Ontological modeling using the system «Binary Model of Knowledge»

Gerald S. Plesniewicz
National Research University MPEI
Moscow, Russia
salve777@mail.ru

Dmitry E. Masherov
National Research University MPEI
Moscow, Russia
masherovdy@mail.ru

***Abstract*—In the paper, a brief description of the ontological modeling system «Binary Model of Knowledge» (BMK) is given. Using the system, experts have an opportunity to create heavy-weight ontologies for complex systems. A case study of applying BMK to development problems of computer systems for railway safety is presented.**

***Keywords*—ontological modeling; knowledge bases; highweight ontologies; railway safety systems**

I. Introduction

An ontology aims at representing knowledge (at conceptual level) of the problem domain and the functionality of a given modelled system.

«Binary Model of Knowledge» (BMK) is an ontological modeling system which is under development at National Research University MPEI (Moscow) and at Institute of Information and Computing Technologies of MES RK (Almaty).

BMK is supported by concept-type languages which provide a friendly interface for experts who are busy in developing ontologies.

In Section 1 we describe briefly the main languages of the system «Binary Model of Knowledge». In Section 2 we present the case study of ontological modeling related to the problem of developing a computer railway safety system.

II. Brief description of the ontological modeling system «Binary Model of Knowledge»

Ontological modeling is an activity in creating ontologies for subject domain and functionalities of applications. For ontological modeling, the methods and methodology of ontological engineering are used [2, 3, 5].

BMK is the ontological modeling system supported by conceptual-type languages which are well readable and friendly to experts-ontologists.

Semantics of BMK languages is based on the formal model of concept [3]. Formal concepts are constructed from names. A *formal concept* with the name C has the following components: (1) set U^C – the *universe* of the concept; (2) set Γ of so-called *points-of-reference*; (3) set $E^C\gamma$ for each $\gamma \in \Gamma$ – the set of *instances* of the concept at point-of-reference γ; (4) reflexive, symmetric and transitive relation ${}^{\sim C}\gamma \subseteq E^C\gamma \times E^C\gamma$ for each $\gamma \in \Gamma$ – *coreferentiality* relation; (5) pair $(E^C\gamma, {}^{\sim C}\gamma) \quad = Ext^C\gamma$ for each $\gamma \in \Gamma$ – the *extension* of the concept at point-of-reference γ; (6) set family $\{(E^C\gamma, {}^{\sim C}\gamma) \mid \gamma \in \Gamma\}$ – the *full extension* of the concept.

T. Gruber define an ontology as «explicit specification of conceptualization» [2]. A conceptualiz/ation includes a choice of suitable concepts and relations between them. (Let us note that the relations can be considered also as concepts.) We define a *formal conceptualization* as a (finite) set of formal concepts with the same set Γ of points-of-reference.

For formal specification of formal concepts suitable formal languages are used. There are different types of sentences in such languages, depended on what components of formal concept they specify. *Structural sentences* specify concept universes; *logical sentences* specify concept extensions; *transitional sentences* specify modifications of concept extensions.

BMK includes the language **LS** for specifying ontologies and its sublanguages **LS-S**, **LS-L**, **LS-T** and **LS-DT** which consist in structural, logical, transitional sentences, and data type specifications (correspondingly). In **LS**, two kinds of concepts are used: *classes* and *binary links* (*relations*).

A. Structural sentences

Elementary **LS-S** sentences are expressions of the forms:

$$C[E], C[A:E], C[A:T], (C\ L\ D), (C\ L\ D)[E], \\ (C\ L\ D)[A:E], (C\ L\ D)[A:T]. \qquad (1)$$

Here C, D, E are names of classes, L is a name of a binary link, A is an attribute name, and T is a data type specification.

C and $(C\ L\ D)$ in elementary structural sentences (1) are their *heads*, and $[E]$, $[A{:}E]$, $C[A{:}T]$ are their *tails.* Several elementary statements with identical heads can be merged into one statement joining their tails. For example, the elementary statements $C[E]$, $C[A{:}D(*)]$, $C[B{:}(\text{String,Integer})]$ are merged into one statement $C[E,A{:}D(*), B{:}(\text{String,Integer})]$.

