A Constraint-Based Approach for Analyzing Financial Market Operations

Samuil Nikolov, Vladimir Nikolov,
Anatoliy Antonov

Abstract: *The article describes a framework for modelling and verification of constraint rules on operations with financial instruments. These constraints are applied on certain attributes of domains of financial objects. A methodology and implementation of automatic constraint analysis in two steps is presented. The first step involves preparation of constraints on specified domains and creation of formulas defining them. The other step consists in waiting for real time transactions and responding to them by alerting the user on newly occurred constraint violations. Computation reduction method is proposed. A satisfaction coefficient is calculated that aids the end user in taking consecutive actions on their portfolio.*

Keywords: *Constraint satisfaction problems, Expert systems, Financial systems, Finance, Programming approaches*

INTRODUCTION

Modern financial markets are characterized by a rich variety of offered financial instruments and include many participants with competitive goals, which are achieved in highly dynamic market conditions. The financial instruments are often pooled in hierarchical structures (classification groups) like portfolios, sub-portfolios by country, currency, instrument type. The market participants must conform to regulatory rules specifying the distribution of the assets under their control in the separate classification groups. In many cases, the rules for asset allocation in the groups are alternative to each other and require decision making. Due to the high complexity of the regulatory rules, the market participants often make mistakes while operating on the highly dynamic markets and violate the rules about the distribution of the limited financial resources. Thus, they impose high risk on the organizations or people whose money they are dealing with. This makes automatization of financial operation analysis in real time a priority. The market participant must make a decision about buying or selling of a certain position before the market trend changes. The dealer has to quickly simulate an operation on a position and make sure all the constraint rules are satisfied before executing the deal on the market. To ensure better experience for market participants and quickly identify constraint violations, formal specification of regulatory rules is required. These regulatory rules are different for each country and are based on local laws.

The problems reviewed in the article are related to constraint programming [1]. The authors formally specify constraints using expressions and use constraint logic programming over finite domains. The expressions are processed by a production system program containing facts and rules. The proposed approach allows “What if...” simulations and flexibility in taking alternative decisions. This guarantees continuity and validity of market participants' actions. The analysis is performed in two steps. During the initial check of the constraints, facts representing domains are asserted and synthetic rules are generated from constraint specification by a special parsing rule. After firing, the new rules yield the required result. The described step in a real-life system involves analysis of a large number of positions included in the different hierarchical structures. The second step involves real time simulation of single financial transactions, causing incremental changes in one or several domains.

This reduces the rules that need recalculation only to those that are logically connected to the changed position avoiding unnecessary recalculations and rule generations. Examples in the article use the CLIPS production system syntax.
Chun et al. [5] show practical realization of constraint problems using the language JSolver. The authors use constraint satisfaction problem solving methods in AI, declarative programming and deterministic search in Java. Their solution is characterized by coding JSolver instructions inside a Java program that calculates the result of the constraints after being executed. The user has no control over constraint setup in run time.
Saad et al. [10] present a general constraint model of a rule based system. It is represented as a new class of nonstandard constraint satisfaction problems called Dynamic Domain Constraint Satisfaction Problem (DD CSP). DDCSP unify several CSP extensions, providing a more comprehensive and efficient framework for rule based reasoning. The proposed solutions to nonstandard problems are based on extensions, through algorithms and methods that are encoded in the rule based systems solving the CSP.
Felfernig et al. [6] develop a methodology for solving CSP in distributed applications using agents. Their proposed solution is used for delivery and integration of product configurations that require integration of configuration systems of different vendors. The solution is based on direct coding of specific CSP problems in the developed system.
The other topic that this publication concerns is the ability of a program to extend itself. Oreizy [8] reviews theoretically the ways this can be done and the problems that could occur during the process. Usually code generation is done by wizards and intelligence features of integrated development environments that do not operate in run time. Some authors like Buck and Hollingsworth [4] generate debug information and performance check routines in run time. This way the program is extended with highly specialized code. Other ways of generating code in run time are used in aspect oriented programming. Interesting developments are used in AspectJ [9] and Aspecktwerkz [3]. They can add filters to the entry and exit points of functions and add new classes to java programs.

