UDK 004.822, 681.51

Constructing Verification-Oriented Domain-Specific Process Ontologies*

Natalia O. Garanina (A.P. Ershov Institute of Informatics Systems, Institute of Automation and Electrometry)

Igor S. Anureev (A.P. Ershov Institute of Informatics Systems, Institute of Automation and Electrometry)

Vladimir E. Zyubin (Institute of Automation and Electrometry, Novosibirsk State University)

User-friendly formal specification and verification of concurrent systems from various subject domains are active research topics due to their practical significance. In this paper, we present the method for development of verification-oriented domain-specific process ontologies which are used to describe concurrent systems of subject domains. One of advantages of such ontologies is their formal semantics which makes possible formal verification of described systems. Our method is based on the verification-oriented process ontology. For constructing a domain-specific process ontology, our method uses techniques of semantic markup and pattern matching to associate domain-specific concepts with classes of the process ontology. We give detailed ontological specifications of these techniques. Our method is illustrated by the example of developing a domain-specific ontology for typical elements of automatic control systems.

Keywords: *process ontology, pattern matching, semantic markup, automatic control system, formal verification*

1. Introduction

Our long-term goal is a comprehensive approach to supporting formal verification of concurrent systems for ensuring their quality by formal methods. The solution includes methods for extracting formal models and properties of concurrent systems from the texts of technical documentation, as well as, instruments for manual correction of the extracted information and enriching it with new entities.

*The research has been supported by the Russian Ministry of Education and Science and the Russian Foundation for Basic Research (grants 17-07-01600, 19-07-00762).

Our envisaged intellectual system for supporting formal verification of concurrent systems will automatically extract and generate system requirements. We developed an Ontology of Specification Patterns as a first step towards creating this system [1]. Another key component of the system is the Process Ontology for concurrent systems [2]. The content of these ontologies, i.e. the sets of instances of their classes, are ontological descriptions of some concurrent system and requirements for it. These descriptions can be extracted from corpus of technical documentation by our system of information extraction from natural language text [3–5]. Such descriptions also can be developed by special editors which also can be used for correction of extracted information. These ontological descriptions for concurrent system processes and requirements are the basis for formal verification of the concurrent system because the Ontology of Specification Patterns and the Process Ontology have formal semantics. To verify a system, it is necessary first to choose a suitable verifier (model checker, in particular) taking into account the formal semantics of the ontology-based requirement presentation. If it exists, we translate the ontological description of the system into the model specification input language of the chosen verifier, and the requirements' description is translated into the property specification input language of this verifier (usually, this language is some temporal logic). Dealing with requirements in our system involves not only the formal semantics of specification patterns, but also the presentation of requirements both in a natural language and in a graphical form.

In this paper, we address both the problem of extracting a concurrent system description from technical documentation and developing editor for constructing and correcting the ontological description of concurrent systems. These tasks use the Process Ontology, which describes concurrent systems as consisting of communicating concurrent processes characterized by local and shared variables, and channels for communication by messages. This ontology has formal semantics based on labelled transition systems [2]. However, for requirement and verification engineers, the Process Ontology is very abstract to be suitable for supporting formal verification with our system.

Since this support system can be used for different subject domains, it is necessary to develop a method to specialize our abstract Process Ontology for specific subject domains in order to construct domain-specific processes instances which have variables and channels corresponding to their subject specialization. For example, in a concurrent system from the domain of Automatic Control System, the sensor-process must necessarily be connected by at least one communication channel with the process-controller. We must construct a Domain-

Specific Process Ontology to be a special case of the Process Ontology. Hence, this new ontology has formal semantics which makes possible formal verification of the systems it describes.

A Domain-Specific Process Ontology differs from the original Process Ontology in a set of axioms and rules that specify domain-specific restrictions on the attributes of the Process Ontology classes. This set of axioms has a declarative character. Ontology axioms can be used to check integrity and consistency of the ontology content. In case of the ontological concurrent system representation, integrity and consistency mean that instances of the ontology processes corresponding to processes of the subject domain have all necessary variables, channels and actions.

