Processes Construction and π-calculus-based Execution and Tracing

Leonid Shumsky, Vladimir Roslovtsev and Viacheslav Wolfengagen
Department of Cybernetics, Moscow Engineering Physics Institute "MEPhI", Kashirskoe sh., Moscow, Russia

Keywords: Business Process Construction, λ-Calculus, π-Calculus, Business Process Execution Semantics, Business Process Debugging.

Abstract: Many of the state-of-the-art business-process modelling and managing techniques rely on methods that lack sound theoretical basement, though the latter being of advantage, as is acknowledged by more and more people, in practical information system design and implementation. The software (and, in fact, the very processes the software is supposed to automate) tend to become 'properly designed', thus ensuring higher degrees of software (and processes) extensibility, adaptability, better verification and execution control. In this paper we discuss a constructive approach to process design and we present process execution semantics based on π-calculus and process analysis and debugging technique based on formalized execution logs.

1 INTRODUCTION

Many of the state-of-the-art business-process modelling and managing techniques rely on methods that lack sound theoretical basement, though the latter being of advantage, as is acknowledged by more and more people, in practical information system design and implementation. The software (and, in fact, the very processes the software is supposed to automate) tend to become 'properly designed', thus ensuring higher degrees of software (and processes) extensibility, adaptability, better verification and execution control. The formalisms in use are mostly variations of network, state diagram (Petri Coloured Nets), document-oriented or event-oriented models, or sometimes are a mix of those (IDEF family, UML). Many models in use aren't exactly formal, or at least aren't used formally, acting more like an instrument for visualization. However, symbolic models appear to be more efficient when it comes to automated processing.

We suggest using for this purpose π-calculus and λ-calculus in conjunction: λ-calculus for high-level (domain-oriented) processes internal structure representation and π-calculus to capture (sub)processes interaction and execution semantics. λ-calculus focuses on variable binding and substitution, while π-calculus deals with names whose sound meaning depends partly on its occurrence context and partly on the chosen evaluation semantics, so that a name may refer to a data object, data transfer channel, atomic process, variable, etc. This uniformity in π-calculus is what makes it particularly suitable to serve as a standardized, extensible framework suitable to use across multiple systems, each using its own specific extension of the standard version. π-calculus features dynamic process construction, passing and executing sub-process, higher-order functions valuation.

We show how the main notions of the process theory are modelled in the π-calculus, and how to benefit from using π-calculus in solving the main tasks. We show how process execution 'formalized logs' may be used to debug and verify processes (something also known as 'process mining').

The rest of the paper is organized as follows. Section 2 presents an abstract machine based on π-calculus to execute processes. Section 3 discusses an approach to build process execution logs as formal entities that may be used for process correctness accession. Section 4 presents a higher-level, methodological, insight process construction

2 PI-CALCULUS AS PROCESS EXECUTION SEMANTICS

This section presents an abstract machine based on π-calculus to execute processes, see (Milner, Parrow

Shumsky L., Roslovtsev V. and Wolfengagen V..
Processes Construction and π-calculus-based Execution and Tracing.
DOI: 10.5220/0004972304480453
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 448-453
ISBN: 978-989-758-028-4

and Walker, 1992).

2.1 Basic Definitions

The alphabet of π-calculus contains the following components. The first one is the set of "names", denoted by $\mathcal{N}$, consisting of small letters. The second one is the set of processes, denoted by capital letters. Terms of π-calculus are constructed inductively by adding prefixes to existing processes or by 'joining' existing (complex) processes. The available construction operations are represented by the following grammar:

$$P ::= \mathbf{0} \mid \bar{x}y.P \mid x(y).P \mid (x)P \mid (P|Q) \mid !P$$

In this grammar the sign **0** stands for the empty process. The prefix $\bar{x}[y_1, \dots, y_n]$ describes transferring the names $y_1, \dots, y_n$ over the link (channel) x; the prefix $x(y_1, \dots, y_n)$ describes receiving data items and binding the names $y_1, \dots, y_n$ to these data items, respectively. Communication between processes is an act of sending some data via a certain name (channel) in one process $(\bar{x}z)$ and receiving them through the same name (channel) in another one $(x(y).P)$, the latter, informally speaking, sort of 'listening' for any 'messages'. In $x(y).P$ the identifier y is bound, so that when a process receives a message, a data item z, this item is being substituted instead of every unbound occurrence of y in P: $P[z/y]$. Another way of bounding name in π-calculus is creating a local name $(x)P$. This statement bounds name x in the process P – such a name becomes locally defined in P and no other inner process cannot interact with P using that name. Meaning of this instruction is to create inner, protected or temporal channels. The grammar of calculus determines two operations for creating new processes from existing ones – parallel execution $(P|Q)$ and replication $!P$. The set $\mathcal{N}$ is the most biggest set of identifiers in use. If it is required for any special task, we can select subsets of $\mathcal{N}$. For example, one might describe a (computational) process via a term of λ-calculus and 'embed' this term directly into the π-calculus process. That would require selecting a subset $\mathcal{V}$ of variables in $\mathcal{N}$.

Note that such an embedding takes some additional efforts so as to bridge the gap between the two worlds – that of the λ-calculus and that of π-calculus. In the next section we will address this issue. The rest of this section describes some prerequisites.

By definition, α conversion is renaming of each occurrence of bound variable in term. In general, if $o[x_1, \dots x_n]$ is a prefix which bounds identifiers $x_1 \dots x_n$ in some term P, then substitution $o[y_1/x_1, \dots, y_n/x_n].[y_1/x_1, \dots, y_n/x_n]P$ is α conversion. If term Q could be derived from term P by finite amount of α conversions then P and Q are α equivalent. π-calculus provides two ways to bound names – prefix of data receive and creating local name – but using extension could increase this amount.

We will use Chemical Abstract Machine (ChAM), see (Berry and Boudol, 1992), to describe execution of processes. Execution of process with ChAM suggests building from source process "molecular solution" (later we will write just "solution") and application to this solution suitable rules of reaction. Reactions could be either reversible, denoted with ↔ or irreversible, denoted by →. Chemical Abstract Machine is much more flexible and easier to extend than simple operation semantic, so will prefer it for our model. The minimum set of rules required to cover basic functionality of π-calculus grammar is:

$\{\|P \mid Q\|\} \leftrightarrow \{\|P, Q\|\}$	Parallel execution
$\{\|!P\|\} \leftrightarrow \{\|P, !P\|\}$	Replication
$\{\|(x)P, Q_1, \dots, Q_n\|\} \rightarrow (x)\{\|P, Q_1, \dots, Q_n\|\}, x \notin FN(Q_1, \dots, Q_n)$	Local variable creation
$\{\|\mathbf{0}, P\|\} \leftrightarrow \{\|P\|\}$	Empty process
$\{\|\bar{x}y.P\|\} \rightarrow \{\|\bar{x}y, P\|\}$	Asynchrony
$\{\|\bar{x}y, x(z).Q, R\|\} \rightarrow \{\|\|^{y}/_{z}\|\|Q, R\}$	Communication

2.2 Capturing Semantics Extension

We introduce the following definitions for the mechanism of terms interpretation in a subject domain. First, we will select a subset of available, pairwise distinct, π-calculus names $\mathcal{L} \in \mathcal{N}$. The set $\mathcal{L} \cup \tau$ we will call labels. A label is either a name of π-calculus or the special identifier τ, which means an unspecified label ('label is not set'). The difference between labels and ordinary names of the π-calculus is their relation with names binding and α conversion. The main feature of labels is that a label could not be bound and hence could not be subject for α-conversion. Adding labels changes the grammar of term construction in the following way:

$$P ::= \bar{x}[y_1, \dots, y_n].P, \{x, y_1 \dots, y_n\} \subseteq \mathcal{N}$$
$$P ::= x(y_1, \dots, y_n).P, x \in \mathcal{N}, \{y_1, \dots, y_n\} \subseteq \mathcal{N} \backslash \mathcal{L}$$
$$P ::= (x)P, x \in \mathcal{N} \backslash \mathcal{L} \qquad P ::= P|Q \qquad P ::= !P$$

We will denote this grammar by Γ_l. If a process term P is valid in this grammar, that fact we will denote with $\Gamma_l \vdash P$. This grammar shows that labels can stand for transmitted data or channel namesused for communication.

As we have already mentioned, a regular π-calculus term describes the structure of a process execution, rather than a subject area process itself. The difference lies in understanding the reduction of corresponding terms. Reduction of a regular π-calculus term (process execution structure term) has no additional meaning – it shows only the order of execution. On the other hand, reduction of a subject area process describes the use and interaction of subject area's entities. Every reduction step captures additional meaning of changing of these objects.