A *structural scheme* is a (finite) set of structural sentences.

Example 1.

```
SCHEME EducationInfo
1. Student[Name:String,
        Belongs_to:StudGroup].
2. StudGroup[Title:String,Tutor:Student].
3. (Student PassedExam Course).
4. PassedExam[ExamList].
```

```
5. Teacher[Name:String,
        Position:{prof,assist},
        Works_at:Department].
6. (Course Conducted_by Teacher).
7. Course[Name:String].
8. Department[Name:String,Staff:
          (Teacher OR Ingineer)(*)]
9. ExamList[Name-of-course:String,
         Name-of-student:Dtring,
         Date-of-exam:Date].
END
```

The statements from (1) define (correspondingly) the following universes of C and L:

U^C = Surr U Iname U {[E:x] | x ∈ Surr},
U^C = Surr U Iname U {[A:x] | x ∈ Surr},
U^C = Surr U Iname U {[A:x] | x ∈ T},
U^L = Surr U Iname U{[C:x, D:y] | x, y ∈ Surr},
U^L = Surr U Iname U {[C:x, D:y, E:z] | x, y, z ∈ Surr},
U^L = Surr U Iname U {[C:x,D:y,A:z] | x, y, z ∈ Surr},
U^L = Surr U Iname U {[C:x, D:y, A:z] | x, y ∈ Surr, z ∈ T}.

Here Surr is the set $\{\#1, \#2, \#3, \ldots\}$ of so-called *surrogates* (object identifiers) and Iname is the set of object names.

For example, the following tuple belongs to $U^{Teacher}$, where $\#27$ is the surrogate of some department:
e = [Name:V.Falk',Position:prof, Work_at:#27].

In a structural scheme, every concept from a structural sentence with a tail defines derived concepts. For example, for the concept Teacher we have the derived concepts

```
  Teacher(Name = 'V.Falk'),
  Teacher(Name = 'V.Falk; Positon =
prof),
  Teacher(Name=prof;Work_at = #27),
  Teacher(Work_at.Name = AppliedMath)
```

and so on. The first term denotes the teacher V. Falk. The second term denotes the teacher V. Falk which is a professor. The third term denoes set of professors working an the department with the surrogate $\#27$. The third term denotes the set of all teachers working at the Applied Mathematics department.

B. Representation of facts

A *fact* is a statement about belonging a given object e to instances of a given concept C at a given point-of-reference γ, i.e. $e \in E^C_\gamma$ or $e \in U^C \setminus E^C_\gamma$. The first fact is *positive* and the second fact is *negative.* In the language **LS**, the facts are written as follows:

- $+(e$ IN C / POR $\gamma)$ and –(e IN C / POR $\gamma)$ when C is a class. Here POR is the abbreviation for «point-of-reference»;
- $+(e_1\ L\ e_2\ C$ / POR $\gamma)$ and –($e_1\ L\ e_2\ C$ / POR $\gamma)$ when L.

Here POR is the abbreviation for «point-of-reference», and the signs «+» and «–» denote that fact are positive and negative.

In the BMK, a tabular representation of facts data representing first sort facts is used.

Example 2. Consider the sentences 3 from Example 1 for the binary relation PassedExam. Table I is an example of a tabular representation of a set of facts for this relation.

Table I
TABULAR REPRESENTATION OF THE BINARY RELATION «PASSEDEXAM»

PassedExam				
Surr	Sign	Coref	Student	Course
.....	....	.....	.....	.....
#33	+	[exam4]	#10	#24
#34	-	[]	#10	#27
.....	....	.....	.....	.....

The rows from Tables I represent the facts: «The student with the surrogate #10 passed the exam on the course with the surrogate #24, and does not pass the exam with the surrogate #27». Also here we have the coreferentialities:
#33 ~exam4 ~[Student:#10, Course:#24],
The language **LS** includes conjunctive queries.

