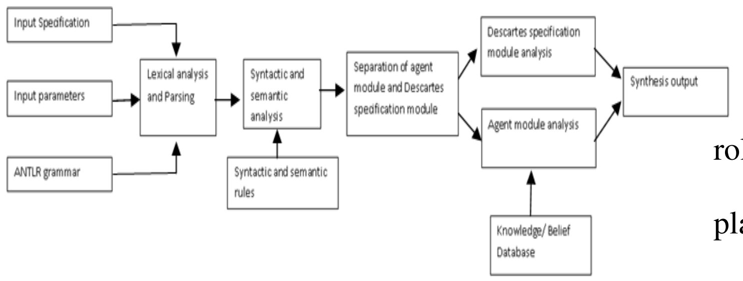

Input Specification
Input parameters
ANTLR grammar
Lexical analysis and Parsing
Syntactic and semantic analysis
Syntactic and semantic rules
Separation of agent module and Descartes specification module
Descartes specification module analysis
Agent module analysis
Knowledge/ Belief Database
Synthesis output

Figure 1. Agent- Descartes Specification Language Processor

4.1 Phase 1 - Lexical Analysis and Parsing

Traditionally, a lexical analyzer is implemented using a finite state automaton. A finite state automaton is an abstract machine that consists of a finite set of states and transactions from state to state, based on an input stream. Syntactic analysis of the input specification is based on a context-free grammar and done by parsing the specifications. In this research effort, an Eclipse tool was used to build and develop the interpreter. Eclipse supports the use of an ANTL plug-in that takes a combined (lexer+parser) grammar and converts the input specifications into tokens. The interpreter also displays the abstract syntax tree of the input specifications after parsing.
Consider the specification of a quoting agent specification described in Figure 2. The quoting agent specifications along with the input parameters are inputs to the interpreter. In the lexical analysis phase, the input specification will be lexically analyzed and converted into the tokens. The lexical analysis accepts a specification in the form of stream, applies the lexical rules, and converts the specification into the lexical tokens shown in Figure 3.

```
agent QUOTING_AGENT_(STOCK_NAME)
goal
        !get_quote_on_stock
attributes
        STOCK_NAME
                STRING
        sector_name
                'value_read_from_kb'
        price_range
                'value_read_from_kb'
roles
        NONE
plans
        triggered_events+
                get_quote
                        context

        (STOCK_NAME)_IS_TRUE

        (SECTOR_NAME)_IS_TRUE
                        methods

        contact_corresponding_sector

        'get_quote_info_from_sector'
                monitor_quote
                        context

        (PRICE_RANGE)_IS_TRUE

        (STOCK_NAME)_IS_TRUE
                        methods

        check_quote_on_stock

        'check_for_quote_info'

        display_monitored_quote

        'display_the_quote_info'
```

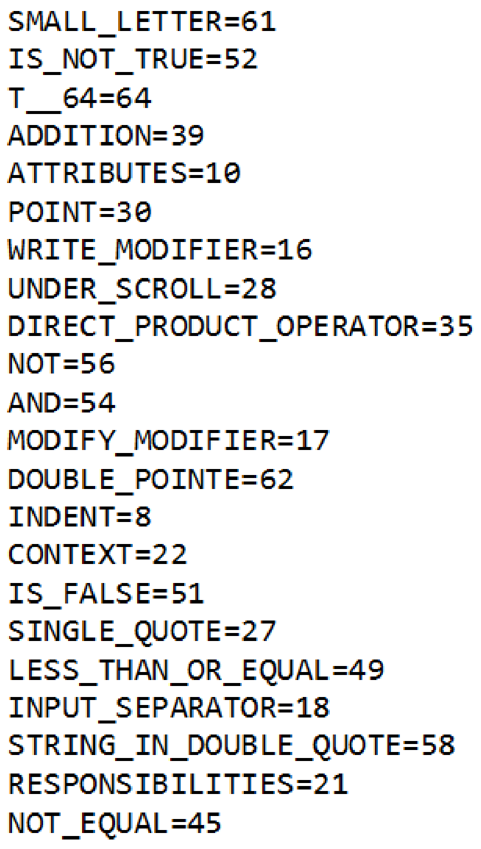
```
return
        GOAL
                TRIGGERED_EVENTS+
                        GET_QUOTE

        CONTACT_CORRESPONDING_SEC
TOR
                        MONITOR_QUOTE

        CHECK_QUOTE_ON_STOCK

        DISPLAY_MONITORED_QUOTE
```

