Adding agility to Enterprise Process and Data Engineering

Sergey ZYKOV
National Research Nuclear University MEPhI, 31 Kashirskoe shosse,
Moscow 115409 Russia

Pavel SHAPKIN
National Research University Higher School of Economics, 20 Myasnitskaya St.
Moscow 110000 Russia

Nikolay KAZANTSEV
National Research University Higher School of Economics, 20 Myasnitskaya St.
Moscow 110000 Russia

and

Vladimir ROSLOVTSEV
National Research Nuclear University MEPhI, 31 Kashirskoe shosse
Moscow 115409 Russia

ABSTRACT

Managing development of large and complex enterprise architectures is a key problem in enterprise engineering. Nowadays one of the breathtaking topics considering enterprise context is real-time system agility. The paper discusses an appropriate general architecture pattern and provides insights how dynamic process management environment could be made. We survey general enterprise software architecture and current agility problems. We introduce a special component called a process knowledge base and justify its crucial role in achieving agility within the enterprise. We study both the architecture of the process knowledge base as well as formal basis for its implementation which relies upon the type theory.

Keywords: Enterprise Architecture, Data Management, Process Management.

1. INTRODUCTION

Usually, a system designed with flexibility as one of its key properties in mind could be decomposed on smaller sub-systems that could be then arranged from "higher to lower" level. Complex and mission-critical enterprise architecture usually focus on: business processes, software systems and data.

Appropriate structuring, however, is not enough in the current economic era of high turbulence, where fluctuating customer demand shapes the enterprise value-added chains every day and leaves non-accommodated enterprises with higher resource consumption, bigger costs and smaller profits. Under such conditions architectural design generally reaches beyond the software engineering scope, and could be related to system engineering. Agile methodologies widely used in 2000s in system engineering could assist in overcoming the modern era challenges in enterprise engineering.

In this paper, we discuss how such agility may be achieved through usage of an integration environment augmented with a knowledge base about formal specifications of processes and artifacts. We outline here a formal basis necessary for such a knowledge base.

The rest of the article is organized as follows. Section 2 gives a survey of the general enterprise software architecture and studies the aspects of enterprise agility. Section 3 outlines the concept of the process knowledge base. Section 4 gives a formal basis for the knowledge base implementation build upon the type theory. In conclusions we summarize the proposed ideas.

2. GENERAL ARCHITECTURE: ENTERPRISE SOFTWARE ENGINEERING MATRIX

Let us begin with a general enterprise software architecture that we will use throughout in this paper. As main architecture pattern we propose to use the Enterprise Engineering Matrix that consists of processes, data and systems [1]. It is presented in fig. 1.

The first perspective shows the dynamics of architecture, namely the decomposition of strategic goals on business processes, actions and tasks. The second perspective reflects the decomposed data objects used in processes while the third one — enterprise systems that operate those data.

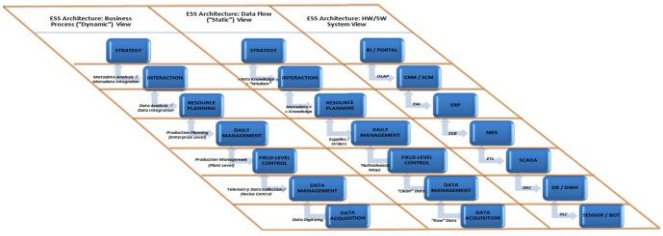

Fig. 1. The Enterprise Engineering Matrix

Agility via the Dynamic Process Management Environment
The agility of Enterprise Architecture is becoming a concept that incorporates the ideas of "flexibility, balance, adaptability, and coordination under one umbrella" [2]. This concept

includes coordination costs needed for communication between process participants and enhancing services orchestration and choreography. This effort requires a number of collaboration initiatives with the customer and expert communities that should be supported either by technical tools or expert communities to find a way to build up the "collaboration of business processes from both sides" [3]. Observing corporate performance from the longitude perspective one can see the intangible components shaping corporate strategies, such as enterprise learning and corporate knowledge. While since 2000s knowledge management could use only static database structures for storing the knowledge assets (wiki, blogs, content management systems — CMS, etc.), nowadays modern semantic technologies could drive the performance of such content (knowledge) storing and integrating it on the level of processes and making it highly accessible for every employee.
We now outline in general the high-level pattern [4-7], which describes enterprise hardware-and-software system conceptual scheme (Fig. 1 shows the diagram). The general idea is that, abstraction level, data aggregation degree and abilities for strategic analysis, justification and decision-making are growing bottom-up across the layer hierarchy. Each layer communicates directly with those adjacent to it, being consumer for lower levels and provider for higher levels.