Example 3.
Query 2. What students from the student's group A13-09 passed examination in algebra at 15-01-2017 and received the assessment 75?

```
?X.Name --(X Passed_exam:Y Course);
    X.Belongs_to.Title = A13-09;
    Y.ExamList.Name-of-course = algebra;
    Y.ExamList.Data = 15-01-2017;
    Y.ExamInfo.Assesment = 4.
```

C. Logical sentences

Logical sentences of the language **LS-L** are constructed of three type terms: C-*terms*, L-*terms* and P-*terms*. C-*terms* denote subsets of the set Surr, L-*terms* denote subsets of the set Surr × Surr, and P-*terms* denote unary predicates defined on Surr.

Consider several examples of C-*terms*, L-*terms*, P-*terms* and **LS-L** sentences.

Examples 4.

1) Student, Student(Name = 'A. Kotov), Student(Belongs_to.Title = A13-11), Belong_to.Name = A13-11) OT PassedExam SOME Course,
2) PassedExam SOME Course THAT Conducted_by Teacher(Name='V.Falk'),
3) Course(Name=CraphTheory)Conducted_by.

The first P-*term* presents the predicate which is true for every student that did not pass at least one exam. The third P-term presents the predicate which is true for all teachers who conducted the course «graph theory».

We use the following metavariables in the description of syntax and semantics of terms and sentences of the language **LS-L**: c for individual constant; C, D, E for classes; L, M, N for binary links; P, Q, R for predicates; S, S_1, S_2 for sentences; V for parameters.

For any expression *exp*, we denote by $\|exp\|$ its value under a given interpretation. For concept and binary link names we suppose $\|C\| \subseteq$ `Surr` and $\|L\| \subseteq$ `(Surr, Surr)`.

Syntax of C-terms:

C, D::= `NOT` C | C : V | (C `AND` D) | (C `OR` D) | C `THAT` P | C `SOME` L | C `ONLY` L | $L.1$ | $L.2$.

Here $L.1 = D_1$ and $L.2 = D_2$ where (D_1 L D_2) specifies L.

Semantics of C-terms:

$\|$ `NOT` $C\| = $ `Surr` $\setminus \|C\|$,
$\|C : V\| = \|C\|$,
$\|C_1$ `AND` $C_2\| = \|C_1\| \cap \|C_2\|$,
$\|C_1$ `OR` $C_2\| = \|C_1\| \cup \|C_2\|$,
$\|C$ `THAT` $P\| = \{x \in \|C\| \mid \|P\|(X)\ \}$,
$\|C$ `SOME` $L\| = \{x \in$ `SurrR` $\mid \exists y \in$ `Surr`.
$Y \in \|C\| \wedge (x,y) \in \|L\|\ \}$,
$\|C$ `ONLY` $L\| = \{x \in$ `Surr` $\mid \forall y \in$ `Surr`.$(x,y) \in \|L\| \to y \in \|C\|\ \}$,
$\|L.1\| = \|D_1\|$,
$\|\ L.2\ \| = \|\ D_2\ \|$.

Syntax of L-terms:

L::= `NOT` L | (L_1 `AND` L_2) | (L_1 `OR` L_2) | `INV` (L).

Semantics of L-terms:

$\|$ `NOT` $L\| =$ `(Surr, Surr)` $\setminus \|C\|$,
$\|L_1$ `AND` $L_2\| = \|L_1\| \cap \|L_2\|$,
$\|L_1$ `OR` $L_2\| = \|L_1\| \cup \|L_2\|$,
$\|$ `INV` $(L)\| = \{(y,x) \mid (x,y) \in \|L\|\ \}$.

Syntax of P-terms:

P ::= `NOT` P | (P_1 `AND` P_2) | (P_1 `OR` P_2) | L `SOME` C | L `EACH` C | (L V) | (L c).