1. DEFINITION OF CONSTRAINTS

1.1 Specifying domains

The financial positions data of the traders in the described system are distributed in a tree structure and are represented by a list of positions associated to every node in the structure. The lists of every sub node of the structure can be obtained by applying certain filter conditions on its parent node. The root node's list is loaded according to a condition associated with it. Domains are a set of positions and their attributes that are combined in a specific node of the hierarchical structure. The method for specifying domains is shown on figure 1. Every position participating in a position list is characterized by a set of attributes – L1, L2, etc. Each attribute has its own value type. Aggregation functions can be used to create new artificial attributes of data domains as shown on the figure. This way, the set of attributes is expanded dynamically. Calculation functions like max, min, and sum can be applied to attributes of the positions in the list to insert new items to the domain data.

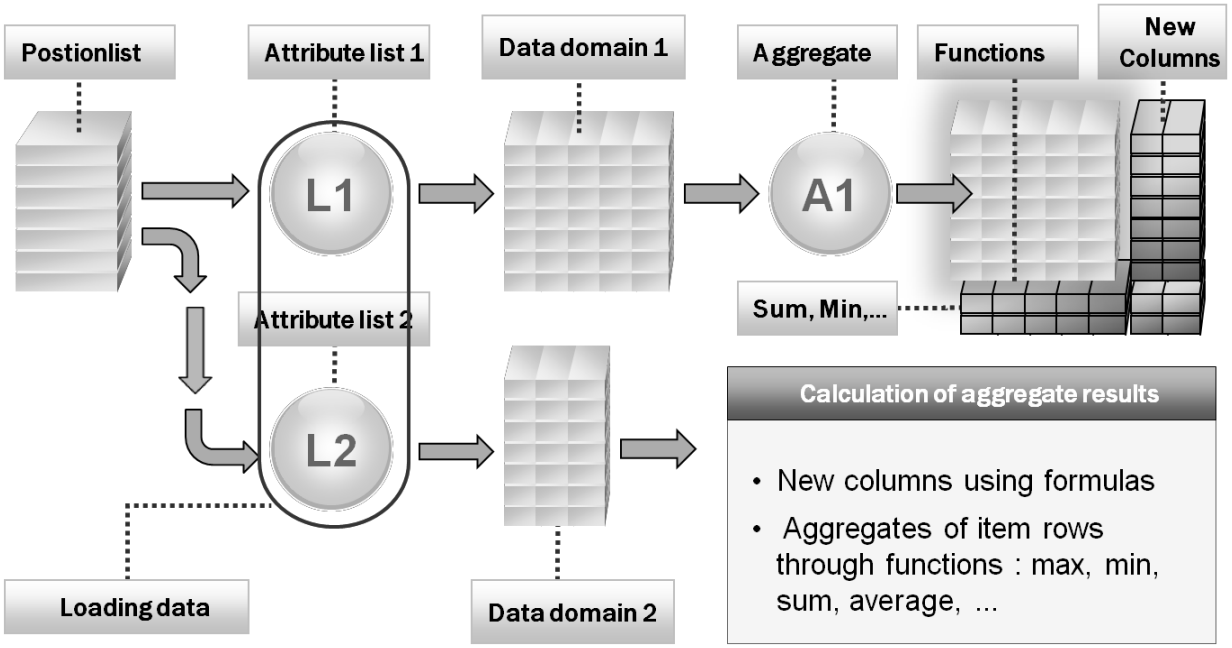

Figure 1. Domain definition method

1.2 Regulatory rule definition

Every regulatory rule is represented by logical expression defined over data in domain's positions as shown on figure 2. Arguments of the expression are: single element attributes, attributes of all elements, attributes added with aggregation calculations and aggregated function values. These arguments can participate in algebraic functions (+, -, *, /, ln, exp), conditional (>, >=, <, <=, =, <>) or loop (foreach) operations and arithmetic if operator. Logical results (true or false) and alerts are calculated from the conditional expressions. For every logical condition, a coefficient of satisfaction / violation can be calculated because it is a comparison between two values. After the logical conditions are defined, they are included in a constraint expression (formula) using the operators and, or,

