CLIPS Representation of Ontology Classes in an Ontology-Driven Information System Builder – Part 1

Samuil Nikolov

Abstract: *The paper describes the structure of a CLIPS program representing an ontology class in an ontology-driven information system builder. The program has a specific format which is presented in the paper. A set of programs can be used to quickly and easily create an ontology-driven information system when loaded in a C++ core which interoperates with an instance of a CLIPS interpreter. An example bank loan class expressed as a CLIPS program is presented in the second part of the publication.*

Keywords: *Ontology-Driven Information Systems, Knowledge Representation, Expert systems*

INTRODUCTION

The goal of the publication is to describe a way for representing ontology classes used by the author while developing an ontology-driven information system (IS) builder. The classes contain the information system business logic and describe its user interface. The general principles of operation and the database structure are presented in other publications with participation of the author and are not subject of this paper. The IS builder allows creation of a full-scale information system by developing a set of CLIPS programs that define all the classes of the target ontology.

Wand and Weber [13] initiated the idea to define a formal model of an information system. Nicola Guarino [4] analyzed the ways an ontology could be used in the information system distinguishing between a temporal and structural dimension and defined the term ontology-driven information systems. Fonseca and Martin [2], Wand and Weber [12] and Hoefferer [5] relate Guarino's classification with previously stated ideas about using conceptual schemas or meta models for formally defining information system structure. In the latest entry in the Encyclopedia of Database Systems, Tom Gruber [3] gives a broader definition of ontology that validates Guarino's definition. Yildiz and Miksch [14] review the challenges that come with integrating ontologies into information systems. Some of the directions in the development of ontology-concerned information systems are conceptual modeling of the IS databases [1] and refining the communication of IS requirements through better structured ontology data [6], [10]. This paper describes a specific approach to representing ontology classes in an information system. According to Guarino's classification of ontology-driven information systems, the described IS uses the ontology in both user interface and application components [4]. No implementation details of such IS were found during the author's research.

The IS presented in the paper is built by development-time dissemination of the ontology data and creating separate script files, each representing one of the ontology classes. The scripting language used is CLIPS rule based programming language. It operates in a CLIPS environment loaded by a C++ core developed by the author which principles of operation are presented in [7]. CLIPS was chosen as a widely used productive development and delivery expert system tool which provides a complete environment for the construction of rule and/or object based expert systems.

1. CLIPS PROGRAM FOR REPRESENTING ONTOLOGY

The ontology classes that define the ontology-driven information system are represented as separate files, each containing a stand-alone CLIPS program. Each file is loaded in the CLIPS interpreter either by special commands issued in the rules of another file, or loaded explicitly by the end user of the information system as described in [7]. The CLIPS definition of ontology classes contains:

- Facts describing the class attributes;

- Facts defining how the attribute values are visualized when the UI is generated;
- Rules for processing of attributes and manipulating the UI appearance.

Bundling of data with the methods operating on them is a widespread design pattern which is one of the main principles of object oriented programming (OOP). Encapsulation enables the programmer to group data and the subroutines that operate on them in one place, thus hiding irrelevant details from the users of an abstraction [9]. The proposed description of ontology classes uses similar principles, but combines facts and rules operating on them. The difference from OOP classes is that ontology classes defined by CLIPS programs can interact with each other only through the instances stored in the database. The database structure is described in detail in [8] and is not subject of this publication. Following are descriptions of the three CLIPS program parts.

1.1. Class attributes description

The basis of the ontology class description in the presented CLIPS program is a set of fact templates, defining the class attributes. The fact templates can be either simple or complex. The simple ones contain a single value – either a string or a floating-point number. Dates and integers are represented with floating point numbers subjected to certain restrictions according to template type. Following are example templates representing two attributes of a financial instrument class - “Instrument identifier” of type text and “Instrument type” of type selectable text:

```
(deftemplate InstrumentId
(slot  type (type STRING)  (default "Text"))
(slot str_value      (type STRING)  (default " "))      (1)
(slot value          (type FLOAT)   (default 0.000000)))

(deftemplate InstrumentType
(slot type (type STRING) (default "ComboText"))
(slot str_value      (type STRING)  (default ""))
(slot value          (type INTEGER)          (default 0))       (2)
(slot list    (type STRING)(default "list_ ITYPE")))
```

The fact template defining instrument type attribute has an additional slot named “list”. It has a default value that coincides with the name of a fact that determines the list of possible values:

```
(list_ITYPE        "Fix Bond"        "Floater Bond"  "Stepped Bond" "Zero Bond" ... )
```

