A compositional approach to building applications in a computational environment

To cite this article: V V Roslovitsev et al 2014 J. Phys.: Conf. Ser. 495 012050

View the article online for updates and enhancements.
A compositional approach to building applications in a computational environment

V.V. Roslovtsev, L.D. Shumsky and V.E. Wolfengagen
Department of cybernetics, National Research Nuclear University “MEPhI”,
Kashirskoe sh. 31., Moscow, Russia
E-mail: vladiros@gmail.com, shumskyld@gmail.com, vew@jmsuice.msk.ru

Abstract. The paper presents an approach to creating an applicative computational environment to feature computational processes and data decomposition, and a compositional approach to application building. The approach in question is based on the notion of combinator – both in systems with variable binding (such as λ-calculi) and those allowing programming without variables (combinatory logic style). We present a computation decomposition technique based on objects' structural decomposition, with the focus on computation decomposition. The computational environment's architecture is based on a network with nodes playing several roles simultaneously.

1. Introduction

Today software engineers face a number of challenges: the evergrowing complexity of information processes and related computations in various subject areas, the exponential growth of processed data volumes coupled with growing meta-data volume rate and complexity; many of the problem domains has a high degree of mutability as their distinguishing feature, and that means one has to upgrade the model of the problem domain and its processes in a timely manner. Under such circumstances a key to information and computation systems efficiency is computational processes decomposition and reuse. That leads to two relatively independent problems:

• (computational) task formulation, preferably, introducing to the model as few elements as possible, which requires a high degree of task decomposition and reuse of previously defined constructs;
• definition of at least one solution to each identified task, and again, it is desirable to reuse existing code as much as possible to lessen costs and risks associated with additional testing and integration.

Solving the first of these tasks leads to conceptual integrity of the software and to minimality of the conceptual framework of the subject area, and its increased potential for analysis. Dealign with the second problem leads to various kinds of open architectures which, on one hand, ensure a relatively high degree of reusability, and on the other, they go a long way towards systems’ extensibility and adaptability.

In the present paper we focus on a specific object (and in particular computational processes) decomposition methodology which leads to a particular design approach. We use this approach...
to design an environment suitable for an organization to host and execute a family of its (internal and otherwise) applications. The need for such environments occurs when

- a unified, domain-specific (organization-specific) framework for software development and support is required;
- hardware (e.g. high-performance CPUs and graphics cards, perhaps, sensors or other devices as well) and software (e.g. libraries for mathematical calculations) resources are to be integrated, with a centralized and controlled access to them;
- there particular requirements for scalability, fault-tolerance, performance, flexibility (e.g. configuring a system to run as prototype for demonstration or as a high-performance fault-tolerant cluster).

Our methodology is based on applicative computational systems (ACSs, they include systems based on λ-calculus and combinatory logic, see [1]), because they implement what is called applicative computing: a way of performing computations that considers a computational block as a closed, relatively independent, entity with no free variables. The block are combined so as to create new, more complex computations.

The rest of the paper is organized as follows. Section 2 contains a brief introduction to ACSs basics and outline the computation decomposition methodology. The section 3 deals with object interaction semantics, and section 4 outlines the design of the mentioned computational environment. In section 5 we present a brief related work review and a conclusion.

2. Computation decomposition

2.1. General considerations

In the present paper the term “applicative computational environment” denotes a set of objects (terms) defined by the corresponding applicative computational system (ACS). ACSs may be thought of as implementations of what is called the applicative computing, which suggest the “compositional way of constructing computational blocks out of previously constructed ones, the blocks being relatively closed and containing no free variables” [14]. An ACS is a formal system, generally based on combinatory logic or λ-calculus, with the following characteristic features:

- objects in ACSs are functional entities with undefined arity\(^1\), that shows itself in course of computations;
- the main way of constructing new objects from existing ones is the application operation, where the object-function is applied to the object-argument;
- the function-argument roles are relative to the context in which an object appears, so that even self-applicability is allowed (in untyped systems; in typed systems this is often prohibited).