Semantics of P-terms:

$\|$ `NOT` $P\|(x) \Leftrightarrow \neg\|P\|(x))$,
$\|P_1$ `AND` $P_2\|(x) \Leftrightarrow \|P_1\|(x) \wedge \|P_2\|(x)$,
$\|P_1$ `OR` $P_2\|(x) \Leftrightarrow \|P_1\|(x) \vee \|P_2\|(x)$,
$\|L$ `SOME` $C\|(x) \Leftrightarrow \exists y \in \|C\|.(x,y) \in \|L\|$,
$\|L$ `EACH` $C\|(x) \Leftrightarrow \forall y \in$ `Surr`$.(x,y) \in \|L\| \to y \in \|C\|$,
$\|(L\ V)\|(x) \Leftrightarrow (x,V) \in \|L\|$,
$\|(L\ c)\|(x) \Leftrightarrow (x, \|c\|) \in \|L\|$.

Syntax of sentences:

S ::= `NOT` S | S_1 `AND` S_2 | S_1 `OR` S_2 | S_1 `IMP` S_2 | `EACH` C P | `FOR-EACH` C P | `FOR-SOME` C P | C_1 `ISA` C_2 | $C_1 = C_2$ | $L_1 = L_2$ | `EXIST` C | `EXIST` L | `NULL` C | `NULL` L. | `PRECOND` ($\star$ S $*$) | `POSTCOND` ($\star$ S $*$).

Semantics of sentences:

$\|$ `NOT` $S\| \Leftrightarrow \neg\|S\|$,
$\|S_1$ `AND` $S_2\| \Leftrightarrow \|S_1\| \wedge \|S_2\|$,
$\|S_1$ `OR` $S_2\| \Leftrightarrow \|S_1 \vee \|S_2\|$,
$\|$ `EACH` C $P\| \Leftrightarrow \|$ `FOR-EACH` C $P\| \Leftrightarrow \forall y \in \|C\|.\|P\|$,
$\|$ `SOME` C $P\| \Leftrightarrow \|$ `FOR-SOME` C $P\| \Leftrightarrow \exists y \in \|C\|.\|P\|$,
$\|C_1$ `ISA` $C_2\| \Leftrightarrow \|C_1\| \subseteq \|C_2\|$,
$\|C_1 = C_2\| \Leftrightarrow \|C_1\| = \|C_2\|$,
$\|L_1 = L_2\| \Leftrightarrow \|L_1\| = \|L_2\|$,
$\|$ `EXIST` $C\| \Leftrightarrow \|C\| \neq \emptyset$,
$\|$ `EXIST` $L\| \Leftrightarrow \|L\| \neq \emptyset$,
$\|$ `NULL` $C\| \Leftrightarrow \|C\| = \emptyset$,
$\|$ `NULL` $L\| \Leftrightarrow \|L\| = \emptyset$,
$\|$ `PRECOND` ($\star$ S $*$)$\| = \|S\|$,
$\|$ `PRECOND` ($\star$ S $*$)$\| = \|S\|$.

D. Specification of changes

In **LS** we can create schemas for determining dynamics of fact bases. The dynamic aspects include:

- the operations that are possible;
- the relationships between their inputs and outputs;
- the changes of fact bases that happen.

Examples 5. An examination list can be considered as an object changing its content during examination. We describe it by means the following schemes.

```
SCHEME ExamList
1. Records SUBS (Name, {20 .. 100})(*).
2. assess: Name --> {20..100}.
3. FOR-EACH X IN RecordsedNames EXIST
   Y IN {20..100} THAT assess(X) = Y.
END
```

Here `SUBS` is the abbreviation for «subset» and « –> » informs that `assess` is a function, possibly partial. The sentence 3 says that the names of students are recorded together with assessments.

```
SCHEME AddRecord
1. ExamList.
2. name?:Name.
3. assessment?: {20..100}.
4. PRECOND(*name? NOT IN Records*).
5. Records : =  Records ADD
                          (name?,assessment?)
END
```

The names with question marks are used as input variables, So, when the specific values `a` and `b` are assigned to the variables `name?` and `assessment?` then the pair (a,b) is added to `Records`. The sentence 4 act as precondition for executing 5.

```
SCHEME FindAssesment
1. name?: Name.
2. assessment!: {20..100}.
3. PRECOND (*name? IN Records*).
4. assesment! = assess(name?).
END
```

The exclamation mark informs that `assessment!` is an output variable. Thus, the scheme `FindAssesment` acts as a query.