We will define the notion of execution context Δ for the term $P \mid \Gamma_l \vdash P$ as a first step to building subject area process model. This context is a map with keys $l \in \mathcal{L}$, and values are conceptual model objects connected with this label and represented, using the approach from (Shumsky et al., 2013). Structure model of a process M_S is a term of π-calculus, which has the empty execution context or it does not have a connected notion for every label, used in the model. Conceptual process model M_C is a term, which has at least one occurrence for each label in execution context.

$P, \Delta \in M_C \Rightarrow \Gamma_l \vdash P \;\&\; \forall n \in FN(P) \cap \mathcal{L} \rightarrow\; n \in \Delta$
Conceptual process model

$P, \Delta \in M_S \Rightarrow \Gamma_l \vdash P \;\&\; \exists n \in FN(P) \cap \mathcal{L} \rightarrow\; n \notin \Delta$
Structure process model

The difference between these two types of models is that non-redex terms in the structural model may be reduced in the conceptual model due to the use of the functions associated with the labels. The functions are taken from process execution context, and additional reduction rules for the abstract machine are required:

$$\{|(l_1(y).P, R)(l_1, c_1; y, c_2; \Delta)|\} \rightarrow \{|([l_2/y]P, R)(l_2, \downarrow_{\mathcal{B}} c_1 c_2; \Delta)|\}$$
(*recieving data*)

$$\{|(\overline{l_1} l_2, R)(l_1, c_1; l_2, c_2; \Delta)|\} \rightarrow \{|(\downarrow_{\mathcal{B}} c_1 c_2, R)(\Delta)|\}$$
(*sending data*)

These rules describe communication of processes with external systems by executing functions, defined for channel-labels. Data to send are described with label and should by taken from process or produced by executing inner function, data to receive are described by template name, which is filled by channel. There are several approaches to execute terms of λ-calculus within terms of π-calculus, see (Boudol, 1998) and (Milner, 1992), so we do not go in further details here.

If term $P \in M_C$ and $Q \in M_S$ and $P =_\alpha Q$, then we will state that process *P* implements the structure of the process *Q*, or just *P* is an implementation of *Q*.

An interpretation of a π-calculus process is a function, which maps structure model of the process to its specific implementation in some chosen subject area, defined by interpretation domain $\Delta^{\mathcal{J}}$. The similar approach is used in interpretation of conceptual models, for example for description models interpretation (Baader et al., 2003).

$$\cdot^{\mathcal{J}}: M_S \times \Delta^{\mathcal{J}} \rightarrow M_C; \Gamma_l \vdash P \in M_S \rightarrow \Gamma_l \vdash P^{\mathcal{J}} \in M_C$$

Interpretation domain is a mapping with structure similar to execution context of a process, which contains all labels and conceptual model entities specific to the subject area. The interpretation function processes an input term as follows: for each label in the term it either inserts in processes' context a value from the interpretation domain, if it is possible, or replaces the label with some regular name:

$(\mathbf{0}, \Delta)^{\mathcal{J}} = \mathbf{0}, \Delta$
$(P \mid Q, \Delta)^{\mathcal{J}} = P^{\mathcal{J}} \mid Q^{\mathcal{J}}, \Delta$
$(!\,P, \Delta)^{\mathcal{J}} = !\,P^{\mathcal{J}}, \Delta$
$(\bar{x}[y_1, \dots, y_n].P,\ \Delta)^{\mathcal{J}} = \overline{[x, \alpha]_{\mathcal{J}}}[[y_1, \alpha]_{\mathcal{J}}, \dots, [y_n, \alpha]_{\mathcal{J}}].P^{\mathcal{J}}, \Delta \cup \Delta^{\mathcal{J}}(x, y_1, \dots, y_n)$
$(x(y_1, \dots, y_n).P)^{\mathcal{J}} = [x, \alpha]_{\mathcal{J}}(y_1, \dots, y_n).P^{\mathcal{J}}, \Delta \cup \Delta^{\mathcal{J}}(x)$

Operator $[\cdot, \alpha]_{\mathcal{J}}$ returns its first argument if it exists in interpretation domain and α-conversion of that argument otherwise.

The ideas of this section are used to describe processes tracing.