The top ("strategic") layer of the enterprise software-and-hardware system is represented by the software toolkit for integrate representation of strategic data slices for enterprise management. It is the dashboard, which allows enterprise top management to track dynamics of key performance indicators. The data is aggregated from software systems for planning enterprise resources, both general-purpose, such as human resources, financials, time and the like, and domain-specific. In oil-and-gas enterprise examples of domain-specific reports would be gas balances, deposit assets, oil-and-gas upstream and downstream, seismic exploration data etc. An enterprise internet portal or a similar tool is used to integrate and visualize high-level reports, and to provide flexible, reliable and secure online access for the management by means of dashboard-like interface.

The next ("relationships") layer's purpose is to inform the end-user employees of urgent updates of standard business processes, such as document approval, communication with clients and suppliers, and target email messaging. In essence, the current layer is represented by the software system for interaction of the key organizational units and responsible enterprise employees with the respective clients and partners. Thus, the "relationships" layer is functionally similar to customer relations management (CRM) system. In oil-and-gas industry we have to work with oil-and-gas shipment contracts, oil-and-gas distributors, gas pipeline producers, etc.

The next, "resource" layer represents the Enterprise Resource Planning (ERP) software systems. Again, this layer consolidates the lower ("accounting") layer data to get a more strategic representation of key business indicators. This level includes ERP software modules and subsystems, which assist in management and planning of oil-and-gas products including: fixed assets, human and financial resources, documentation management, facility supplies for oil-and-gas deposit construction, and oil/gas processing, supplies for NPP unit construction, electricity processing. Possible instances of such systems for the NPP industry include Siemens and Catie software products [9].

The "accounting" layer, contains lower-level software systems as compared to the previous ones. While the previous layer is more of analytical kind and focuses on forecasting dynamics of key production indexes (such as revenues, profits, overheads, personnel defections, expenditures etc.), the current layer's tasks are more operational in nature. This includes software systems for accounting, warehousing, inventory management and the like. In NPP, that includes reaction unit construction, shipment, and assembly, monitoring NPP unit assembly maps, and technical conditions

The next architectural layer, called the "supervisory" layer, contains the software systems, which incorporates the "drivers", i.e. the interfaces between software and hardware components. This layer contains the SCADA systems for the end-users who interact with field-based devices and sensors, which perform plant operations, such as assembly-line production. In oil-and-gas we have to deal with, e.g., exploration and seismic data maps, and the systems interacting with the devices and sensors, which perform plant operations, such as drilling exploration wells, and oil-and-gas production. In NPP - unit assembly maps, technical conditions, and the anthropic-oriented systems interacting with devices and sensors, which perform plant operations such as heat generation and reaction unit temperature/pressure control.

The data layer is represented both by databases and data warehouses. Naturally, that includes DBMS with data mining plug-ins, analytical and online transaction processing (OLAP/OLTP), middleware, and enterprise content management tools. In case of the enterprise software-and-hardware system, essential features of the data layer are: (i) big data size, (ii) high availability, and (iii) data base/warehouse heterogeneity. Therewith, the heterogeneity can be subdivided into architectural (such as non-normalized data, legacy systems data), and structural (such as weak structured flows of audio- and video data, and scanned documentation). For instance, in NPP there are custom-integrated with both domain-specific PLM and ERP applications for NPP design and production lifecycle management, electricity production and distribution, and with online 6D modeling and data visualization tools (including 3D-visualization of the units to be designed). Heterogeneity, as usual, is both architectural and structural [8]. Below the data layer, one more layer can be identified, which is the "hardware" layer. It includes devices such as programmable logic controllers, sensors and the like, including human-machine interfaces. The hardware layer operates in terms of analog data, which is aggregated at the Layer 0, where it is converted into digital form, stored, and used for enterprise applications.