The declarative aspect of an ontology of domain-specific processes is suitable for checking the correctness of descriptions of already created or extracted concurrent systems. But for creating or correcting such a system, a constructive approach based on patterns of domain-specific processes is better. In this paper, we propose the method of constructing the domain-specific content of the Process Ontology using domain-specific patterns. The construction of this content includes several steps. First, we enrich classes of the Process Ontology (Section 2) with semantic markup attributes containing a string description of terms from a subject domain. The resulting new ontology called Semantically-Marked Process Ontology (Section 3) allows us to construct the domain-specific content of the Process Ontology. Then, patterns of domain-oriented processes are defined as instances of the Process-Oriented Semantic-Markup Patterns Ontology (Section 4). We illustrate our method with the example from the subject domain of Automatic Control Systems (Section 5). Development of the process ontology for this domain is especially important because a user-friendly formal specification and verification of automatic control systems, and, in general, cyber-physical systems have crucial practical significance.

2. Process Ontology

We consider an *ontology* as a structure, which includes the following elements: (1) a finite non-empty set of *classes*, (2) a finite non-empty set of *data attributes and relation attributes*, and (3) a finite non-empty set of *domains of data attributes*. Each class is defined by a set of attributes. Data attributes take values from domains, and relation attributes' values are instances of classes. *An instance of a class* is defined by a set of attribute values for this class. *A content* of an ontology is a set of instances of its classes.

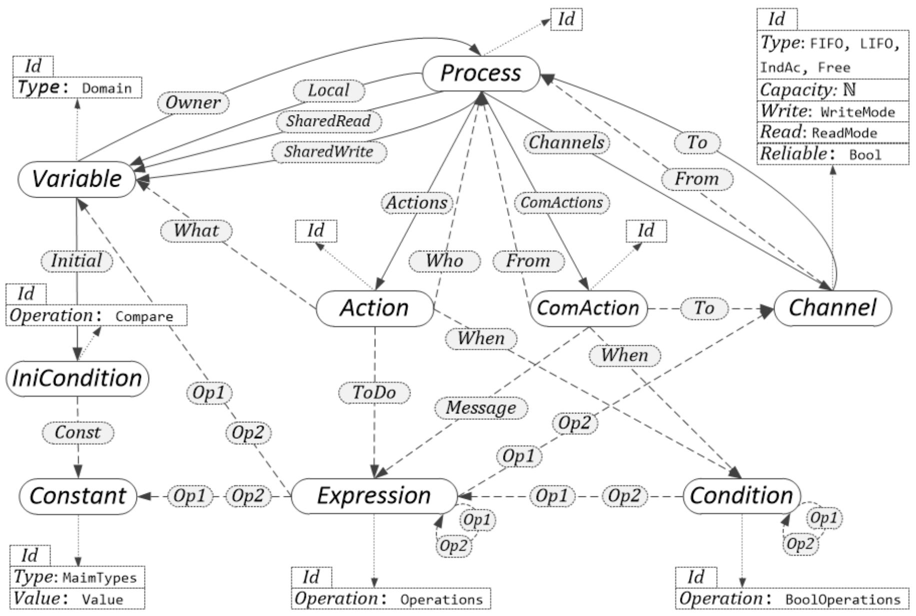

Fig. 1. Process Ontology.

The Process Ontology PO provides an ontological description of a concurrent system by a set of its instances. We consider a concurrent system as a set of communicating processes. Processes (described by the class Process) are characterized by sets of local and shared variables; a list of actions on these variables which change their values; a list of channels for the process communication; and a list of communication actions for sending messages. The process variables (the class $Variable$) and constants (the class $Constant$) take values in domains of basic types (Booleans, finite subsets of integers or strings for enumeration types) and finite derived types. Initial conditions of the variable values can be defined by comparison with constants. The actions of the processes (the class *Action*) include operations over variables' values. The enable condition for each action is a guard condition (the class $Condition$) for the variable values and the content of the channels. The processes can send messages through channels (the class $Channel$) under the guard conditions (the class $Condition$). The communication channels are characterized by the type of reading messages, capacity, modes of writing and reading, and reliability. Figure 1 represents the Process Ontology. Classes are presented by white ovals. Relations between classes are shown as dashed arrows with names in grey ovals. These arrows are solid if the relation is one-to-many, and dotted, if the relation is one-to-one. Class data attributes placed in dash-dot rectangles are connected with their classes by dash-dot arrows.

