Dynamic Capabilities of a Compliance Check Software System

Ventsislav Nikolov

Abstract: *In this paper automated logical dependencies support for a limit checking software system is considered. By providing and observing of dynamic limits the system based on production systems truth maintenance is able to work effectively on solving large scale problems. The system is appropriate for regulatory checks in different fields.*

Keywords: *Compliance check, Limits, Production system, Automated check*

ДИНАМИЧНИ ВЪЗМОЖНОСТИ НА СОФТУЕРНА СИСТЕМА ЗА ПРОВЕРКА НА ЛИМИТНИ ОГРАНИЧЕНИЯ

Венцислав Николов

Резюме: *В статията се разглежда автоматизирана поддръжка на логически зависимости в софтуерна система за лимитни ограничения. Чрез осигуряване и наблюдение на динамични лимити, системата, базирана на продукционна система с поддръжка на истиност, може да работи с висока ефективност за решаване на значителни по сложност проблеми. Системата е подходяща за проверка на регулаторни изисквания в различни области.*

Ключови думи: *Проверка за съответствие, Лимити, Продукционна система, Автоматизирана проверка.*

1. Introduction

In every corporation, institution or market participant there are some regulatory requirements that should be regularly checked for violation. These requirements can be on different levels: international, national, corporate, team or other specific level. Some of these regulations are obligatory while others are only recommending. Their check can be done either manual or automated or partly automated. The advantage of the automated check is that it can be done faster and the system doing that could be permanently executed so for example some automated trading systems can always observe for any violations [1] [2] [7] [12] [14]. A software system started on a server can be used as automated checking system observing some preliminary defined limits definitions. Such limits could be for example: “the amount of assets in a given investment must not exceed 15% of all items in the portfolio”. The solution presented here is realized as compliance check software system based on a production system which is optimized to observe any rules having left-hand side as antecedent and right-hand side as consequent [9]. The developed system is generally intended for the financial institutions, but it could easily be adapted for other domains. The system here is able to check automatically some defined beforehand rules which are based not only on the regulations but also on the hedge or insurance in order to prevent any significant losses in the investment. The assets are considered in the system as elements which can be grouped in sets of

elements representing portfolios. Every element contains properties according to the instrument type of the asset. An example of an element is a financial contract and its properties could be: start date, end date, name, value, etc. The properties depend on the type of the contract and some of them can be changed in contrast with others that stay permanent during the contract time period. For example, the values can be changed but the start and end dates stay the same.
Some similar software solutions provide graphical controls as user interface that is not always very flexible [13]. In our solution a specific language is defined for this purpose which allows convenient rules to be defined for internal data types and limits. In [10] such a system is described with three data types and the last one, that is the most important, shows the violated and satisfied limits.

2. Solution

The statements of the input language are translated into production system code for automatic check. Such a system works in a quite different way compared to the traditional object-oriented languages. The production systems comprise productions which are considered as rules which can be executed in arbitrary order according to the pattern matching. Example of two statements in the compliance check language realized here is:

```
set set1 = subset(set0, (not (PV < 100.6)));
double_result result4 = avg(set1, 1.0 + 0.2 * volume);
```

The first statement defines a new set called set1 as subset of another existing set called set0 selecting only the elements of set0 for which the property PV is not smaller than the constant value 100.6.

The same example with exchanged statements will generate two rules but if there is not conflict the activation of the second rule in some cases could be before the first one. Yet if a fact matches both rules left-hand sides then according to the conflicts resolution strategy the rules fire. That is why a check is done in the parsing stage that all identifiers definitions exist before using them.

The translation from the compliance check language into the production system code is done by ANTLR [11] which automatically generates lexical analyzer and parser by analyzing a grammar provided beforehand. The translation is conducted by performing actions expressed by code fragments into the grammar which are executed when the parser recognizes an expression.

Tree representation

The entire structure of a program in the complieance check language can be represented as a tree providing an easy way for visualization and edit of the nodes. The tree is built in the parsing stage and always is available in the memory on demand. It can be provided to the user so that if a node is selected the information it contains can be used to modify the programming code. Once the expressions are loaded however, it is impossible to modify them while the system works. Modification is permitted only when the system is stopped.