Various applied systems may be obtained by extending the set of initial objects. In some cases there may be more rules for producing complex objects, e.g. in categorical combinatory logic one would also add rules for pair \([·, ·]\) and pairing \(⟨·, ·⟩\) constructors. This suggests [11] that an environment’s definition comprises two aspects: the set of initial objects and the set of ways of combining objects. A generalized scheme for environment definition may be suggested then [8, 9]:

\[(i) \text{ (induction basis)}\]

\(^1\) This applies mostly to complex objects in untyped systems; in practice, when typed system are used, objects arity is either part of objects definition, or may be derived from the object’s structure.
(a) there is an infinite set of variables and a set (possibly, empty) of atomic objects,
(b) variables and atomic objects are atoms, and all atoms are objects;

(ii) (induction step)

(a) there is a predefined set of term-forming entities pertaining to the meta-level (functions, as they are called), including, at least, the application operation;
(b) if \(A_1, \ldots, A_n\) are objects and \(\sigma\) is an \(n\)-ary term-forming function (TFF) then \(\sigma A_1 \ldots A_n\) is an object.

Basically, given a (finite or infinite) set of initial (better still, primary) objects and a finite set of ‘constructors’ (ways of combining existing objects into more complex ones), one consecutively construct new objects out of existing, simpler ones, much like a building is constructed from bricks, panels and blocks, or a machine is assembled from parts, which, in turns, consist of other, smaller, parts. Combinatory logic and \(\lambda\)-calculus each embodies a distinct approach to object construction. That of combinatory logic (CL) follows the idea that any computable objects may constructed from constants exclusively, which ensures that:

a) there are no free variables in an object,
b) and hence, the object is closed in that it does refer to any part of the outside environment, eliminating any otherwise possible side effects.

Variables in CL are used for ‘auxiliary’ purposes, such as combinatory characteristics representation (i.e., speaking from the perspective of the classical logic, representing the rule of introduction-elimination of the symbol – ACSs are equational systems, and objects conversion is a bidirectional process). In \(\lambda\)-calculus, on the contrary, variables are used quite extensively, and one of the main things that \(\lambda\)-calculus focuses on is variable binding and substitution. Briefly, the \(\lambda\) is a binary operator that binds a variable (the first argument) in an object (the second argument); a variable in an object that is not bound is free in that object. \(\lambda\)-calculus, thus, implements explicit control over object’s dependencies on the outside environment. On the conceptual level, with respect to objects construction, the difference between CL and \(\lambda\)-calculus is that CL focuses on modeling entities within a predefined framework, whereas \(\lambda\)-calculus makes it possible (e.g. via supercombinators, see [12]) to ‘generate’ a suitable basis on-the-fly.

2.2. Initial objects

Usually, part of a definition of a specific ACS (which is a formal system) is a specific class of initial objects of which all other objects are built using a specified set of constructors (or TFFs, as we call them). It may be so that some the initial objects may be decomposed and ‘expressed’ using other initial objects; some of the initial objects may be atomic in the sense they cannot be decomposed into smaller units without loss of the general logical integrity. These latter, atomic objects, we will also call primary. Initial objects that are not primary we call secondary, and all other objects are derived [14].

The question of choosing the set of atomic objects is a good one, and two points are worth noting here. The first is that, actually, objects atomicity is a matter of motivated assumption: further decomposition of such objects is either redundant or leads to inconsistencies [13]. Second, though one may choose different suitable sets of atomic objects for different purposes, those sets would share a common computing foundation – for which combinators K and S are good candidates, as argued in [14]. These objects have a sound interpretation on various contexts, and they offer a relatively easy way to construct relatively complex entities, like lists.

2.3. Process constructors

Our approach, to an extent, was inspired by Dana Scott’s flow diagrams [11]. A program may be seen as something, called ‘flow diagram’, through which information would flow, subjected
to some transformations in the process. These diagrams are made of elementary, atomic transformation units connected via arcs that do not affect the information flowing through them and that transfer the information from one block’s output to another block’s input. From the mathematical point of view, these elementary transformation blocks are atomic combinators, and connectives represent the composition operation \((f \circ g) = f(gx)\). In ACSs this operation is actually an ordinary combinator (a detailed explanation may be found in section 2.2 in [12]), so that a program is basically a term in an ACS (a combinator, preferably). In [11] it is also shown that some blocks may function as those ‘controlling’ (directing) the flow, conditional branching and parallel switch been the most obvious examples; they may be called ‘diagram constructors’ in that they join several smaller diagrams (branches) into a bigger one. A generalization of that idea was presented in [8], and later a somewhat refined version, concerning both computational processes and data objects, in [9].