Classes of PO are universal because they do not take into account the features of a subject domain. In the next section, we define an extension of ontology PO — a semantically-marked process ontology that specifies necessary information about the subject domain.

3. Semantically-Marked Process Ontology

In this section, we formally describe our method of the semantic markup of the Process Ontology. This markup is used for matching the abstract processes of PO to specific processes of a chosen subject domain. The marking up is performed by enriching the classes of ontology PO with string labels corresponding to the concept of the subject domain. This classes with several service classes form the new semantically-marked process ontology. The instances of the subject domain processes can be constructed using this new ontology and the Process-Oriented Semantic-Markup Patterns Ontology described in the next section,

The *semantically-marked process ontology* ($SMPO$) contains domains $Classes$, $Domains$, $Types$, $Values$ corresponding to elements of PO ontology, domains $SLabel$, $SAttribute$, classes $AValue$, $Element$ and $Element\_T$ ($T \in Domains$) corresponding to semantic labelling, and classes of ontology PO enriched with semantic attributes based on listed new domains and classes.

Domains $Classes$ and $Domains$ include names of classes and domains from ontology PO. Domain $Types = Classes \cup Domains$ includes all names from ontology PO. Domain $Values$ includes all attribute values of PO: $Values = \cup_{T \in Types} Val(T)$, where $Val(T)$ is values of T, which are instances for $T \in Classes$ and the corresponding values for $T \in Domains$.

Domain $SLabel$ is a finite set of semantic labels which are strings. String labels specify information associated with the attribute values of ontology PO. This information can be about a subject domain (ex., "sensor" or "pressure") or special features of modeling processes (ex., "periodic start").

Domain $SAttribute$ is a finite set of semantic attributes which are string. Like labels, these semantic attributes specify subject domain information associated with the attribute values of ontology PO. The difference is that strings of the semantic attributes must be a string description of the attribute values of ontology PO (ex., "100", "$true$" or "instance of class $Controller$").

Further in class definitions, we add a superscript $*$ for multi-valued attributes and superscript 1 for mandatory single-valued attributes. Class $AValue$ (which instances are called attribute values) has two single-valued attributes: $Attribute^1$ with values in $SAttribute$ and $Value^1$ with values in $Values$ which specify the name of a semantic attribute and its value.

Class T of ontology $SMPO$ is some class of ontology PO enriched with two attributes: $SLabels^*$ with values in $SLabel$ and $SAattributes^*$ with values in $AValue$ (called *marking attributes*) which add the semantic markup to instances of class T. This attributes connect abstract notions of Process Ontology with a chosen subject domain. Attribute $SLabels$ specifies a set of semantic labels. Attribute $SAttributes$ specifies a set of semantic attributes with their values. Attribute $SAttributes$ cannot contain two instances of class $AValue$ with the same value of attribute $Atrribute$ for unambiguity of naming values in ontology PO. Attributes of T without the markup are called *base attributes*.

Class $Element\_T$ is constructed for each domain $T \in Domains$. This class has the marking attributes and attribute $Value^1$ with values in T. Attribute $Value$ specifies a value, and the marking attributes add the semantic markup to this value. Thus, in ontology $SMPO$, values of PO domains can be marked up.

Class $Element$ has only the marking attributes $SLabels^*$ and $SAattributes^*$. This class models new semantic classes (classes defined only by the semantic markup) in ontology $SMPO$. New semantic classes is used to construct new subject-oriented classes for ontology of processes in specific domains. This classes are used just for a readable description of a subject domain. They must be transformed to elements of ontology PO.