An example of the translation and tree representation can be given for the following expression:

```
set set0 = positions;
```

```
set set1 = subset(set0, (not (CLEAN_VALUE < 100.6)));
set set2 = subset(set0, CLEAN_VALUE = 1.5e+2) and set1;
```

It creates a new set of elements as a subset of set0, which should already exist, combined with the elements of set1 that should exist too. The production system used here is Drools [3] and the expressions above are translated into the following rules:

```
rule "inline2" // subset with given filter from set 'set0'
  when
        NamedItem(name == "inline0", $pos : value)
        eval((((getPropertyValue($pos,"CLEAN_VALUE" )) == 150.0) ))
  then
        insertLogical(new NamedItem("inline2", $pos));
end

rule "inline3" // 'inline2' and 'set1'
  when
        NamedItem(name == "inline2", $pos0 : value)
        NamedItem(name == "inline1", $pos1 : value)
        eval((getPropertyValue($pos0, "id")).equals((getPropertyValue($pos1, "id"))))
  then
  insertLogical(new NamedItem("inline3", $pos0));
end

// custom set 'set2' is 'inline3'
```

As it can be seen every custom name is internally represented by its system name generated by the internal name generator. This is done in order to avoid any problems with the variables names provided by the user. The translation process is automated by the parser and lexer. Taking into account the common production systems features the expressions can easily be translated into another language like well-known C based CLIPS [4] [5].

One of the main advantages of the production systems is that they are based on the RETE algorithm [8] which is optimized in respect to the rules execution. It is a pattern matching algorithm based on internally built nets which nodes correspond to the patterns in the left-hand side of the rules and there are facts associated to them which determine the patterns matching. In opposite, the native rule matching algorithms are very ineffective. One of the disadvantages of the RETE algorithm is that in some cases it may cause significant memory consumption.

The tree representation of the upper expression is shown in fig. 1. A tree in the user interface is shown in fig. 2.

```
|-set2 (1:65)(1:110)[set set2=subset(set0, CLEAN_VALUE=1.5e+2) and set1;]
```

```
|       ...
|       +
|       |-inline2 (1:76) (1:100) [subset(set0, CLEAN_VALUE = 1.5e+2)]
|       |       ...
|       |       +
|       |       |-set0 (1:1) (1:0) []
|       |       |       ...
|       |-set1 (1:32) (1:63) [subset(set0, (not (PV < 100.6)))]
|       |       ...
|       |       +
|       |       |-set0 (1:1) (1:0) []
|       |       |       ...
```

Fig. 1 Tree of syntactic elements

The tree in fig. 1 is internal programming structure which contains syntax elements in its nodes similarly to an abstract syntax tree. The coordinates of the first and last characters of the syntax elements are shown as number of line and number of character in the line. These coordinates can be seen in braces in fig. 1. The text of the element is also available as information.

Selecting a given node in the user interface tree, shown in fig. 2, provides additional information as type of the node, its definition and other specific information.