III. Railway safety control system

A distributed railway control system (RCS) is a critical safety system. RCS consists of:

- switch boxes (SB), each one locally controlling a point, i.e. the boundary between two segments of a single track or a railway crossing,
- train control computers (TCC) residing in the train engines and collecting the local state information from switch boxes along the track in order to derive the decision whether the train may enter the next track segment [4].

We introduce a formal approach for domain specification and decision inference using BMK schemas with some constucts of Z-notation [9,10]. The presented approach allows agents to make decisions to signal trains and update the control system.

Consider the system configuration depicted in Fig. 1. The tasks of train control and interlocking are distributed on computers residing in each train $t1$, $t2$ and switch boxes $sb1$, $sb2$, each one controlling a single point, the boundary between two segments of a single track or a railway crossing.

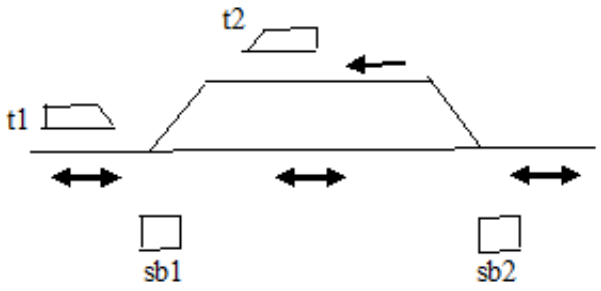

Figure 1. Example of a network with trains

The system TCC acts as follows:

- Each switch box stores the local safety-related information. In particular, this information includes the state of the traffic lights guarding the railway crossing (i.e., the track segments that are currently connected by the local point, or whether a train is approaching the switch box).
- In order to pass a railway crossing or to enter a new track, a train's TCC communicates with the relevant switch boxes to make a request for blocking a crossing, switching a point, or just reserving the relevant track segments at the SB for the train to pass.
- Depending on their local state, the switch boxes may or may not comply with the request received from a TCC. In any case, each SB returns its local state information to the requesting TCC. After having collected the response from each relevant SB, the TCC evaluates the SB states to decide whether it is safe to approach the crossing or to enter the next track segment.

We implemented a model to describe RCS. Specification schemas are used to define the domain concepts and operator schemas to enforce the train system safety. The definitions are split into the static and dynamic parts. The static part defines classes which are not changed during the system operation and the dynamic defines classes with changing states. The static part corresponds to the `TrainDefinitions` schema and the dynamic to the `TrainSystem` schema.

```
SCHEME TrainDefinitions
    RouteSegments SUBS
        (Segment,{'<--','-->','<->'})(*).
    Arcs SUBS(RouteSegments,RoutSegments).
    Network == SUBS (RoutSegments,Arcs).
    Train[Network,SwitchBlok(*),Lock(*),
    /   Direction: ,{'<--','-->'},
        Reservation(*)].
        Reservation: SwitchBlock -->
                              SwitchBlock(*)].
END

SCHEME TrainSystem
    TrainParameters[
    speed: Train --> Speed,
    envelope: Train -->
                    (Points, Points)(*),
    segment: Train --> Segment,
    direction: Train --> {'-->', '<--'}].
    TYPE TrainSignals == {ContinueGreenZone,
    StopTrainRedZone,IncreaseSpeed,
    ContinueCrossing, StopForCrossing} .
    TYPE BlockStatus = {Free,Occupied,Clear,
    RedZoneForEveryTrain}.
    ControlRoom [
    sectorStatus: BlockStatus,
    crossingAreaStatus: BlockStatus,
    ///sectorOccupiedBy: Train(*),
    segmentsOccupied: Segment(*),
    segmentOccupiedBy: Train(*)s,
    directionOfRedZone: Direction(*)s,
    timeInSector:(Sectors,Train) <-> Time,
    signal: Train <-> TrainSignals
    ].
END.
```

The class `TrainParameters` stores current state of `TrainSystem` which includes

- speed (type Speed is defined as an alias of integers);
- train «envelopes». A train envelope is a neighborhood of the train depending on the train speed;
- segments which are currently occupied by trains train directions.