We illustrate addition of information about a subject domain to elements of ontology PO using the example of a sensor measuring temperature in degrees Celsius in the range from 0 to 1000. This sensor is specified by the following instance of class `Process` of $SMPO$ ontology:

```
Process(BAVs, SLabels:{"sensor"},
 SAttributes:{AValue (Attribute:"Dimension", Value:"temperature"),
  AValue(Attribute:"unit", Value:"Celsius"),
  AValue(Attribute:"range",
   Value:Element(SLabels:{"range"},
    SAttributes:{AValue(Attribute:"left", Value:"0"),
     AValue(Attribute:"right", Value:"1000")})})
```

Listing 1: Sensor instance

Here the tuple $T(A_1 : V_1, \ldots, A_n : V_n)$ denotes an instance of class T with values V_1, ..., V_n of attributes A_1, ..., A_n, the set $\{V_1, \ldots, V_n\}$ lists values of a multi-valued attribute, and $BAVs$ are base attributes from ontology PO.

Thus, with ontology $SMPO$ we can describe instances of notions from a subject domain by the semantic markup. However, this ontology is not enough to specify subject notions as elements of concurrent systems, i.e., to specify restrictions on sets of their instances. In the next section, we define a process-oriented ontology of semantic-markup patterns. This ontology is used to define notions of some subject domain using patterns by imposing restrictions on instances of classes of ontology $SMPO$ (what semantic markup can be added to them), as well as the arity and values of their attributes (the number of values of the attributes and what semantic markup can be added to these values).

4. Process-Oriented Semantic-Markup Patterns Ontology

Process-oriented semantic-markup patterns ontology ($POSMPO$) includes domains and classes of ontology $SMPO$, domains $AMatchSizes$ and $AMatchOperations$, and class $AMatch$.

Let n, m be nonnegative integers. Domain $AMatchArities = \{"m", "m|0", "mn", "mn|0", "m-", "m-|0", "-n"\}$ is used for restrictions on the number of attribute values of an ontology element matched with a pattern.

Domain $AMatchOperations = \{$"=", "<", "<=", "!=", ">", "=>", "in", "oneof", "all"$\}$ is a set of matching operations. They specify which values of ontology $SMPO$ must be matched to each other. The set of values of this domain can be extended for a specific subject domain.

Instances of $POSMPO$ classes are called *semantic-markup patterns*. Each pattern specifies a set of $SMPO$ instances matching with this pattern. Class T of ontology $POSMPO$ has the same attributes as class T of ontology $SMPO$, but they have values in $AMatch$. Class $AMatch$ specifies the rules for matching attribute values of $SMPO$ classes with patterns for them. This class has the following attributes: Ar with values in $AMatchArities$, Op with values in $AMatchOperations$ and Pat^* with values in $Values[SMPO]$ which contains all values of all attributes of all $SMPO$ classes similarly to domain $Values$ based on ontology PO. Attribute Ar restricts the number of matched values. Attribute Op defines the matching operation. Attribute Pat specifies patterns for attribute values.

Let $V.A$ denote the value of attribute A of instance V, and $|S|$ denotes the power of set S. We consider that instance V of class T from ontology $SMPO$ is matched with pattern P of class T from ontology $POSMPO$ iff for each attribute A of P such that $P.A = AMatch(Ar : R, Op : O, Pat : V_1, \dots, V_n)$ the following holds:

1. If $R = "m"$ then $|V.A| = m$.
2. If $R = "m-"$ then $|V.A| \geq m$.
3. If $R = "-m"$ then $|V.A| \leq m$.
4. If $R = "m-k"$ then $k \leq |V.A| \leq m$.
5. If $R = "m|0"$ then $|V.A| = m$ or $|V.A| = 0$.
6. If $R = "m-|0"$ then $|V.A| \geq m$ or $|V.A| = 0$.