3. ENTERPRISE PROCESS KNOWLEDGE BASE AS A KEY COMPONENT FOR AGILITY

As shown above, large enterprises usually use a broad range of different software systems. Business processes also often involve the usage of multiple systems. On the other hand, each system introduces new datasets that usually have a unique format but, possibly, semantically intersect other datasets. In order to automate process and data integration tasks the enterprise needs a special software component that could serve as a knowledge repository and provide tools for building both top-down and bottom-up integration solutions. This calls for an

enterprise-grade integration bus, and with it come a whole zoo of integration processes that require managing and maintaining.

Process knowledge base of the enterprise (PK-Base) is designed for consolidation, storage, handling and use of standard and most successful processes of the organization, as well as to provide information on existing standard methods of solving problems.

The system allows simplifying the solution of typical problems encountered in the work of employees, significantly increases productivity by reducing the cost of the search for solutions to reduce the risk of suboptimal decision-making, enables the reuse of results of performed work, simplifies the training of new employees.

The proposed solution has three main Components (modules):

1) Base processes that describe the company's activities. Processes are stored as reference models for each type of action, inputs and outputs, as well as optional additional metrics such as cost of performing step and links to related processes or processes of the lower level.
2) The business goals processing module. The user can formulate a request to the knowledge base in the form of goals seeking for an algorithm to achieve what he needs to get. The answer from KP-Base would be a process, built from the available primitives, allowing to achieve this goal. All the related documents and objects of the conceptual model are applied to this constructed process.
3) Integration Module. KP-Base can be used to automate processes. Some process steps may be associated with performing reception or transmission of data in one of the available methods of implementation. Steps of obtaining information from a known source could be automated, as well as sending the results of step or process, invitations, notifications of errors or completed action, etc.

The PK-Base plays a key role in achieving agility within the enterprise. It offers a repository of formal specifications and their implementations and tackles the lack of documentation and excessive volatility problems of agile systems. It comes useful in both top-down and bottom-up process construction when processes and/or artifacts are either are build based on their formal specification through either composition of existing artifacts processes (in a self-organized way by employees that are subjects of the corresponding tasks [10]), or through consecutive decomposition of more abstract constructions

4. TYPE-THEORETIC APPROACH TO PROCESS KNOWLEDGE MANAGEMENT

Now as we described the general architecture of the PK-Base and its key role in improving enterprise agility let us consider an approach to implement such a system. We will think of enterprise process model as of computational objects that could be represented in some form of computer programs. In order to preserve high agility we need to use tools that allow us to automate both the synthesis of processes according to some formal specification (top-down approach) and the analysis of existing composite processes by inferring or verifying their specifications (bottom-up approach). Both problems need to be addressed with minimal overhead costs. Thus, we have to use tools that allow us to make logical inferences about programs. In computer science, the necessary means are provided by type theory.

There is a connection between type theory and formal logics known as Curry-Howard isomorphism [11]. It enables one to formulate the integration tasks as logical formulae and then automatically or semi-automatically connect them with software solutions. Same formal logic approaches could be used both for process and data integrations [12, 13]. The logical formula that is used in the process integration task is a formal statement of the corresponding business goal while the formula that is used in data integration is a concept description.

A goal (of a process) represents a logical description of a state that must be achieved by the organization which automates a process. The goal formalization represents a high-level specification for the process being constructed. In other words the goal description answers the question “What the process does?” and the resulting process model answers the question “How should it be done?”. Thus, the tasks of goal description and process modelling require different knowledge and skills: the goal description requires a high-level problem domain knowledge whereas the process modelling requires knowledge of technologies and programming skills.

A concept description is a statement that describes distinctive features of a class of data objects. A set of logical concept descriptions forms an ontology. Complex descriptions are built from simpler ones using a set of special operations called concept constructors. Different sets of constructors correspond to different logical systems which are formalized by description logics [14].

Both process goals definitions and concept descriptions — i.e. formal specifications — could be represented by logical formulae. If a complex integration solution is to be built or generated from scratch the corresponding formal specification could also be pretty complex. This fact could make the whole task as complex as programming the software by hand.

The key to achieve the required agility is the possibility to split the while process of generating an integration solution into several steps of specification refinement. One could start with simplest definitions and then automatically or semi-automatically refine them. Once the specification is precise enough the resulting solution will be automatically generated. From the formal point of view the fact that the specification S_2 refines the specification S_1 corresponds to the implication between the corresponding statements, i.e. $S_1 \rightarrow S_2$.