The class `ControlRoom` stores railway sector information:

- `sectorStatus` and `crossingAreaStatus` are statuses of a sector and a crossing which are inferred with train system state;
- a set of trains occupying the sector `sectorOccupiedBy`;
- a set of segments occupied by any train from `segmentsOccupied`;
- helper variable for storing the train on a segment from `segmentOccupiedBy`;
- helper variable for storing the direction of train on an unsafe segment from `directionOfRedZone`;
- time of trains in sector `timeInSector`.

The following operator schemas are similar to Z-notation schemas: the input of schemas is a set of variables which can be used by the schemas to update the state and to output other variables [10]. Operator schemas can use classes, attributes and types from other schemas using *schema importing*. To import a schema, its name and an import annotation must be explicitly written at the start of the current schema definition. An import annotation specifies whether the current schema modifies the state of an imported schema. Import annotations use symbols «=» and « <> » where « = SN » means that that current schema does not modify the schema SN and « <> SN » means that the schema SN is modified.

Like Z-notation schemas input variables in BKM schemas use «?» suffix and output variables use «!». Variable names can match any attribute name from the imported schemas. If names match and «=» import annotation was used, then both the variable and the attribute change when they are assigned to a value. If « <> » import notation was used the attribute is left unmodified.

`LinearSafety` and `CrossingSafety` schemas set the train signals to enforce safety on linear segments and at railway crossings.

```
SCHEME LinearSafety
    =TrainSystem
    <> ControlRoom,
    train?: Train,
    sector?: Sectors,
    directionOfRedZone!: Directions,
    sectorStatus!: BlockStatus
    PRECOND
    segmentSectors(segment(train?))
    sector?
    (*FOR-EACH
    other:Trains EMPTY(envelope(other)
    AND envelope(train?)sectorStatus':=
                                        Free
    signal'(train?) := ContinueGreenZone
    *)
    (*EXIST other:Trains THAT
      NOT_EMPTY(envelope(other) AND
      envelope(train?))
      timeInSector(sector?, train?) >
      timeInSector(sector?, other?)
      AND speed(train?) < speed(other)
      sectorStatus' := Occupied
      segmentOccupiedBy := other
      signal'(other) := StopTrainRedZone
      signal'(train?) := IncreaseSpeed
      directionOfRedZone! :=
      direction(other)
    *)
    (*EXIST other:Trains THAT
      NOT_EMPTY(envelope(other) AND
      envelope(train?))
      timeInSector(sector?,train?) <
      timeInSector(sector?, other?)
      AND speed(train?) -> speed(other)
      sectorStatus' := Occupied
      segmentOccupiedBy' := other
      signal'(other) := IncreaseSpeed
      signal'(train?) := StopTrainRedZone
      directionOfRedZone!:=
      / direction(other)
    *)
END
```

`LinearSafety` schema receives input variables `train?` and `sector?`. The precondition checks that the `train?` is in the `sector?`. If the precondition fails, then the schema returns a corresponding message, otherwise the schema infers train signals. If there are no other trains near the `train?`, then it is reported that the sector is free. If there is other train behind the `train?`, then the `train?` is signaled to accelerate and the other is signaled to stop. If there is other train in front of the `train?`, then the signals are produced for the corresponding trains.

```
SCHEME CrossingSafety
    =TrainSystem,
    <> ControlRoom
    crossing?: Crossing(*)
    train?: Trains
    crossingAreaStatus!:BlockStatus
    PRECOND
    segment(train?) IN
            crossingSegments(crossing?)
    (*EXIST other:Trains THAT
      NOT_EMPTY(envelope(other) AND
      envelope(train?)) AND EXIST
      envelopeSegment:
      pointSegments(envelope(train?))
      envelopeSegment==crossingSegments
```

```
      (crossing?)
      crossingAreaStatus! :=
                    RedZoneForEveryTrain
      signal'(train?) := StopForCrossing
    *)
    (*
      EXIST other:Trains THAT
      EMPTY(envelope(other) AND
      envelope(train?)) AND
      FOR-ALL envelopeSegment:
      pointSegments(envelope(train?))
      envelopeSegment ==
      crossingSegments(crossing?)
      crossingAreaStatus! := Clear
      signal'(train?) := ContinueCrossing
    *)
      *No train near the crossing*
END
```