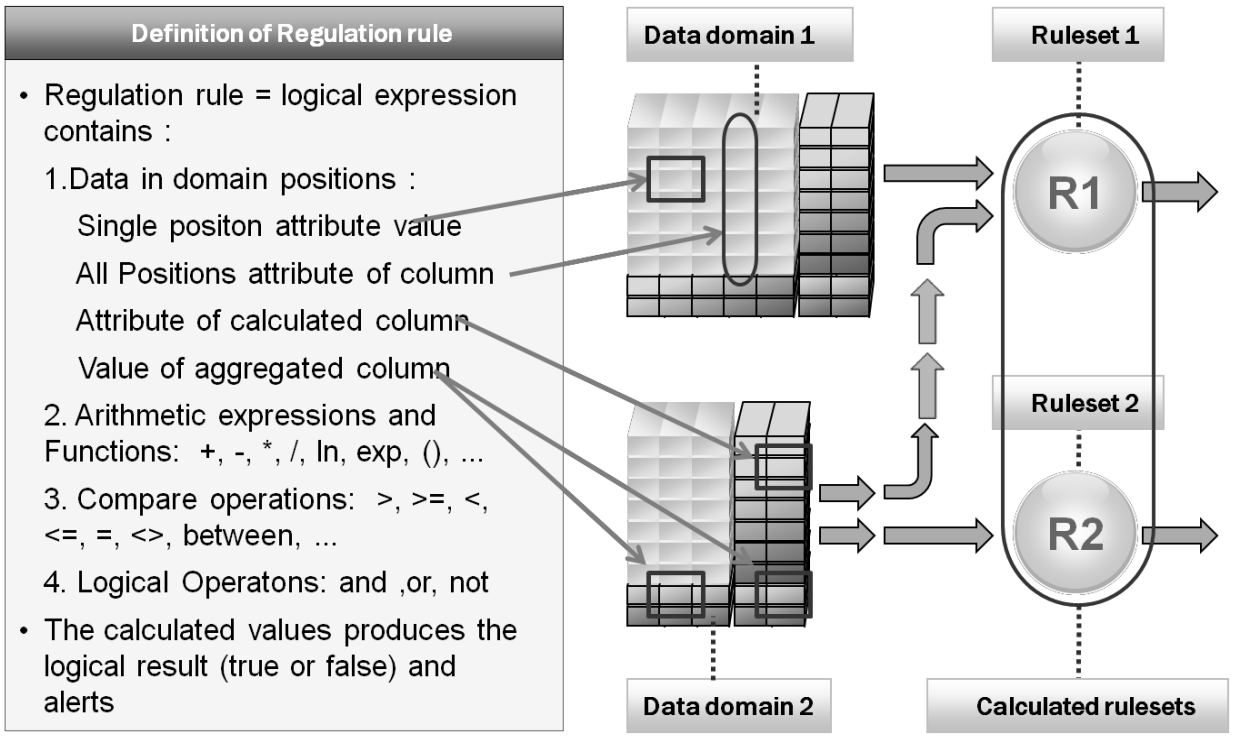

Figure 2. Definition of regulation rule

not, or not, and not and parentheses.

2. PARSING AND CALCULATION OF CONSTRAINTS

After defining the constraint expression, compilation and compliance check can be started. When a domain attribute definition is encountered, the parsing code generates a set of facts. If an operation is encountered – it generates a rule and adds it to its own set of rules. After the parsing is finished, the newly generated rules are fired, and their result is the required constraint compliance. The code generated for one of each set of operations that occur during the process of parsing will be reviewed.
When parsing a domain definition specified by a single position or filtered from the whole list of positions, all the necessary attributes are loaded. For each position, the rule generates two facts – the first contains root element identifier, the position identifier and the list of concerned attributes. The second one contains a domain identifier and the position identifier and is used to associate the position with the specified domain. For example, to generate a position in German 5-year bunds with notional amount of 100000 associated with the domain GovernmentBond the parser generates the following production system facts:

```
(Portfolio German5yBund 100000.0)                    (1)
(GovernmentBond German5yBund)
```

To define unions, intersections, differences and symmetric differences of two domains, new rules are added to the production system program. They use two declarations of domain association to generate a third one - to the new resulting domain. The following example rule shows a generated rule that unites the domains "government bond" and "corporate bond" in a new domain – "bond":

```
(defrule GenUnionBond
  (or
   (logical (GovernmentBond ?PosId))                 (2)
   (logical (CorporateBond ?PosId))
  )
=>
   (assert (Bond ?PosId))
)
```

The parsing rule also generates rules for arithmetical and group operations on the defined domains. It generates different identifiers for the results of each such operation. For example, to summarize the notional amounts of all the loaded positions, the following rule is generated:

```
(defrule Sum1
  (logical (forall(Portfolio ?PosId ?PosValue )))
  ?c1<-(Aggregate1 ?Value)
  =>                                                 (3)
   (if (<> ?PosValue 0.0) then
          (retract ?c1)
          (assert (Aggregate1 (+ ?Value ?PosValue)))
))
```