7. If $R = "m - n|0"$ then $n \leq |V.A| \leq m$ or $|V.A| = 0$.
8. If $O = " = "$ then $n = 1$ and $V' = V_1$ for each $V' \in V.A$. The cases when $O \in \{" = ", " < ", " <= ", "! = ", " > ", " > "\}$ are defined similarly. This case restricts comparable attribute values.
9. If $O = "in"$ then $n = 1$ and $V' \in V_1$ for each $V' \in V.A$. This case defines membership of attribute values.
10. If $O = "oneof"$ then V is matched with $upd(P, A, V_i)$ for some $1 \leq i \leq n$, where $upd(Q, B, U)$ denotes the result of setting the value U to attribute B of instance Q. This case chooses some pattern value for the attributes.
11. If $O = "all"$ then there are S_1, ..., S_n such that $V.A = \{S_1, \ldots, S_n\}$, $S_i \cap S_j = \emptyset$ for $S_i \neq \emptyset$, $S_j \neq \emptyset$, and $upd(V, A, S_i)$ is matched with $upd(P, A, V_i)$ for each $1 \leq i \leq n$. This case chooses all pattern values for the attributes.
12. If O is undefined, and $A \neq SLabels$, or $T = AValue$ and $A \neq Attribute$, then V' is matched with $P.A$ for each $V' \in V.A$. This case reduces matching set of attribute values to matching separate attribute values of the set. The remaining cases are special ones for classes $SLabels$, $AValue$ и $SAttributes$.
13. If O is undefined and $A = SLabels$ then $P.SLabels \subseteq V.SLabels$.
14. If O is undefined, $T = AValue$, $A = Attribute$ then $n = 1$, and $V.A = V_1$.
15. If O is undefined and $A = SAttributes$ then $attributes(P.SAttributes) = attributes(V.SAtributes)$, where $attributes(AV)$ is the set of attributes in instance AV of class $AValue$.

We have defined a process-oriented ontology of semantic-markup patterns which combines the Process Ontology with descriptions of notions of a subject domain. A particular set of instances of this ontology gives the rules for constructing the corresponding subject-oriented process ontology. Classes and domains of $POSMPO$ provide a language for constructive using axioms which restrict abstract processes of PO with respect to a subject domain because these axioms can specify only numbers of attribute values and their ranges. In the next section, we construct some typical elements of Automatic Control Systems (ACSs) using classes $POSMPO$.

5. Domain-Specific Process Ontology for Typical Elements of Automatic Control Systems

In this section, we define semantic-markup patterns for typical elements of automatic control systems: simple and complex sensors, controllers, actuators and the controlled object.