`CrossingSafety` schema receives input variables `train?` and the state of the `crossing?`. The precondition checks that the `train?` is near the `crossing?`. If the precondition fails, then the schema returns a corresponding message, otherwise the schema infers train signals. If there is other train near the train and crossing, then the `train?` is signaled to continue, otherwise it is signaled to stop. Besides signaling the train system is updated.

IV. Conclusion

We gave a brief description of the ontological modeling system «Binary Model of Knowledge» inteded for designing and interpreting heavyweight ontologies. Such ontologies are needed when describing problem domains for advanced applications. We present a case study of modeling the railway safety control system. We can analyze the resulting ontology using the reasoning block of the system «Binary Model of Knowledge». The case study demonstrates adequacy of the system «Binary Model of Knowledge» for designing and interpreting heavyweight ontologies for complex systems.

Acknowledgment

This work was supported by Russian Foundation for Basic Research (project 14-07-0387) and Ministry of Education and Science of Kazakhstan (project 0115 RK 00532).

References

[1] T. R. Gruber. “Toward principles for the design of ontologies used for knowledge sharing.”. International Journal of Human-Computer Studies, Vol. 43, Issues 4-5, pp. 907-928, November 1995.

[2] V. Devedzic. “Undestanding ontological engineering.” Communications of the ACM, Vo.l. 45, Issue 4, pp. 136-149, 2002.

[3] T. Halpin. “Ontological modeling: Part 1”. Available in: http://www.orm.net/pdf/OntologicalModeling1.pdf

[4] A.E. Haxthausen, J. Peleska. "Formal developing and verification of a distributed railway control system." IEEE Transaction on Software Engineering, 26 (8) , pp. 687-701.

[5] R. Mizoguchi. “Tutorial to ontological engineering”. Available in: https://ru.scribd.com/document/56477570/Mizoguchi-2004-Tutorial-on-Ontological-Engineering

[6] G.S. Plesniewicz. “Formal ontologies”. Open semantic technolologies for intelligent systems (OSTIS 2012). – Minsk, 2012, pp. 163-168. (In Russian.)

[7] G.S. Plesniewicz., D.E. Masherov, Nguen Thi Min Vu, A.B. Karabekov. “Binary Knowledge Model: specifying, instantiating and interpreting advanced ontologies.” Proceedings of the 9th International Conference on Application of Information and Communication Technologies. – Rostov-on-Don, Russia, 2015, pp.314-319. (In Russian.)

[8] Plesniewicz., D.E. Masherov, Nguen Thi Min, A.B. Karabekov. Methods and Languages for Ontological Modeling. – Almaty: IICT publ., 2015, p. 179. (In Russian.)

[9] J.M. Spivey. The Z notation: a reference manual. – Oxford (UK): Prentice Hall Int., 1989, p. 158.

[10] J. Woodcock, J/ Davis. “Uzing Z.” Available in: http://www.cs.cmu.edu/ 15819/zedbook.pdf

ОНТОЛОГИЧЕСКОЕ МОДЕЛИРОВАНИЕ С СИСТЕМОЙ «БИНАРНАЯ МОДЕЛЬ ЗНАНИЙ»

Г.С. Плесневич, Д.Е.Машеров

В статье дано краткое описание системы онтологического моделироваия «Бинарная Модель Знаний» (БМЗ). Используя эту систему, эксперты могут создавать «весомые» (heavyweight) онтологии для сложных приложений. Рассмотрен пример применения БМЗ к задачам разработки компьютерных систем безопасности железнодорожного движения.