For the other group and arithmetical operations, similar rules are generated. The

approach is also used when generating logical rules. The difference is that in their case, besides the result - true or false, the parser also generates instructions to calculate the level of satisfaction of the logical operation. Adding comparison operation for two previously generated arithmetic operations – one aggregation and one multiplication is done by generating and adding the following rule to the program:

```
(defrule Less1
  (logical (Aggregate2 ?Value1))
  (logical (Multiplication1 ?Value2))
  =>                                                                    (4)
  (assert (Less1 (< ?Value1 ?Value2)  (/ (- ?Value2  ?Value1) ?Value2 )))
)
```

The final result of the constraint processing is a combination of logical condition results connected with and, or, not, and not, and or not. The generated rules use reasoning maintenance [11] principle when fired. It requires the production system to make sure that the assertions of facts on the right-hand side (RHS) of a rule are logically dependent upon pattern entities matching patterns on the left-hand side (LHS) of a rule. Thus, when a rule's precondition fact is retracted, the production system will automatically retract the facts asserted in the RHS from its knowledge base. As many of the generated rules use the results from other rules, removing a fact describing a position will cause a chain of fact retracting and cause recalculation of the rule results. The notable thing here is that such action will cause recalculation of only the concerned rules or only the parts of the constraint that are affected by the position removal or change.

3. REAL TIME OPERATION

After parsing the constraint expression and generating the rules and facts for it, the production system contains a set of newly created mutually dependent rules in its working memory. Their preconditions are either domain attribute descriptors or results of operations on such. Due to simulation of market trading, new attribute values can be added, or the existing ones can be changed or deleted. Checking the compliance of the portfolio with the regulatory requirements is a complex process due to the need for affiliation checks, aggregation and group function calculations. For example, to execute the maximum function, the system has to perform different actions if the maximal element is removed compared to the case when a smaller element is removed. Adding a new element to a domain that is a parameter of minimum function somewhere inside the expression should cause different behavior if the new element is the minimum, or if it is larger than the current one in the domain. Such considerations apply to all the other aggregation or calculation functions. This is the reason why they are implemented by generating a rule rather than a simple expression using production system's internal functions as in [2]. The rules have a set of facts as preconditions - representing the function arguments and assert a new fact - representing the function result. The changed value of the modified position affects only limited set of rules as is shown on figure 3. Firing only the affected rules recalculates the satisfaction coefficient of the logical conditions and the market trader can be alerted by the system about violated or nearly- violated rules resulting from his intended transaction. By reducing the number of fired rules, the end user is notified about the suitability of his market intentions before the current market conditions change.