Subsequent specifications could differ in several ways. First, they could have different detailing level: the reified specification can describe more steps, properties or components of the corresponding process or concept. For example, the basic process goal definition could be described as “Hire an employee with skill Sk_1” and then reified to “Hire an employee with skills Sk_1 and Sk_2 then sign up this employee in the new employee training program”. In the field of data integration, a simple concept description could look like “Oil exploitation rate” and the more detailed version — “Oil exploitation rate on south region sources”.

Second, the reified specification can be described in more complex and expressive logic system. I.e. simple specifications can be described in propositional logic while complex specifications could use quantifiers, i.e. predicate logic.

Different types of logics: modal logics, temporal logics, linear logics etc. could be used to describe the required properties of specifications or achieve certain features such as decidability.

Type Theory Application

Let us recollect some basic definitions of the type theory which we will be using further. Type system is a flexible syntactical method of proving nonexistence of certain kinds of behavior in a program using classification of language expressions according to the kinds of in values they compute [15].

More formally, the Type Theory (TT) studies processes of type inference and type checking in programs. For this purpose, it is necessary to have a formal representation of programs — λ-calculus, where programs are interpreted like the composition of computable functions. We will be giving only major definitions omitting details that can be found in [16].

Basic constructs in λ-calculus — basic λ-terms — are defined as follows. Variables are denoted by arbitrary strings of letters and numbers. Constants are also denoted by strings. We will distinguish them based on a context. Abstraction of λ-term M by a variable x — λx.M is a unary function of parameter x. Application — is an application of a function (which is a term) M to an argument N (which is another term) and is denoted as (MN). Braces have left associativity and can be omitted if possible.

The key moment here is a concept of a function as an object. This, in particular, relieve from the necessity to consider functions with many arguments — they can be regarded as a function of single argument, computational result of which is a new function and so on.

Basic rule of computing ("reduction") a value of expression is (λx.M)N = M[x := N], where M[x := N] is a result of substituting all free occurrences of x for N in M. This rule is also equipped with a set of rules that enable reduction of not only a full term but of it's parts as well.

Types are defined as follows. If V is a type variable or constant then V is a type. If V and U are types then V → U is a type.

Finally, the typing rules. Let Γ be some context, then Γ ⊢ m : V means that term m has type V in a context Γ. For instance for simple terms like variables this is stated explicitly. For the consideration of architecture at the top level, without details of the implementation, it is enough to observe simply typed λ-calculus, which has the following system of typing rules:

$$\frac{\Gamma \vdash t: V \to T \qquad \Gamma \vdash u: V}{\Gamma \vdash t\,u: T} \quad \text{(Application)}$$

$$\frac{\Gamma, m: V \vdash n: T}{\Gamma \vdash \lambda\, m\,.\, n : (V \to T)} \quad \text{(Abstraction)}$$

Formalizations

From the Curry-Howard correspondence point of view the arrow symbol in logic corresponds to the arrow symbol in types, i.e. the formula $A \to B$, "A implies B", corresponds to the type of functions from A to B. Types correspond to formulae and terms of these types correspond to proofs of these formulae. If we could construct a function of type $A \to B$ it will, given an object of type A (a proof of A), compute an object of type B (a proof of B) — thus it will prove that A implies B.

In the area of process integration functions of type $A \to B$ correspond to processes that transform some state A into the state B. In the area of data integration it is a transformation function that transforms instances (objects) of concept A into instances of B. In description logics taxonomic relation between concepts $A \sqsubseteq B$ also correspond to implication $A \to B$ — this exhibits the fact that taxonomic relations are just a special kind of systems of transformation functions. These functions are called 'casting functions' and transform the representation of an object from more to less specific concept. Besides transformation functions we can formalize data extraction and loading functions. Their simplest types are $Unit \to A$ and $A \to Unit$. Here A is a type, which corresponds to the concept of extracted/loaded data. Unit is a special type with only one value. We use the type Unit as input or output value types of functions that do not accept or do not return any values. In other words, the value of type Unit has no information attached to it and is unique because there is no way to distinguish it from other values of this type.