3 TRACING PROCESSES EXECUTION USING 'SEMANTIC' LOGS

The main purpose of every modelling tool is to provide means for adequate reflection those featires of real objects in subject areas that are relevant in a given context, and that includes making certain hypotheses about these objects and their formal verification. In process modelling a common way for conformance checking of models are functions of fitness, simplicity, precision, and generalization (van der Aalst, 2013). This section focuses on applying a precision function to process models of π-calculus.

A starting point for any measurements of models conformance checking is the notion of a process log. In general, a log is a set of process traces – results of a single execution of that process represented as a sequence of actions and action results. Models conformance analysis is based on conformance checking between process models and existing logs – specifically, on checking whether or not a given log could be obtained by execution of that model/

Other factors are: how many new traces could be obtained from that model, which traces are redundant, etc.

Execution log for a process conceptual model is represented as a list of traces. Our approach involves representation of the trace as a function with predefined structure, which relates specified characteristics of processes execution with ChAM. In this paper we will use following characteristics – type executed action (tracked reactions of chemical machine), labels, involved in reaction (quantity of labels is defined by reaction type and processes structure) and conceptual model's objects, which connected with selected labels. We will use special extension of our mechanism of processes execution to define and select these trace's characteristics. This extension is needed to observe applied reactions of ChAM.

The main idea of this extension is to execute processes not directly with ChAM, but with the help of some external tool, which allows retracing process execution progress. This extension will be realized with operator *N* with following rules of application to ChAM solution:

$$N(\{|(l_1(y).P,R)(l_1,c_1;y,c_2;\Delta)|\}) =$$
$$[eo,(l,y),(c_1,c_2)] :: N(\{|([l_2/y]P,R)(l_2,\downarrow_\beta c_1c_2;\Delta)|\})$$
$$N(\{|(\bar{l}_1l_2,R)(l_1,c_1;l_2,c_2;\Delta)|\}) =$$
$$[ei,(l_1,l_2),(c_1,c_2)] :: N(\{|(\downarrow_\beta c_1c_2,R)(\Delta)|\})$$
$$N(\{|(\bar{x}l_1,x(z).Q,R)(l_1,c_1;\Delta)|\}) =$$
$$[c,(\tau,l_1),c_1] :: N(\{|[^y/_z]|Q,R\})$$
$$N(S) = \textbf{\textit{if}}\ S \downarrow_{ChAM} = S\ \textbf{\textit{then}}\ S\ \textbf{\textit{else}}\ N(S \downarrow_{ChAM})$$

The operator *N* of process observation can be implemented in any variation of process modelling system, since it does not depend on specific aspects of modelling. This operator maps a molecular solution of the ChAM to a list of tracked reactions, the last element being the process execution result (which is a solution with no reducible reaction). Note that π-calculus names that are not labels do not occur in the log, the empty label τ being in their stead. This is because a name that is not a label is not an identifier, either, and has no interpretation after the process being executed, so that such a name will be redundant at best.

ChAM reduction rules are not assigned priority rating, so that they may be executed at an arbitrary order. The operator *N* helps in solving the problem, assigning priorities to operations, so that every time there more than one candidate rule, the one with highest priority is preferred. That does not change the ChAM itself, but the log depends on the process model only.

Process execution log item is represented with a term that follows these two rules: first, it must contain information on process execution progress, and second, it must contain a routine, a function, to check its correctness. Presently, we stick to the simplest solution (which also helps to make things more clear):

$$T = \lambda v.(\lambda xyz.[v.1 = x]\ \&[v.2 = y]\&[v.3 = z])[t,l,c]$$

Here, the constants *t*,*l*,*c* correspond to specific values relating to the log item, and the internal expression performs a simple parameter correspondence. Note that this check may be of arbitrary complexity, given that it passes the type checking. For instance, we may take into account the label's and canonical model objects' weights (van der Aalst et al., 2011), or other data. Such encapsulation of the checking method into the log item results into additional flexibility allowing various checking algorithms for each process model, and even for each subprocesses of the same process.

A simple correspondence checking function (routine) *Ch* that will be described later takes a process execution log and a model against which the log is to be checked. The result is either a discrete (e.g. binary) or continuous value that shows conformance degree. All the checking logic encapsulated in the log entry, the checking function's task is simply to initiate parallel execution of the process using the execution control operator and to run each execution step against a current log item object. After that, the checking function must do the right aggregate operation and construct a well-formed resulting object.