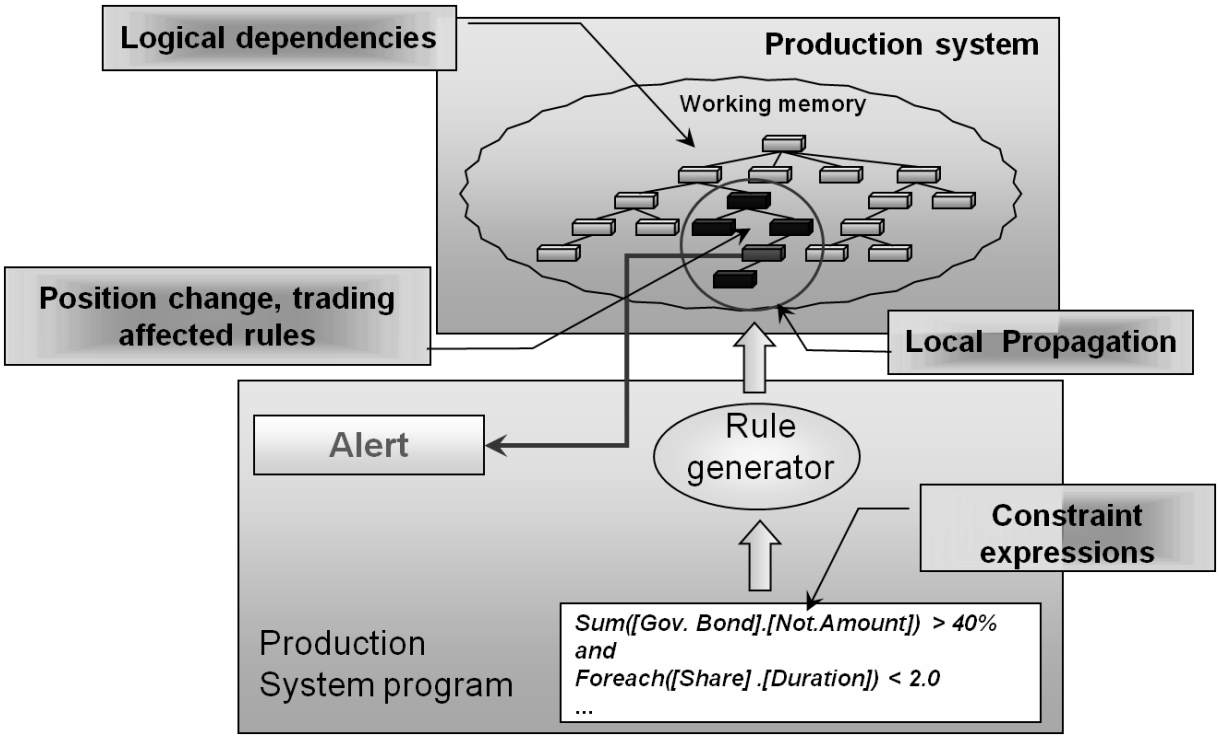

Figure 3. Local propagation of changes in the system in run time

4. EXAMPLE CONSTRAINT DEFINITION

As an example, a regulation will be reviewed that requires an investment company (a pension fund for example) to invest more than 50% of their assets in government bonds, less than 40% of their assets in corporate bonds, less than 20% in shares and less than 90% in any type of bonds. The first thing that needs to be done while defining the constraint is to determine the necessary domains and attributes inside it. In this case, a single attribute is used - the notional amount of the concerned position lists, which is referred in the constraint expression as [Not.Amount]. The domains are as follows: the full portfolio of the investment company ([Portfolio]), a subportfolio containing government bonds ([Government Bond]), a subportfolio containing corporate bonds ([Corporate Bond]), a subportfolio containing Shares ([Share]) and the union between government and corporate bonds ([Government Bond Union Corporate Bond]). The next step is to define each of the arithmetical and logical operations on them. For instance, to define the first logical operation (at least 50% of the investment assets should be in government bonds) a declaration should be added that the sum of the domain containing the notional amounts of government bonds should be more than the sum of the notional amounts of the positions in the entire portfolio, multiplied by 0.5. This can be specified as:

$$\text{Sum}([\text{Government Bond}].[\text{Not.Amount}]) > 0.5*\text{Sum}([\text{Portfolio}].[\text{Not.Amount}]) \quad \textbf{(5)}$$

Similarly, all the logical operations are generated and connected with the “and” logical operation. The final expression, representing the example regulatory requirements is as follows:

$$\begin{aligned}&\text{Sum}([\text{Government Bond}].[\text{Not.Amount}]) > 0.5*\text{Sum}([\text{Portfolio}].[\text{Not.Amount}])\\ &\text{and Sum}([\text{Corporate Bond}].[\text{Not.Amount}]) < 0.4*\text{ Sum}([\text{Portfolio}].[\text{Not.Amount}])\\ &\text{and Sum}([\text{Share}].[\text{Not.Amount}]) < \ 0.2*\text{ Sum}([\text{Portfolio}].[\text{Not.Amount}])\end{aligned} \quad \textbf{(6)}$$

and Sum([Government Bond Union Corporate Bond].[Not.Amount]) < 0.9* Sum([Portfolio].[Not.Amount])

When the production system program is started, first the parsing rules are fired to generate new facts and rules. In this case, they recognize the domains and their attributes and load the notional amounts of the available positions. For each loaded position, a fact like the first one in (1) is generated. A set of basic domain assignment facts like the second fact in (1), is asserted, taking into consideration each position underlying instrument's type. The domain identifiers in the facts are GovernmentBond, CorporateBond and Share. For the complex domain representing the union between "Government Bond" and "Corporate Bond", the rule (2) is generated. It asserts a new fact that specifies the members of the Bond domain as all the members of government or corporate bond domains. For each distinct "Sum" operation in the condition expression, a rule similar to (3) is generated. The rule names and resulting facts are substituted with properly generated identifiers. The rule condition is changed so that it uses the domain identifier that is being summarized. It is fired for each fact containing the particular domain identifier and a position identifier. The rule (3) summarizes all the notional amounts of portfolio positions and is generated for each right side of the comparisons in the example expression. For the comparison operations and the logical and between them, rules similar to (4) are generated. After the parsing phase, the program contains a set of new facts and a set of generated rules that represent the example expression. The production system program thus expands itself as result of parsing the input expression.