Figure 2. Quoting Agent Specification in Agent Descartes



```
SMALL_LETTER=61
IS_NOT_TRUE=52
T__64=64
ADDITION=39
ATTRIBUTES=10
POINT=30
WRITE_MODIFIER=16
UNDER_SCROLL=28
DIRECT_PRODUCT_OPERATOR=35
NOT=56
AND=54
MODIFY_MODIFIER=17
DOUBLE_POINTE=62
INDENT=8
CONTEXT=22
IS_FALSE=51
SINGLE_QUOTE=27
LESS_THAN_OR_EQUAL=49
INPUT_SEPARATOR=18
STRING_IN_DOUBLE_QUOTE=58
RESPONSIBILITIES=21
NOT_EQUAL=45
```

Figure 3. List of Tokens

After lexical analysis, the input specification gets separated into individual tokens with a unique identifier value associated with each token. This unique identifier value is used to create the Abstract Syntax Tree (AST), as well as to apply the parser rules defined in the ANTLR grammar. As seen in Figure 3, the keywords AND, NOT, IS_FALSE, and IS_NOT_TRUE have unique identifier values assigned as 54, 56, 51, and 52, respectively. The lexer and parser library available with the Eclipse interpreter tool automatically generate these values.

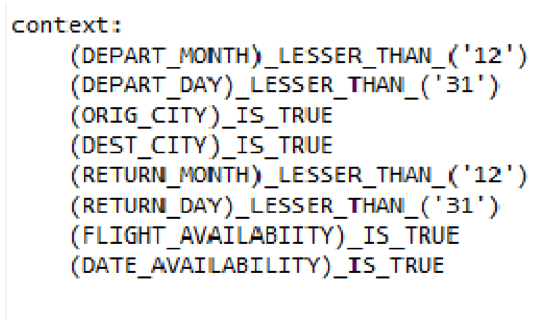
During the syntactic analysis phase, an AST is formed from the tokens created as a result of the lexical analysis phase. During this phase, if any syntactic error occurs, the program will be terminated.

4.2 Phase 2 - Symbol Table Creation

The agent Descartes specification language processor implements a symbol table to analyze the agent specifications. The interpreter reads the input specifications and then ensures the specification to be syntactically correct. The next step reads and interprets the agent specification based on the goal, attributes, roles, and plans. The interpreter distinguishes the different agents in the given specification and then uniquely identifies each of the agent's goal, attributes, roles, and plans. The symbol table is also used to store values for each of the agent primitives to be used for output synthesis. In the Descartes specification language, a specification tree consists of match nodes and reference nodes. The reference nodes return the values stored in the corresponding match nodes. This language feature of reference nodes acquiring the values stored in the corresponding match nodes is preserved in the extensions to specify agent systems. The parser rules are used in the creation of the symbol table. The symbol table is used to store and retrieve the values of the match nodes with respect to the agent properties.

4.3 Phase 3 – Knowledge/Belief Base

The knowledge/belief base constitutes an important part of the Descartes – Agent language processor for specifying agent systems [9, 10]. The agents achieve their goals by acting autonomously based on the rules and perceptions. The knowledge/belief base consists of an initial set of agent beliefs. The interpreter reads the knowledge/belief base before processing the output for the specifications. The knowledge/belief base that consists of a set of rules is specified using the logical, arithmetic, and Boolean primitives of the Descartes specification language. The extended logical primitives IS_TRUE, IS_NOT_TRUE, IS_FALSE, IS_NOT_FALSE were also implemented in this research effort. A sample knowledge/belief base file used along with an agent specification can be found in Figure 4.



```
context:
    (DEPART_MONTH)_LESSER_THAN_('12')
    (DEPART_DAY)_LESSER_THAN_('31')
    (ORIG_CITY)_IS_TRUE
    (DEST_CITY)_IS_TRUE
    (RETURN_MONTH)_LESSER_THAN_('12')
    (RETURN_DAY)_LESSER_THAN_('31')
    (FLIGHT_AVAILABIITY)_IS_TRUE
    (DATE_AVAILABILITY)_IS_TRUE
```

Figure 4. Knowledge/Belief Base

The interpreter updates the knowledge/belief base during runtime. Also, a user can add the initial set of agent beliefs to this base before the execution of an agent specification.

4.4 Phase 4 - Agent Module Analysis

Identifying the 'agent' keyword provides for the separation of an agent module from the rest of a module declaration in the specification. Once this analysis is made, then agent analysis follows. The sequence of steps to be followed while analyzing an agent module can be found in Figure 5.