Some of the above ideas are commonly used in (functional) programming languages, but hardly so when it comes to subject domain modeling and designing software architecture\(^2\). However, the idea of using the combinator model for computational processes, both at conceptual level and in implementation, has a number of obvious advantages:

- every process is closed, i.e. has no implicit dependencies or references to the outside environment\(^3\);
- because of that, too, and because a process is, more or less, an ordinary object, an existing process, be it simple or complex, may be re-used to define other processes;
- a process may be passed to another process as an argument, thus allowing computation adjustment at run-time;
- very powerful, yet relatively simple, process algebras may be formulated.

With the idea of computation distribution in mind, in [9] it is suggested that data processing should proceed in three stages. First, the input object is decomposed onto several components and each of them, secondly, is sent to a corresponding specialized (sub)process (which may be either a structural part of the given process, or an independent entity). Finally, the partial results yielded by the ‘subprocesses’ are aggregated in some way, e.g. ‘wrapped’ with a TTF into a resulting object. One powerful approach in process construction is defining a set of ‘standard’ process ‘schemes’ to combine the already constructed processes into a more complex one.

Let \( F : T_1 \rightarrow \ldots \rightarrow T_m \rightarrow T_{m+1} \) be a TFF. For every such TFF let us define a set of projecting objects \( \pi^f_i : T_{m+1} \rightarrow T_i \) (with \( i \in [1, m] \)), such that:

\[
\pi^f_i(Fa_1 \ldots a_m) = a_i,
\]

– for every suitable set of objects \( a_1 : T_1, \ldots, a_m : T_m \). Finally, consider the currying ‘operator’ \( \mathbb{H}^n \) with the combinatory characteristics:

\[
\mathbb{H}^n f[x_1, \ldots, x_n] = f x_1 \ldots x_n
\]

Assume that \( \langle f_1, \ldots, f_n \rangle x = [f_1 x, \ldots, f_n x] \) holds for every suitable (e.g. type-compatible, according to a certain type system of choice) set of objects \( f_1, \ldots, f_n \) and \( x \). Then a process that takes an object \( Fa_1 \ldots a_m \) and yields another object \( Gb_1 \ldots b_l \) would in general look like this:

\(^2\) We believe that the reasons for that are more or less the same as why functional programming, despite its growing popularity and everdeeper impact on programming culture, is still rejected by many developers as being ‘excessively theoretical’.

\(^3\) In real systems, there are many cases when side effects are necessary, e.g. logging; these issues are addressed through a more complicated type system which takes in account side affects.
\[ \mathcal{F} = \mathbb{H}^l G \circ (g_1, \ldots, g_l) \circ (f_1 \circ \pi^1_\mathcal{F}, \ldots, f_m \circ \pi^m_\mathcal{F}) \]  

(1)

or

\[ \mathcal{F} = \mathbb{H}^l G \circ (g_1 \circ f_1, \ldots, g_l \circ f_l), \]  

(2)

where all \( f_i \) are subprocesses that perform a certain pre-processing of the input (on a by-component basis in case of (1), and in case of (2) on a more general one). The responsibility of subprocesses \( g_1, \ldots, g_l \) is to yield individual components \( b_1, \ldots, b_l \) of the resulting object. Finally, \( \mathbb{H}^l G \) unpacks the tuple \([b_1, \ldots, b_l]\) and produces the final result. Of course, both in (1) and (2) one may add explicit post-processing handlers. An essential point is that every component \( b_1, \ldots, b_l \) is evaluated independently of all others, which is rather handy in parallelized environments. If, on the other hand, some of these components depend on the others, an appropriate scheme may also be devised, but it is likely to be more complex and, obviously, yield a lower parallelization rate.