We will build such a function on a step-by-step basis. First, we construct a function of parallel execution of the process log and model, the former being the more difficult that not every reduction step has a corresponding log item (not all reduction steps are logged). We overcome this difficulty by defining an operator ℕ that applies the defined above operator N to the process until the next logged reduction is at last executed:

$$\mathbb{N}S = \textbf{\textit{if}}\ length(NS) = 1\ \textbf{\textit{then}}\ \mathbb{N}S\ \textbf{\textit{else}}\ NS$$

Now, our function that starts parallel process and log execution will look like this:

$$ChS[L,P] =$$
$$\textbf{\textit{if}}\ head(L)(head\ \mathbb{N}P)$$
$$\textbf{\textit{then}}\ ChS(tail\ L, tail\ \mathbb{N}P)$$
$$\textbf{\textit{else}}\ ChS\left(L, P \cup \left(B(head\ \mathbb{N}P)\right)\right)$$

The idea is, whenever a log item cannot be obtained from the model, we add a new element that strictly corresponds to the missing item and resume the checking routine.

Parallel execution function *ChS* is used as a basis for building the simple checking function *Ch*, and there are three major strategies (some of their combinations are also valid) to build it:

1. The checking routine should abort on the first encounter of an invalid log item that cannot be obtained from the model;
2. The routine could count all the erroneous items in the log;
3. We could count all items additional items that remained unchecked after the routine finished.

The checking process may be augmented to enable correct model evaluation and increase correctness, using a typed system (Pierce and Sangiorgi, 1993).

4 COUPLING WITH THE APPLICATIVE APPROACH

Applicative Computing is a way of organizing and performing computing based on compositional construction of computational blocks out of simpler, previously build blocks, each block being closed and with no free variables (Wolfengagen, 2010). The formal means of constructing those blocks are studied in applicative computational systems (ACSs) which focus on developing the notion of object as a functional entity that may be applied to an argument object or passed as an argument to another object. As it turns out, ACSs are particularly suitable for building domain-specific frameworks for compositional processes design, especially their typed versions, and especially in conjunction with semantic models (e.g. description logic or a frame theory) when concepts are embedded in the computational model as types. In (Wolfengagen, 1984) embedding a frame theory into λ-calculus is shown, and (Shapkin, 2010) shows embedding description logic into the typed λ-calculus.

In this section we give a number of methodological considerations on lambda-calculus and combinatory logic from the perspective of process construction.

4.1 The Combinatory Logic Approach

An important notion in ACSs is that of a combinator. This notion comes from combinatory logic and means an object build out of a predefined set of 'constants' – initial objects, in the sense of (Wolfengagen, 2010). The only main object constructor is the application operation, though in applied theories others may be introduced, either as syntactical elements denoting some (initial) combinators, or as meta-level entity, as suggested in (Roslovtsev and Luchin, 2009) and (Roslovtsev et al., 2013).

Basically, all combinators are built out of *constant atomic objects* having no references to the environment of any kind, thus eliminating side-effects. This is a very helpful feature that reduces the number of state synchronization points, concurrency issues, etc., and facilitates and improves overall design.

The combinatory logic philosophy is that of describing things in terms of a fixed set of basic (*initial*) entities, some of them considered *atomic*. The basis is extensible, to some degree, but the only way for extension is to turn some of the complex objects to the initial category.

4.2 The Lambda-calculus Approach

In λ-calculus objects are build, basically, out of variables using the application and functional abstraction (meta-)operators. Constants, if they are Turing-computable objects, may be synthesised, so that philosophically λ-calculus suggest an adaptive approach when the computational basis is actually synthesised, perhaps dynamically, through detection of the most commonly used sub-objects (expressions) and their injection in the system as initial objects.

The notion of a combinator may be naturally extended to the case of λ-calculus: a combinator is an object with no free variables, so in a combinator all occurring variables are bound with the functional abstraction operator. λ-calculus may be seen a reference theory, of sorts, in that an occurrence of a bound variable in a term is actually a reference to a specific item or a specific spot in the outside environment. This formalises and facilitates objects dependencies control, and also helps in localizing and controlling side effects when they are indeed intended and necessary (though at a cost of some technical complexity).

Again, the system may be extended in two ways: the conventional one consists in adding new combinators to the set of initial objects, complemented, perhaps, with some 'syntactic sugar', as is shown in (Barendregt, 2012, ch. 6); the second way (already mentioned above), simplifies capturing the domain-specific semantics.