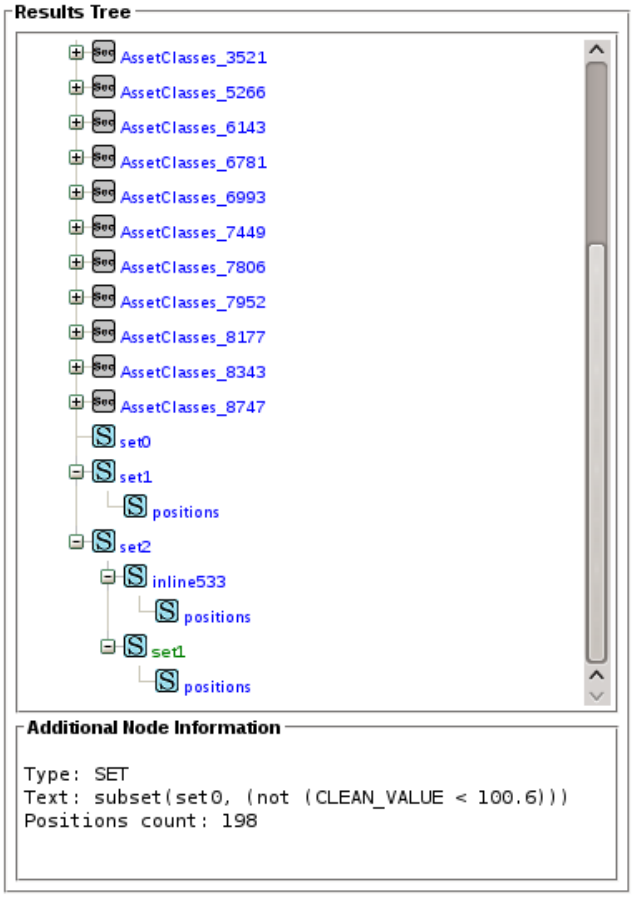

Fig. 2 Tree representation of the language elements

Software implementation

The main part of the parser relies on the objects structure realized using the “interpreter” design pattern [6]. Every element, either terminal or non-terminal, is implementation of an interface using a context of internal data and producing the resulting element of the transformed language. The context data contains data structures, internal names generators and other programming constructs needed for the translation process.

```
public interface DroolsElement {
  public String createElement(Context cxt);
}
```

Dynamic capabilities

It is important that the memory state depends on the time moment of the system working. The system usage can be separated to the following stages:

1) The script is loaded.

2) The elements of the general (default) set are inserted.

3) Additional elements are inserted or existing are removed or modified.

4) The resulting limits and other data are extracted from the memory.

The resulting limits extraction is just retrieving and analyzing the facts in the production system working memory. Once loaded the script stays in the system and every change in the facts set causes changes of the other dependent facts. The modification of an element in the production system is equivalent to its removal and insertion again with new data filled in its fields. Thus the logical dependencies are updated in real time and the system dynamically keeps the facts updated.

The dynamic capabilities are usable either in the case when there are too many values for a given property of an element or when it is not known which values can be assigned to it. For example, if the elements have a property called country and a set should be created for every country then an expression should be written for every possible country.

```
set set1 = subset(positions, country = 'Germany');
set set1 = subset(positions, country = 'Italy');
...
```

Instead of that only one expression can be written using the dynamic capabilities:

```
dynamicSet(set1, country, 'countries');
```

The named prefix ‘countries’ can be used after that to refer all sets together or selecting some of them:

```
limit myLimit = Forall('prefix1', distinct_by(country = 'all'), Sum('countries', volume * 0.1) >= Sum('countries', PV * 0.2));"
```

Omitting the “distinct_by” construct causes all the corresponding sets to be compared. If a tree node, that is a set of elements, does not exist in a tree then the check is not performed and the limit is not violated.

A variety of additional operations are possible for the dynamic sets: union of subsets according to a given condition, counting the number of dynamic subsets, using of different mathematical functions, etc.

External functions

Taking into account that the system is realized as a software library it provides some interfaces which can be implemented outside the library in the embedding system and thus additional functions can be used to extend the existing ones. For example, in the expression below an external function called “add_year” is used which adds a year to a specified date:

set mySet1 = subset(positions, date = function("add_year", "10.01.2009", "1"));

The number of parameters of the “function” varies according to its purpose. Another example is a function returning the date for which the check is performed:

set mySet2 = subset(positions, date < function("eval_date"));

It is also possible to use a function as parameter to another function:

set mySet3 = subset(positions, date = function("add_year", function("eval_date"), "1"));

Thus, the additional functions can be added without need to change the library which provides additional flexibility to the system.

Summary

The system is realized as a usable software module and it is practically used as part of other more complex systems. It has been tested for complex expressions and a huge number of elements in the memory. For example, 180 lines of expressions in one of the tests were transformed into 340 production rules and 10000 elements were inserted as facts into the production system working memory. In the first inference these facts caused creation of new 110 436 facts overall that lasted 13 minutes and 19 seconds. After that an element is removed which causes removing of other 11 logically dependent facts and that operation lasts 129 milliseconds. Insertion of a new element lasts 1 ms and it does not caused automatic insertion of outer facts. Obtaining of information for the facts in the working memory however also takes some time, about 300 ms were needed. The logical inference and the limits results were completely correct. In fig. 3 the table of the limits can be seen in the user interface where the violated limits are shown with “No” in the column “satistfied”.