Let us briefly describe some methods that enable the automatic integration solution construction. Namely, we will consider the usage of functional composition and product operations to construct terms of the given type. The composition of functions f and g is a function $f \circ g = \lambda x. fgx$. The composition operation (. ∘ .) is associative: p ∘ (q ∘ r) = (p ∘ q) ∘ r, and therefore, the parenthesis may be omitted. The following rule for typing the composition is valid:

$$\frac{\Gamma \vdash f: C \to B \qquad \Gamma \vdash g: A \to C}{\Gamma \vdash f \circ g: A \to B}$$

In order to simplify the definition of product we will introduce the notion of ordered pair (a, b) of type $A \times B$ if $a: A$ and $b: B$.

The product of functions f and g is a function $f \times g$ which, given a pair (x, y) computes the pair (fx, gx). The corresponding typing rule is as follows:

$$\frac{\Gamma \vdash f: A \to B \qquad \Gamma \vdash g: C \to D}{\Gamma \vdash f \times g: A \times C \to B \times D}$$

Practical meaning of the described operations could be defined in the following way. In process integration, the functional composition is the sequential execution of separate processes to achieve the required state via an intermediate state. A product of two processes is a process that executes them in parallel while the type-level product $A \times B$ is the product of states, i.e. a state when both states are present simultaneously.

In data integration, a composition corresponds to chaining different transformations. $A \times B$ is the concept of pairs of objects from and from and the product of transformations is a transformation between pairs of objects. Pairs could be nested to represent tuples of arbitrary size.

Typing rules for composition and product could be understood as rules for generating the terms of given type. If we read these rules bottom-up we see that in order to construct an object of a special type we need to construct two objects of other types and then connect them using the given operation. This idea can be straightforwardly implemented as a function that constructs terms of a given type in a given context.

The described combination of functional composition and product operations represent a powerful mechanism to automate the generation of sequential functions that use 'parallel' branching via products. Still, the 'separation' and 'joining' functions of types like P→ $A \times B$ and $A \times B \to Q$ must be provided manually. It can be avoided by including more operations into our basis which requires additional analysis and is not included into this brief survey.

From the practical point of view we constructed a mechanism to automate the generation of sequential or parallel processes which can branch if atomic 'separation' and 'joining' processes are provided; sequential or parallel data extraction-transform-load tasks which are automatically combined with packing/unpacking objects to/from tuples ('separation' and 'joining' functions).

Formalizing Patterns

Further analysis of occurring artifacts would very likely yield certain typical configurations, or patterns, that multiple processes followed. These patterns might span across multiple enterprise activities, or they may be specific to certain domains. Such patterns are useful in multifold ways; when it comes to enterprise agility, if there is a number of processes constructed using a certain pattern, it suffices to alter just the pattern to propagate changes to all the relevant processes.

In general, a pattern is a function $\lambda x_1 \dots x_n . p$, where x_1 through x_n are certain parameters and p is a process definition in which $x_1, \dots, x_n$ usually occur at least once. The occurrence of $x_1, \dots, x_n$ is not mandatory from the formal point of view; unused parameters normally should be eliminated, but may be (temporarily) retained for maintaining legacy patterns/processes. The term p defines the structural aspect of processes, and pattern type describes certain constraints applied to processes. Consider a number of examples.

Given a process we will write $p = \mathcal{P}(p_1, \dots, p_n)$ to denote that p is constructed using the pattern $\mathcal{P}$ from subprocesses $p_1, \dots, p_n$.

As a first example, consider we want to specify that processes of certain kind must necessarily contain a certain step. For example, every contract, before been approved by executive council (C), must be preliminarily reconciled by the technical (T) and marketing (M) directors (we don't care in what order), and even before that – ratified by the financial department (F). The corresponding pattern would look like this:

$$\lambda p{:}\, A \to F' f{:}\, F' \to F\ t{:}\, F \to T\ m{:}\, F \to M . c \circ (t \times m) \circ f \circ p$$

The process structure is this: $\mathbb{t} = c \circ (t \times m) \circ f \circ p$. p stands for an arbitrary process that prepares a contract for review by financial departments. f is a process of reviewing and approving the contract within the financial department (assuming that there may be more than one way of doing so; otherwise, the parameter f should be replaced with a corresponding constant). t and m are reviewing processes for technical and marketing directors, respectively. The part $t \times m$ indicates that the processes are performed independently and in parallel; again, we assumed there may be more than one way of doing each work, otherwise a corresponding constant would have been used. Finally, $c{:}\, T \times M \to C$ stands for the final approval procedure by the executive council, which is fixed. If at some point we decided to change that procedure, we would not have to review all the processes involving this procedure, we would go away with just replacing the process c with another one of the same type.