Another feature of the fact template (2) is that its “value” slot is of type “INTEGER”. In this case, the index of the chosen text is stored in the database, instead of the text itself. This is done to enable easier translations to other languages of the combo box texts, which are mostly used for visualizing such selectable text attributes [8].
Complex attributes contain more than one value. Most commonly, these are tables of data which columns are represented by CLIPS multifield. An example of these attributes is the following table description, containing cash flow payments of a financial instrument. It has eight columns, containing payment number, type and date of payment, remaining amount, payment sum, etc. The meaning of the columns and the type of data inside them are specified in additional facts, linked to the main template through the default values of its col_names and col_types slots:

```
(deftemplate CASHFLOW_GRID
(slot type           (type STRING)  (default  "Grid"))
(slot col_num        (type INTEGER)          (default 8))
(slot  col_names  (type  STRING)  (default  "cn_CF_GRID"))
```

```
(slot col_types (type STRING) (default "ct_CF_GRID"))
(multislot Data0 (default ))
....
(multislot Data7 (default ))) (3)
(cn_CF_GRID "No" " Type" "Date" "Rem.Am." "CF Payment" "PV" "Years" "Discount Factor")
(ct_CF_GRID "NUM" "STR" "TIME" "NUM" "NUM" "NUM" "NUM" "NUM")
```

Besides the templates that define the ontology class attributes, the clips program also contains similar templates that define output data, which are most often results from calculations or user actions. They are marked as such and are not stored in the database. For example, such fields contain the present value, duration, dispersion and other analysis results of the currently input financial instrument in an “Instrument” ontology class.

1.2. Facts describing the user interface generated for the ontology class

The description of the UI contains a set of facts each containing a control identifier, control type, coordinates and size of the control as well as the name of the template fact containing the value of the visualized class attribute. They are arranged in a special way, forming a hierarchical structure of dialog boxes, containing several group boxes which in turn contain a set of user interface controls. These facts are declared in the CLIPS program with the special effect construct and are available as soon as the file is loaded into the CLIPS environment. By manipulating connected facts inside the program's rules, the controls can be moved or hidden from view. Although the defined hierarchical structure does not allow nested group boxes, such can be achieved by specifying proper group box coordinates and sizes.

The attribute “InstrumentId” from (1) can be visualized in a text box by defining the following fact in the CLIPS file:

```
(dialogfield
(type "edit")
(x-coord 291) (y-coord 50) (4)
(length 133) (height 12)
(variableID "InstrumentId"))
```

Besides the structural facts, describing dialog boxes and group boxes, the user interface definition can contain facts that do not concern the ontology class and just define static help text. For example, the following fact can be declared to show the text “Instrument ID:” that on mouse hover displays a bubble help specifying the purpose of the edit field following it. In this example it clarifies the edit field used for ID input from the above (4) definition.

```
(dialogtext
(type "StaticText")
(name "Instrument ID:")
(x-coord 215) (y-coord 50) (5)
(length 69) (height 8)
(explain "Identifier of the object of the calculation."))
```

It is also possible to visualize the same class attribute in different user controls. The complex attributes can be visualized in grids, charts or tree controls, while the simple ones can be output in a wide specter of user controls like:

- edit fields – single and multiline;
- combo and list boxes, also custom-drawn, allowing selection of image lists like national flags for currencies;
- embedded web-controls;
- buttons – standard and check-boxes
- date and time pickers;
- scrolls and sliders;

- system-specific controls for editing theoretical distributions.

1.3. Rules for attribute processing

A simplified version of CLIPS rule syntax is:

```
(defrule rule_name
   (Conditional Element-1)
   (Conditional Element-2)                    (6)
          ...
   =>     (actions))
```

If the list of conditional elements in the rule's left-hand side (LHS) is satisfied, the actions in the right-hand side (RHS) are performed. The conditional elements contain wildcard elements that can be used as variables in the RHS. These can be single or multifield values of facts, fact slots or fact addresses as well as a combination of logical operations on them. Following is an example of a rule's LHS. It uses the values of several fact templates representing the simple attributes “Instrument identifier”, “evaluation yield curve” and “produce amortization cash flows” as well as the values stored in a complex fact template, representing a set of custom parameters. Also, the LHS of the rule contains the addresses of two facts – the first one describing the amortization payments of a financial instrument and the second one– a fact used as a flag, meaning that the proper evaluation yield curve is loaded from the database. It is added to the CLIPS knowledge base by a previously fired rule and ensures the proper firing order of the rules on the agenda. The address of the amortization payments table is used by the right-hand side of the rule to modify the fact and fill in its multifields with calculated values.

```
(defrule Calculate_Grids_Monte_Carlo (ProduceAmoCashflow (value ?vProduceCashflow )) (VALUATION_CURVE (str_value ?CurveNameSel )) (InstrumentId(str_value ?vInstrumentId)) (AMO_PARAMETER_GRID (Data1 $?DataAP0) ) (AMO_PARAMETER_GRID (Data1 $?DataAP1) ) (AMO_PARAMETER_GRID(Data2 $?DataAP2) ) (7) (AMO_PARAMETER_GRID (Data3 $?DataAP3) )
?AmoGrid<-(AMORTIZATION_GRID)
?f2<-(CurveReady)
=>      …
```