Simple and complex sensors, and related entities are defined by patterns in Listing 2.

```
Process( // Simple sensor
Local:AMatch("0"),
SharedRead:AMatch("1", Variable(SLabels:{"Observed value"})),
SharedWrite:AMatch("0"),
Actions:AMatch("0"),
Channels:AMatch("1-",
Channel(SLabels:{"Channel from sensor to controller"})),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending observed value from simple sensor"})),
SLabels:{"Simple sensor"},
SAttributes: {AValue("Physical quantity",
Element(SLabels:{"Physical quantity"})})

Element( // Physical quantity
SLabels:{"Physical quantity"},
SAttributes: {AValue("Dimension", AMatch(Op:"in", Pat:Dimension)),
AValue("Unit", AMatch(Op:"in", Pat:Unit)),
AValue("Range", Element{SLabels:{"Range"}})})

Element( // Range
SLabels:{"Range"},
SAttributes: {AValue("Left", AMatch(Op:"in", Pat:Float)),
AValue("Right", AMatch(Op:"in", Pat:Float))})

Variable( // Observed value
Users:AMatch(Op:"all",
Pat:{AMatch("1", Process(SLabels:{"Controlled Object"})),
AMatch("1-", Process(SLabels:{"Simple sensor"}))}),
SLabels:{"Observed value"},
SAttributes: {AValue("Physical quantity",
Element(SLabels:{"Physical quantity"})})

Channel( // Channel from sensor to controller
From:AMatch("1", "oneof", {Process(Slabels:{"Simple sensor"}),
Process(Slabels:{"Complex sensor"})}),
To:AMatch("1-", Process(SLabels:{"Controller"})),
Type:AMatch("1", "=", "FIFO"),
Capacity:AMatch("1", "=", 1),
Write:AMatch("1", "=", "Old"),
Read:AMatch("1", "=", "Keep"),
Reliable:AMatch("1", "=", "true"),
SLabels:{"Channel from sensor to controller"})

ComAction( // Sending observed value from simple sensor
From:AMatch("1", Process(Slabels:{"Simple sensor"})),
To:AMatch("1-", Channel(SLabels:{"Controller"})),
Message:AMatch("1",
Expression(Op1:AMatch("1",
Variable(SLabels:{"Observed value"})))),
SLabels:{"Sending observed value from simple sensor"})

Process( // Complex sensor
SharedRead:AMatch("1-", Variable(SLabels:{"Observed value"})),
SharedWrite:AMatch("0"),
Channels:AMatch("1-",
Variable(SLabels:{"Channel from sensor to controller"})),
ComActs:AMatch("1-",
ComAction(SLabels:{"Sending message from complex sensor"})),
SLabels:{"Complex sensor"},
SAttributes: {AValue("Physical quantity",
Element(SLabels:{"Physical quantity"})})

ComAction( // Sending message from complex sensor
From:AMatch("1", Process(Slabels:{"Complex sensor"})),
To:AMatch("1-", Channel(SLabels:{"Controller"})),
SLabels:{"Sending message from complex sensor"})
```

Listing 2: Sensors

In this and the following listings, we use the following abbreviations: `SLabels:S` for `SLabels:AMatch(`

`Value:S)`, `AMatch(R, O, P)` for `AMatch(Ar:R, op:O, Pat:P)`, where `R`, `O`, or `P` can be omitted, `AMatch(R, O, P)` for `{AMatch(R, O, {P})}`, `AValue(A, V)` for `AValue(Attribute:A, Value:V)`, and `T` for `AMatch(Op:"in", Pat:T)`.

These patterns impose the following restrictions on sensors. Sensors must read observed values from variables shared with the controlled object and cannot change it. They must have outgoing channels connecting them with controllers and communication actions for sending messages to the controllers. There must be at least one controller and at least one shared variable associated with each sensor. Simple sensors have no local variables and actions whereas complex sensors have ones. Simple sensors can observe exactly one shared variable and send the observed value unchanged to controllers. For sensors, processing physical quantities must be defined. They are characterized by dimensions (`"temperature"`, `"pressure"`, `"density"`, etc.), units of measurement (`"centimeter"`, `"kilogram"`, `"volt"`, etc.) and ranges.

Controllers, *actuators* and *controlled objects* are restricted by patterns in Listing 3.

```
Process( // Controller
SharedRead:AMatch("0"), SharedWrite:AMatch("0"),
Channels:AMatch("all",
{AMatch("1-", Channel(SLabels:{"Channel from sensor to controller"}),
AMatch("0-", Channel(SLabels:{"Channel from actuator to controller"},
AMatch("1-", Channel(SLabels:{"Channel from controller to actuator"},
AMatch("0-", Channel(SLabels:{"Channel from controller to controller"})}),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending message from controller"})),
SLabels:{"Controller"})

Process( // Actuator
SharedRead:AMatch("0"), SharedWrite:AMatch("0"),
Channels:AMatch("all",
{AMatch("1-", Channel(SLabels:{"Channel from controller to actuator"}),
AMatch("0-", Channel(SLabels:{"Channel from actuator to controller"},
AMatch("1", Channel(SLabels:{"Channel from actuator to controlled object"})}),
ComActs:AMatch("1-", ComAction(SLabels:{"Sending message from actuator"})),
SLabels:{"Actuator"})

Process( // Controlled object
SharedRead:AMatch("0"),
SharedWrite:AMatch("1", Variable(SLabels:{"Observed value"})),
Channels:AMatch("1-", Channel(SLabels:{"Channel from actuator to controlled object"}),
ComActs:AMatch("0"),
SLabels:{"Controlled object"})
```