Supposing we want to build a specification of an integration processes that provide transition from state A to state B, and we want such processes to satisfy a specific condition that they must pass through a given state C. A corresponding pattern would look like that: $\mathcal{P} = \lambda(p_1{:}\, A \to C\ p_2{:}\, C \to B) . p_2 \circ p_1$.
The type of this pattern is $(A \to C) \times (C \to B) \to (A \to B)$. It means that if we want to construct a process from state A to state B using this pattern, we must first come up with two (possibly, composite) processes: one from the initial state A to the intermediate state C, and the other from the intermediate state to the final state B.

Of course, if we decided to alter the pattern $\mathcal{P}$ so as to require corresponding processes to pass through a certain state D instead of C (let's call the new pattern $\mathcal{P}'$), we would also have to come up with two transformations for all processes of types $A \to C$ and $C \to B$:

$$\mathbb{t}_1{:}\, (A \to C) \to (A \to D),$$
$$\mathbb{t}_2{:}\, (C \to B) \to (D \to B).$$

Now, if we are given a number of processes $p^i = \mathcal{P}(p_1^i, \dots, p_n^i)$ and we want reengineer those processes so that they would conform the new pattern $\mathcal{P}'$, we will, in fact, need a procedure (a process) of type:

$$\mathbb{T}{:}\, \big((A \to C) \to (A \to D) \times (C \to B) \to (D \to B)\big) \to (A \to B) \to (A \to B).$$

We can read this type as 'if we need a (non-identical) transformation of processes of type $A \to B$, we need a couple of additional processes – one that transforms processes of type $A \to C$ to processes of type $A \to D$, and the other that transforms processes of type $C \to B$ to processes of type $D \to B$'. We silently presume that we consider only those processes of type $A \to B$ that are built using the pattern $\mathcal{P}$. If there are others, we use the technique that in functional programming is known as pattern-matching: we write the transition $\mathbb{T}$ in such a way that it accepts only arguments of the kind $\mathcal{P}(x, y)$ (where x and y are variables of types $A \to C$ and $C \to B$, respectively), and skips all other objects (of type $A \to B$). The fact that a certain object is constructed via a certain pattern is not usually reflected in object's type, but may be conveyed through other kinds of metadata, if necessary.

In λ-calculus, we do not particularly care if we deal with functions or data objects – so long as all type constraints are satisfied. It means that if certain technique is applicable to processes, than it is also applicable to data objects.

Many of the artifacts enterprise deals with on daily basis are structured objects, like contracts or other documents. What if we wanted to produce a specification on the structure that all contracts of certain kind (K) must conform to? For example, we want all contracts to include four sections: general terms (G), financial obligations (F), technical terms (T) and signature list (S). The signature list consists of signatures of financial, technical and marketing directors (S_f, S_t and S_m, respectively) and signatures of tree members of executive council: S_{c1}, S_{c2}, and S_{c3}. So we can write:

$$K = G \times F \times T \times S,$$

where

$$S = S_f \times S_t \times S_m \times S_{c1} \times S_{c2} \times S_{c3}.$$

The corresponding pattern for such contracts, in its simplest form, would be like this:

$$\mathbb{T} = \lambda g{:}G\ f{:}F\ t{:}T\ s : S.(g, f, t, s),$$

which has type

$$G \to F \to T \to S \to G \times F \times T \times S,$$

or, in this case equivalently,

$$G \to F \to T \to S \to K.$$

Now, suppose that general terms are written by a secretary, except one last clause which is provided by, say, the marketing director; in this case, assuming that G^i stands for the type of general term item and that $G = [G^i]$ (general terms section is a list of general terms), secretary's work should be a process looking like this: $\lambda l{:}\left[G^i\right].\lambda i{:}G^i.(l@i, f, t, s)$. Its type is $\left[G^i\right] \to (G^i \to F \to T \to S \to K)$. If choosing specific template were also part of secretary's work, then the corresponding type would have been: $(G \to F \to T \to S \to G \times F \times T \times S) \to \left[G^i\right] \to \left(G^i \to F \to T \to S\right)$.