The right-hand side of the rules consists of a sequence of instructions as in a standard procedural language. Using CLIPS functions, the programmer of the rule can assign new values to the slots describing the ontology class attributes or other output controls. Using the abilities of the CLIPS production system to interface with C++, a set of external functions is added to the CLIPS programming language, enabling the implementation of the builder of ontology-driven information systems first introduced in [7]. Some of the functions are used for:

- loading a new CLIPS file and thus defining instances of another ontology class
- determining the current working parameters of the information system – like the logged user, the unique identifier of the currently defined instance, the current organizational entity which are reviewed in [7];
- accessing the database of stored class instances;
- controlling certain UI elements like progress indicators and message boxes;
- exporting data to XML [11], MS Excel, MS Word, importing data from specific XML and text formats;
- performing financial calculations and setting up, running and receiving the results of a Monte Carlo Simulation.

Figure 1 Algorithm of the IS-builder while processing a CLIPS file program

A flowchart of the algorithm used in the information system builder for processing a CLIPS program is shown on Fig. 1. When the file is loaded in the CLIPS environment, the deffact constructs, which are the default facts in the program, are added to the CLIPS knowledge base automatically. They contain data about the positions of the UI controls and the identifiers of all connected fact templates. The

default facts define static texts if they have no connected identifier, like (5). If they have a fact slot "VariableID" like (4), they define the positions and types of UI controls for visualizing ontology class attributes. Reading of the default facts by the C++ core is the first phase of the IS-builder algorithm the second phase of the algorithm consists of adding the values of the ontology class attributes to the CLIPS knowledge base by asserting facts describing them. If it is a first run of the system on a newly created instance of the class, default values are asserted. Such default values are specified in the fact template slot definition in the "default" property as shown in (1) and (2). To add facts with default values of their slots to the CLIPS knowledge base, one has to assert the template name like:
(assert (InstrumentId))
Otherwise, if there are values stored in the database for the opened instance, they are used for specifying the fact slot values. And if this is not a first run, the values collected from the user interface control inputs are used:
(assert (InstrumentId (value "[stored or input instrument identifier]")))
CLIPS rules are activated after adding the facts describing ontology class attributes to the knowledge base. The rules process the facts in the knowledge base and assert new values to some of the facts. After the rules finish, the knowledge base is scanned by the C++ core and the new values of the facts are visualized in the corresponding user interface controls. The end user of the generated information system can modify some values and start a new calculation or store the defined ontology class instance to the database to be used by other CLIPS programs or for consecutive opening and viewing.

REFERENCES

[1] H. El-Ghalayini, Mohammed Odeh, Richard McClatchey and Tony Solomonides. «Reverse engineering ontology to conceptual data models, » in Proc. Databases and Applications. 2005, pp. 222-227.
[2] F. Fonseca and J. Martin. «Learning the differences between ontologies and conceptual schemas through ontology-driven information systems, » Journal of the Association for Information Systems, vol. 8, 2007
[3] T. Gruber. «Ontology, » in the Encyclopedia of Database Systems, Ling Liu and M.Tamer Özsu,Ed.New York: Springer-Verlag, 2009, pp. 1963-1965
[4] N. Guarino. «Formal ontology and information systems, » in Proc. FOIS'98, 1998, pp. 3-15
[5] P. Hoefferer. «Achieving business process model interoperabilityusing metamodels and ontologies, » In Proc. ECIS, 2007
[6] M. Jarrar, J. Demey, R. Meersman. «On using conceptual data modeling for ontology engineering, » Journal on Data Semantics, pp.185-207, Oct. 2003
[7] S. Nikolov and A. Antonov. «Framework for building ontology-based dynamic applications, » in Proc. CompSysTech, 2010, pp. 83-88
[8] S. Nikolov. "Storing data of ontology-based dynamic applications," in Proc. Sixth International Scientific Conference Computer Science, 2011, pp. 134-139
[9] M. Scott. Programming language pragmatics, Morgan Kaufmann, 2006
[10] P. Spyns, R Meersman and M. Jarrar. «Data modelling versus ontology engineering, » ACM SIGMOD vol. 31, Dec. 2002, pp. 12-17
[11] S. Trifonova and S. Nikolov. „XML presentation of financial instruments, "in Proc. Unitech, 2010
[12] Y. Wand and R. Weber. «Research commentary: information systems and conceptual modeling-a research agenda» Info.Sys.Research, Dec.2002, pp.363-376
[13] Y. Wand and R. Weber. "An ontological model of an information system," IEEE transactions on Software Engineering, vol. 16, 1990
[14] B. Yildiz and S. Miksch. «Ontology-driven information systems: challenges and requirements» in Intl. Conf. on Semantic Web and Digital Libraries, 2007

ABOUT THE AUTHOR:

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