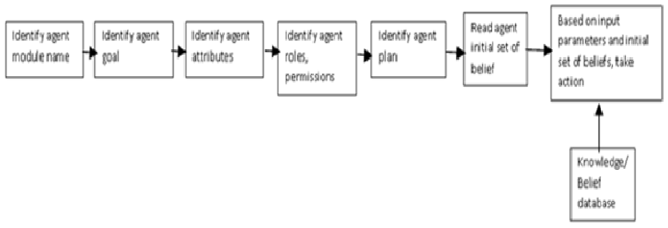

Identify agent module name
Identify agent goal
Identify agent attributes
Identify agent roles, permissions
Identify agent plan
Read agent initial set of belief
Based on input parameters and initial set of beliefs, take action
Knowledge/ Belief database

Figure 5. Agent Module Analysis

The agent module analysis starts by identifying the agent module name. The agent module name, which is similar to the Descartes specification module declaration, is appended by the 'agent' keyword. Then, the agent goal and the input parameters are analyzed. Once the agent goal has been identified, then the different agent roles, associated permissions, and the responsibilities are identified. An agent plan consists of the context and the list of actions to be taken by the agent in order to achieve the goal. Once, the agent plan, based on the set of rules, has been identified then the input analysis and synthesis of output follows.

4.5 Phase 5 - Semantic Analysis

The semantic analysis of the specification takes the tokens and the abstract syntax tree as input. The ANTLR grammar has been added with the set of semantic rules to identify and analyze if the specifications are semantically correct. The semantic analysis checks if the input parameters match the analysis tree, checks if every input parameter has a corresponding analysis tree, checks for reference nodes that have corresponding match nodes, and checks if the analysis and synthesis trees are semantically correct. Once the extended Descartes specification has been analyzed semantically then the next phase is to generate the symbol table.

4.6 Phase 6 - Output Generation

The last phase in the implementation of the Descartes - Agent language processor is the output generation phase. The process of output generation can be explained using the following three steps:

a. Check if the input matches the specification. If the input does not match with the specification, then stop the execution process.

b. Check the input with the belief/knowledge base that consists of agent rules. If the input does not satisfy the belief/knowledge base rules, then stop the execution process.

c. Based on synthesis (return) tree and the belief/knowledge base rules display results.

Display the triggered events methods and the decision made from step 2. Display an error if there is a mismatch.

The above three steps summarize the actions taken by the interpreter to analyze the input and then to synthesis the output. The input to the interpreter consists of four files: grammar (lexer+parser), agent specification, input to the specification, and the belief/knowledge base of agent rules. The outputs obtained after execution of the specification are as follows: a list of tokens, abstract syntax tree, symbol table, and the output file.

The processor was implemented using the Java programming language. Eclipse was the tool used to build and run the interpreter. The plugin used along with the Eclipse tool to define the language grammar was the ANTLR (Combined grammar) plugin. The ANTLR plugin uses an LL(*) Parser Generator. This plugin used along with the Eclipse tool allowed for the grammar to be successfully built before using as input to the interpreter. Parr and Fisher [8] in their paper discussed the LL(*) to parse the input specifications based on the ANTLR grammar. LL(*) parsing allows the feature of parsing the input stream based on arbitrary look ahead. LL(*) parsers are top-down parsers and are used to parse the input from Left to Right. Since the Descartes specification language uses tree structures to specify the software systems and uses indentation to denote various levels, the LL(*) technique was used to look ahead several characters during parsing. The LL(*) parsers use regular expressions to perform the look ahead function.

5 EVALUATION AND CONTRIBUTIONS TO THE STATE OF ART

This section discusses an evaluation of the Descartes – Agent language processor based on the notations and modeling criteria given by Sturm and Shehory [13]. Also, the contributions of the developed tool to the field of formal agent specifications are also described in this section.

Notations are symbols used in the methodology to represent elements in the system. A modeling technique describes a set of models used in the methodology to depict a system with different levels of abstraction. The list of properties used to assess the notations and modeling techniques used in the research effort are presented in Table 3. A summary of the evaluation of the extended Descartes specification processor based on notations and modeling techniques can be found in Table 3.

Notations and modeling techniques [50]	**Description**	**Evaluation of the Descartes – Agent language**
Accessibility	This criterion refers to the simplicity of the notations used in the method. Experts and novices should be able to understand the concepts	4
Analyzability	The capability to identify aspects that may seem unclear and the interrelations between such aspects	4
Complexity management	Ability of the method to deal with different levels of abstraction	4
Executability	Ability to execute the specification to test the aspects	4
Expressiveness	Applicability to multiple domains	4
Modularity	Ability to specify the agent system in an iterative incremental manner	4
Preciseness	Ability to avoid misinterpretation	4

Table 3. Evaluation Based On Notations And Modeling Techniques [13].

The Descartes – Agent language allows for the formal specification of inherently complex agent systems with agent properties, such as goal, roles, attributes, plans, and initial set of beliefs. The tool support implemented in this research effort allows for the execution of the formal specification of agent systems. The analysis of syntactic and semantic correctness of the formal agent specifications was achieved by the tool support implemented. The overhead of manually validating the specification correctness has been eliminated in this research effort through the implemented tool support.