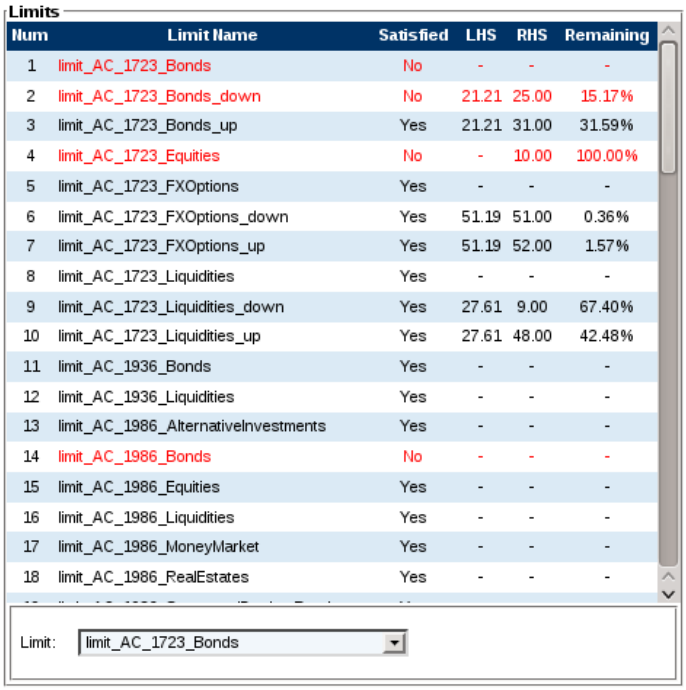

Fig. 3 The list of resulting limits

There are different kinds of limits. Some of them are comparison between two sides: left-hand side and right-hand side and in this case the calculated values of both sides are additionaly shown. Some other limits however are only check of some condition for all elements in a given set or many sets. Such limits are shown without values for left-hand and right-hand side.

References

[1]. Algorithmic trading. An overview. Instinet, 2005

[2]. Baker, G., S. Tiwari. Algorithmic trading: perceptions and challenges. Edhec Risk Advisory, 2004.

[3]. Bali, M. Drools JBoss Rules 5.0 Developer's Guide. Develop rules-based business logic using the Drools platform. Packt Publishing, 2009.

[4]. CLIPS Reference Manual, Volume I, Basic Programming Guide, 2007.

[5]. CLIPS Reference Manual, Volume II, Advanced Programming Guide, 2008.

[6]. Gamma, E., et all. Design Patterns: elements of reusable object-oriented software. Addison Wesley, 1994.

[7]. Kim, Kendall. Electronic and algorithmic trading technology. The complete guide. Elsevier, 2007.

[8]. Ligeza, A. Logical Foundations for Rule-Based Systems. Springer Science & Business Media, 2006.

[9]. Mayr, H. Database and Expert Systems Applications: Proceedings of the 12th International Conference, DEXA 2001 Munich, Germany, September 3 – 5, 2001.

[10]. Nikolov, S., V. Nikolov, A. Antonov. A constraint-based approach for analysing financial market operations. Proceedings of the 14th International Conference on Computer Systems and Technologies, ACM New York, NY, USA, ISBN: 978-1-4503-2021-4, pp. 231 – 238.

[11]. Parr, T. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf, 2007.

[12]. Rao, S. Algorithmic Trading: Pros and Cons. Tata Consultancy Services Limited, 2007.

[13]. Strasberger, M. Risk Limit Systems and Capital Allocation in Financial Institutions. Banks and Bank Systems, Vol. 1, Iss. 4, 2006.

[14]. Ward, S., Sherald, M. Successful Trading Using Artificial Intelligence.

For contacts:
Dr. Ventsislav Nikolov
Senior Software Developer
Eurorisk Systems Ltd.
31, General Kiselov Str., 9002 Varna, Bulgaria
E-mail: vnikolov at eurorisksystems dot com