4.3 Two-level Process Modeling

The best part of the paper was devoted to explaining how π-calulus may be used for not only process modelling and execution, but also for process tracing and verification. Though π-calculus itself may be used for process modelling and even developing process algebras, its primitives are a little too 'low

-level' from the perspective of subject area sematics.

We suggest that higher-level process modelling, more aware of subject domain semantics, will profit from using ACSs. First, ACSs are computational systems that rely on constructive definition of processes (and data objects, too) and explore their equivalent transformations (including optimizations, in some contexts). Second, very powerful type systems for ACSs are known, and, in fact, developed as their part; besides, connection with variations of logics are more or less well explored, which helps in direct usage of subject domain concepts and relationships in processes (and data objects) definition.

Though there are numerous abstract machines for ACSs, distributed and parallel execution remains yet relatively poorly explored; π-calculus and, in particular, our approach would fill the gap to some extent. The idea, briefly speaking, is to transform the semantic-aware applicative model to a more 'lightweight' system based on π-calculus to perform actual execution.

5 CONCLUSION

In this paper we presented an approach to business process execution tracing. Usually, executing a process on an abstract machine (AM) comprises a series of steps, each consisting in executing a simple, relatively low-level, instruction that changes AM's current state. However, given an arbitrary step (and a number of preceding steps) it is difficult to evaluate whether process execution goes as expected, or which phase of the process is being actually executed. What we suggested may be seen as an extension an AM for executing π-calculus processes that facilitates this kind os 'semantic' process debugging.

REFERENCES

van der Aalst, W. M., van Dongen, B. F., and Adriansyah, A., 2011. Conformance checking using cost-based fitness analysis. In *Proc. of the 15th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 55–64.

van der Aalst, W. M., 2013. A general divide and conquer approach for process mining. In *Proc. of the Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 1–10.

Barendregt, H., 2012. *The Lambda Calculus, its Syntax and Semantics*. Studies in Logic, Mathematical Logic and Foundations, vol. 40, College Publications, 2012.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., 2003. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press

Berry, G. and Boudol G., 1992. The Chemical Abstract Machine. *Theoretical computer science, vol. 96, no. 1*, pp. 217–248, 1992.

Boudol, G., 1998. The π-calculus in direct style. *Higher-Order and Symbolic Computation, vol. 11, no. 2*, pp. 177–208.

Milner, R., 1992. Functions as processes. *Mathematical structures in computer science, vol. 2, no. 02*, pp. 119–141.

Milner, R., Parrow, J., and Walker, D., 1992. A Calculus of Mobile Processes. *Information and Computation*, vol. 100, no. 1, pp. 1–40.

Pierce, B. and Sangiorgi, D., 1993. Typing and subtyping for mobile processes. *In Proceedings of Eighth Annual IEEE Symposium on Logic in Computer Science, LICS'93*, pp. 376–385.

Roslovtsev, V.V., Luchin, A.E., 2009. Concept of Higher-Order Applicative Computational Environment. In *Proceedings of the 11th international workshop on computer science and information technologies CSIT'2009*, pp. 48-53.

Roslovtsev, V., Wolfengagen, V., Shumsky, L., Bohulenkov, A. and Sakhatskiy, A., 2013. Applicative Approach to Information Processes Modeling. Towards a Constructive Information Theory. In *Proceedings of the 15th International Conference on Enterprise Information Systems (ICEIS 2013)*, Vol. 2, Angers, France, July 4-7, 2013. – pp. 221-226

Shapkin, P., 2010.Computing with Concepts in am Applicative Programming Language. In Proc. of the 2nd International Conference on Applicative Computational Systems (ACS'2010), Moscow, Russia, 2010. – p. 205-213. [In Russian]

Shumsky, L., Roslovtsev, V., Belyaev, E., Bordonos, A., Kazantsev, N. A *Synthetic Approach to Building Canonical Model of Subject Area in Integration Bus*. IEEE ISKO Maghreb Proceedings 2013

Wolfengagen, W.E., 1984. Frame Theory and Computations. In *Computers and Artificial Intelligence*, Vol. 3, No. 1, 1984. pp. 3-32.

Wolfengagen V.E., 2010. *Applicative computing. Its quarks, atoms and molecules*. Ed. Dr. L.Yu. Ismailova. Moscow, Russia: Center JurInfoR, Ltd.