The Descartes – Agent language for specifying agent systems can be used as a model for formally specifying complex agent systems. The complexity comes from specifying the different agent properties such as agent goal, plans, attributes, and roles. Extended Descartes does not add complexity to Descartes to specify agent properties such as goal, attributes, roles, plans, and initial set of beliefs. Use of tool support and case studies to validate the Descartes - Agent language allows for the syntactic and semantic correctness of the agent specifications.

6 FUTURE WORK

The future of the approach discussed in this paper will be beneficial to the development of critical software systems that use agent oriented software engineering over the traditional methods. The intelligent software systems used in making autonomous decisions and in achieving goals could be potentially embedded in software systems during five to ten years from now. The approach discussed in this paper intends to contribute to the research and development of agent oriented software systems. The specifications written in a formal specification language can be converted to UML diagrams. A future direction of this research is to develop an automated tool that does the following: execute the specifications, convert the specifications to UML diagrams, and then produce partial code. Development of architecture models to aid the design and development of the agent specifications is also one of the future directions.

7 SUMMARY

Software agent systems are extensively used in the current world due to the explosive growth of information and data. The agent systems are used to implement complex tasks and features. Software agents have their goals with properties like autonomy, reactiveness, interaction, and learning. The agents achieve their goal by cooperation of agents. Agent oriented software engineering is now developing with major emphasis on the requirements and design phases. New tools and techniques to guide the requirements and design phases of the agent system development are increasingly being developed for advancement. In this paper, discussion of a formal specification language used to specify agent systems along with tool support to execute the specifications was discussed. Due to an increasing need for reliable specifications in software development, this paper discussed the implementation details of a language processor used to execute the agent specifications. Different stages in developing the language processor for executing the formal specifications were discussed in this paper. Also, the different stages in analyzing and processing the agent model used to write the formal specifications was also discussed in this paper.

8 REFERENCES

[1] Urban, J. E., A Specification Language and Its Processor, Ph.D. Dissertation, Computer Science Department, University of Southwestern Louisiana, 1977, received an Association for Computing Machinery Doctoral Dissertation Award, 1978, http://awards.acm.org/award_winners/urban_1921243.cfm

[2] Berk, E. and Ananian, C. S., JLex: A Lexical Analyzer Generator for Java (TM), http://www.cs.princeton.edu/\~appel/modern/java/JLex/current/manual.htm%l, 2005.

[3] Hudson, S. E., CUP User Manual, Graphics Visualization and Usability Center, Georgia Institute of Technology, March 1996.

[4] Taibi, T., "Converting BPSL Behavioral Specification to FSP Using a Java-Based Parser Generator," International Journal of Computing and Information Sciences, Vol. 1, No. 1, 2003, pp. 33-42.

[5] Lee, B.-S. and Bryant, B. R., "Contextual Natural Language Processing and DAML for Understanding Software Requirements Specifications," Proceedings of the 19th International Conference on Computational Linguistics-Volume 1, Association for Computational Linguistics, 2002, pp. 1-7.

[6] Quesada, L., "A Model-Driven Parser Generator With Reference Resolution Support," Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 394-397, pp. 394-397.

[7] Gomez-Sanz, J. J., Fuentes, R., Pavon, J., and Garcia-Magarino, I., "INGENIAS Development Kit: A Visual Multi-Agent System Development Environment," Proceedings of the 7th international Joint Conference on Autonomous Agents and Multiagent Systems: Demo Papers. International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 1675-1676.

[8] Parr, T. and Fisher, K., "LL (*): The Foundation of the ANTLR Parser Generator," ACM SIGPLAN Notices, Vol. 47, No. 6, 2012, pp. 425-436.

[9] Subburaj, H. V., Urban, J. E., and Shah, M. R., "Specification of Safety Critical Systems with Intelligent Software Agent Method," Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE 2012), Redwood City, San Francisco Bay, CA, 2012, pp. 578-581.

[10] Subburaj, H. V. and Urban, J. E., “A Formal Specification Language for Modeling Agent Systems,” Proceedings of the Second International Conference on Informatics and Applications (ICIA) – IEEE, Poland, 2013, pp. 300-305.

[11] Subburaj, V. H. and Urban, J. E., “Intelligent Software Agent Design Issues With Extensions to the Descartes Specification Language,” Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering, San Francisco Bay, California, 2010, pp. 297-305.

[12] Medina, M. A. and Urban, J. E., “An Approach to Deriving Reactive Agent Designs From Extensions to the Descartes Specification Language,” Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems, IEEE, 2007, pp. 363-367.

[13] Sturm, A. and Shehory, O., “A Framework for Evaluating Agent-Oriented Methodologies,” Agent-Oriented Information Systems, Springer, 2004, pp. 94-109.