In any case, when the secretary has done his part, we have a partially filled template – an object of type $G^i \to F \to T \to S \to K$. If, for example, the contract had to be processed by the financial department, who had to supply financial terms for the contract and sign the contract of financial director's behalf, the corresponding process would have the type: $\left(G^i \to F \to T \to S \to K\right) \to (G^i \to T \to S_t \times S_m \times S_{c1} \times S_{c2} \times S_{c3} \to K)$. After that, the treaty would be passed to, e.g. technical and marketing directors, and then to executive council.

These examples illustrate that, theoretically, we can automatize, to a certain degree, the very process of reengineering enterprise architecture based on shifts in strategic goals.

5. CONCLUSIONS

We have presented a general enterprise architecture that is aimed at raising the level of business agility. The key role belongs to the process knowledge base that accumulates business process and data integration artifacts as well as their specifications. The PK-Base reduces the "communication chaos" and provides tools for automatic or semiautomatic integration solution generation both in a top-down as well as in a bottom-up manner.

We give a type-theoretical basis to construct the required tools for automatic synthesis and analysis of integration solutions. Semantically the proposed approach is similar to automated type-based program generation, which was studied in previous decades. Our plan is to extend this theoretical approach to solve real-world process and data integration problems.

9. REFERENCES

[1] Zykov S.V. (2015) Enterprise Applications as Anthropic-Oriented Systems: Patterns and Instances. In: Proceedings of **9th KES Conference on Agent and Multi-Agent Systems: Technologies and Applications**, Springer, 2015, pp. 275-283.

[2] Dyer, L. and Ericksen, J. (2009). Complexity-based Agile Enterprises: Putting Self-Organizing Emergence to Work. In A. Wilkinson et al (eds.). **The Sage Handbook of Human Resource Management**. London: Sage: 436–457.

[3] Gromoff A., Kazantsev, N., Kozhevnikov, D., Ponfilenok, M. and Stavenko, Y. (2012). Newer Approach to Create Flexible Business Architecture of Modern Enterprise. **Global Journal of Flexible Systems Management**. 13(4), Springer-Verlag, 207-215

[4] Gamma E., Helm R., Johnson R., Vlissides J. (1998) **Design Patterns CD: Elements of Reusable ObjectOriented Software**, Addison-Wesley, 1998

[5] Fowler M. (2002) **Patterns of Enterprise Application Architecture.** Addison-Wesley, 2002

[6] Freeman E., Bates B., Sierra K., Robson E. (2004) **Head First Design Patterns**, O'Reilly, 2004

[7] Hohpe G., Woolf B. **Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions**. Addison-Wesley, 2004

[8] Lattanze A. (2008) **Architecting Software Intensive Systems: A Practitioner's Guide**. Auerbach, 2008

[9] Zykov S. (2009) Designing patterns to support heterogeneous enterprise systems lifecycle. In: **Proc. 5th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR)**, 2009

[10] Zykov S. (2010) Pattern Development Technology for Heterogeneous Enterprise Software Systems. **Journal of Communication and Computer**, 2010, 7(4), 56-61

[11] Fleischmann, A. (2010). What Is S-BPM? S-BPM ONE — Setting the Stage for Subject-Oriented Business Process Management. **Communications in Computer and Information Science**. Heidelberg: Springer Berlin.

[12] Sørensen M. H., Urzyczyn P. (2006) **Lectures on the Curry-Howard isomorphism**, Vol. 149. Elsevier, 2006.

[13] Shapkin P., Marenkov A., Shumsky L., Roslovtsev V., Wolfengagen V. (2015) Towards the Automated Business Process Building by Means of Type Theory. Proceedings of the **7th International Conference on Subject-Oriented Business Process Management S-BPM ONE**. 2015.

[14] Shapkin P., Pomadchin G. (2015) A Type-Theoretic Approach to Cloud Data Integration. In: **Proceedings of the 11th International Conference on Web Information Systems and Technologies WEBIST**. INSTICC Press, 2015.

[15] Baader F., et al. (2007) **The Description Logic Handbook: Theory, Implementation, and Applications**. Cambridge: Cambridge University, 2007.

[16] Pierce B. C. (2002) **Types and programming languages**. The MIT Press, 2002.

[17] Barendregt H. (1991) Introduction to generalized type systems. **Journal of functional programming**, 1(2):125–154, 1991.