Listing 3: Controllers, actuators and controlled objects

Controllers and actuators must not have shared variables. Controllers must have output channels connecting them with other controllers and actuators, and input channels connecting them with sensors and actuators. Actuators must have output channels connecting them with controllers and the controlled object, and input channels connecting them with controllers. There must be at least one sensor and at least one actuator connected with a controller through input and output channels, respectively. There must be at least one controller and the only controlled object connected with an actuator through input and output channels, respectively. The controlled object must be connected with actuators by input channels. There must be at least one shared variable, one sensor and one actuator associated with the controlled object.

Each pattern gives rules for defining an element of ACS in the Process Ontology. With a set of such patterns, we can specify a system of concurrent processes implementing typical elements of ACS. Thus, our method can be used to specify domain-specific processes.

6. Discussion and Conclusion

The method of developing a domain-specific process ontologies based on three core ontologies [6] has several remarkable properties. Verification-oriented process ontology PO specifies a compact universal process model with a labeled transition system as its formal semantics, which can be used in formal verification methods and model checking, in particular. Semantically-marked process ontology $SMPO$ makes possible marking instances of PO classes for associating them with concepts of a subject domain. Moreover, it is also possible to mark values of PO domains and describe new domain-specific classes. Process-oriented semantic-markup patterns ontology $POSMPO$ specifies restrictions on the semantic markup of instances of $SMPO$ classes, defining the subject concepts associated with these instances. Unlike the declarative approach describing a domain-specific process ontology by a set of axioms, this approach specifies the ontology as a set of patterns (instances of ontology $POSMPO$) for defining domain-specific processes constructively as instantiation of patterns from this set. All three ontologies are based on simple concepts that can be used as ontology design patterns [7, 8].

In the future, we plan to add new kinds of matching operations (for example, the current set of operations does not allow us to express the property that different attributes have the same instance as a value), to refine the process ontology for automatic control systems and to advance the method for building other domain-specific process ontologies.

References

1. Garanina, N., Zyubin, V., Lyakh, T., Gorlatch, S. An Ontology of Specification Patterns for Verification of Concurrent Systems // Proc. of the 17th Intern. Conf. SoMeT-18. New Trends in Intelligent Software Methodologies, Tools and Techniques. Series: Frontiers in Artificial Intelligence and Applications, Amsterdam: IOS Press, 2018. Vol. 303. P. 515-528. DOI 10.3233/978-1-61499-900-3-515.
2. Garanina N.O., Anureev I.S. Verification oriented process ontology // Proc. of 9th Workshop "Program semantics, specification and verification: theory and applications" (PSSV 2018). 2018. P. 58–67.

3. Garanina N.O., Sidorova E.A., Bodin E.V. A Multi-agent Text Analysis Based on Ontology of Subject Domain // Lecture Notes in Computer Science, 2015. Vol. 8974. P. 102–110.
4. Garanina N.O., Sidorova E.A. Context-dependent Lexical and Syntactic Disambiguation in Ontology Population // Proc. of the 25th Intern. Workshop on Concurrency, Specification and Programming (CS&P). Humboldt-Universitat zu Berlin, 2016. P. 101–112.
5. Garanina N.O., Sidorova E.A., Kononenko I.S., Gorlatch S. Using Multiple Semantic Measures For Coreference Resolution In Ontology Population // Intern. Journal of Computing. 2017. Vol. 16. No. 3. P. 166–176.
6. Scherp A., Saathoff C., Franz T., Staab S. Designing core ontologies // Applied Ontology, 2011. Vol. 6. No. 3. P. 177–221.
7. Gangemi A., Presutti V. Ontology Design Patterns // Staab, S., Studer, R. (eds.) Handbook on Ontologies. 2nd edn. Springer, 2009. P. 221–243.
8. Ontology design patterns. `http://www.ontologydesignpatterns.org`.