The next step is to check if the current portfolio setup conforms to the specified requirements. This is done automatically when the rules are generated – after the parsing stops, the newly generated rules are on the production system's agenda and firing according to their dependencies. The results of the constraints application on the tested portfolio are inside the asserted facts and can be visualized with proper tools. Every logical condition result fact also contains the satisfaction coefficient of the condition. Considering it, the end user can change the nominal of the positions, delete some of them or insert new ones to certain domains in order to satisfy the conditions, if these are not satisfied initially.

If the portfolio satisfies the constraints encoded in the expression, the user can experiment with adding, deleting or modifying positions and thus simulating what will happen if actual transactions are exercised. Let's suppose the position described with the fact (1) needs to be sold to the market. This action is done by retracting the two facts that were asserted when the position was loaded. Retracting them will remove the logical support for the assertions in several other rules. Removing a fact's logical support causes production systems to retract the fact itself from its knowledge base. For instance, the fact that states that German5yBund is a member of the Bond domain, that is generated by a firing of rule (2) would also be retracted. The sum of the portfolio positions calculated by rule (3) will also be retracted as will be the sum of the notional amounts of the positions in GovernmentBond domain. The sums will then be recalculated again without the nominal of the German5yBund. Retracting the fact that declares the position as member of the Bond domain will cause also recalculation of that domain's sum, but the sums of corporate bonds and shares will not be affected as their preconditions are not affected by the change. Thus, only the calculations that are necessary are performed which allows faster verification of the compliance.

CONCLUSIONS AND FUTURE WORK

The main features of the presented methodology are:

- The described approach allows representing regulatory rules using expressions defined by the user. The production system's program is evolving due to user input, generating facts and rules corresponding to the supplied expression. Processing of the

expressions cause loading of only the object sets and attributes that are concerned by the regulatory rules;

- The code generation, loading and execution approach combined with the mechanism of local propagation of changes allows distinguishing and executing only the necessary calculations thus enhancing the productivity in market simulation mode. This is possible due to generating rules for all arithmetic, comparison and logical operations as well as the group functions included in the expression. Only affected rules are fired after a knowledge base modification occurs.

The presented approach is implemented by the authors in a commercial system. A developed interactive interface allows the user to correctly form the expressions corresponding to the regulatory rules. The expressions can contain string, date and enumerated types of attributes besides the numeric ones, demonstrated in the article. The development supports use of financial position lists and is a part of a framework for building ontology-based dynamic applications [7]. Actual participation of the user in the market is being simulated by the implemented system. The described approach can be used in other cases where constraint modelling is required.

REFERENCES

[1] Apt K. Principles of constraint programming. Cambridge University Press, 2003.

[2] Bessiere C. Constraint propagation. Foundations of Art. Intell. 2, 2006, pp.29-83.

[3] Bonêr J. Aspectwerkz - dynamic aop for java. In Proc. of the 3rd Int. Conf. on Aspect-Oriented Software Development, Mancaster, UK, 2004, ACM Press.

[4] Buck B. and Hollingsworth J. An API for Runtime Code Patching. Int. J. High Perform. Comput. Appl. 14, 4 , 2000, pp. 317-329.

[5] Chun A., Hon A., Chun W., Waltz Filtering in Java with Jsolver, In Proc. of the Practical Applications of Java, London, 1999.

[6] Felfernig A. et al. Distributed Configuration as Distributed Dynamic Constraint Satisfaction. In Proc. of the 14th IEA/AIE, Budapest, Hungary, 2001, pp. 434–444.

[7] Nikolov S. and Antonov A. Framework for building ontology-based dynamic applications. In Proc. CompSysTech, 2010, pp. 83-88.

[8] Oreizy P., Medvidovic N. and Taylor R. Architecture-based runtime software evolution. In Proc. 20th int. conf. on Software engineering, 1998, pp.177-186.

[9] Russ M. AspectJ Cookbook. O'Reilly Media, 2004.

[10] Saad B. et al. DDCSP: Constraint satisfaction problem used for rule-based system. Int. Conf. on Data Storage and Data Engineering, IEEE, 2010.

[11] Smith B. and Kelleher G. *Reason Maintenance Systems and their Applications*. Halsted Press, 1988, New York, NY, USA

ABOUT THE AUTHORS

Eurorisk Systems Ltd.
31, General Kiselov Str.
9002 Varna, Bulgaria

Samuil Nikolov
E-mail: samuil at eurorisksystems dot com
Vladimir Nikolov
E-mail: nikolov at eurorisksystems dot com
Anatoliy Antonov
E-mail: antonov at eurorisksystems dot com