In (1) and (2) every \( f_1, \ldots, f_m, g_1, \ldots, g_l \) may be either an internal part of \( \mathcal{F} \) or an external process (and may be replaced independently), or it may stand for a ‘process variable’ \( p_i (i = [1, k]) \), or contain its (free) occurrence(s). Functional abstraction of \( \mathcal{F} \) by such variables will produce a scheme:

\[ \hat{\mathcal{F}} = \lambda p_1 \ldots p_k . F \]

Given already constructed processes \( p_1, \ldots, p_k \), one may construct a new complex process \( \hat{\mathcal{F}} p_1 \ldots p_k \). Note that, actually, the scheme \( \hat{\mathcal{F}} \) may be replaced with another one, provided that all typing constraints are satisfied. Next, very naturally (most type systems would allow that) one can introduce variables ranging over schemes, thus leading to ‘scheme constructors’.

Most of those schemes and scheme constructors are likely to be domain-specific and the purpose of their introduction is to facilitate (automate, if possible) the construction of processes in such environments where:

- the number of processes is exceedingly high and/or processes structure is very complex and requires extensive decomposition;
- processes makeup and/or structure is unfixed by the domain nature and adjustments are to be made in a timely fashion.

Further discussion as to the TFFs’ internal structure is to be found in [10].

3. \( \pi \)-calculus as process invocation semantics

Having a number of computational objects in an environment, it is then necessary to define how they are supposed to interact with each other. Conventional ACS give little or none attention to this question, presuming all objects being in the same ‘space’ and thus able to act ‘directly’ upon each other. This seems to work anyhow for both the lower level of the operational semantics, which is exactly what the numerous abstract machines are, and on the higher level of program architecture – so far as ‘monolithic’, (more or less) single-threaded programs of the old days are concerned. However, things get much more complicated for parallel and distributed systems. It is where the \( \pi \)-calculus seems a worthy candidate to help to formalize computational objects’ interaction.

\( \pi \)-calculus is an extension of the standard calculus of communicating systems (CCS) allowing advanced process description: sending (and retrieving) named data blocks over named channels, the channels themselves being legal ‘data blocks’. Every process in this theory is a sequence of subprocesses that either send or receive data blocks (purely computational processes are not
considered). The main connectives to build complex processes out of simpler ones are: sequential execution, parallel execution and conditional branching. The main advantage of \( \pi \)-calculus is ability to describe processes with dynamic structure, i.e. process whose structure is determined in runtime depending on execution progress.

In our approach we consider computational processes which are represented by closed applicative objects (combinators), therefore we suggest using an extension of asynchronous polymorphic \( \pi \)-calculus as process execution semantics. Using semantic of \( \pi \)-calculus brings several benefits and we cover the most important of them in this part of paper. This calculus was designed for modelling interaction between systems [5, 6]. Its models better suits for information process, so it is required adaptation and a certain extension of the standard calculus model. The main aim of using \( \pi \)-calculus is to formalize not only data exchange but processes components execution as well. Also, such an extended model allows defining processes that involve data exchange between several systems. Assuming (1) as process’ scheme and \( X \) standing for a certain input object of \( \mathcal{F} \), here below are stated the main process execution rules:

(i) Every ‘projection’ mapping \( \pi^k_x \) is translated to \( a_i (\mathcal{X}) \cdot ([\pi^1_x \mathcal{X}] 1) \ldots ([\pi^n_x \mathcal{X}] m) \), i.e. execution of the process \( \mathcal{F} \) begins with distribution, over a number of channels \( a_i \), data required to get input objects components. The statement \([|Kx_1 \ldots x_n]|c\) means execution of the combinator \( K \) with a mount point \( c \). A mount point is a channel through which a function would return its result. This way of execution functions in processes by modeling int \( \pi \)-calculus is known as “encoding” [4].

(ii) Every subprocess \( f_1, \ldots, f_n \) must not contain any data exchange actions, i.e. it must be a purely computational, closed (at least, within \( \mathcal{F} \) process, in which case the following holds: \( f_i \rightarrow i(d).[[f_i d]]i \).

(iii) Every subprocess \( g_1, \ldots, g_l \) receives, as its input, a data block containing all the processed components of \( \mathcal{X} \) and yields a certain component of the resulting output object. Accordingly,

\[ g_l \rightarrow t(1(y_1) \ldots m(y_m).\overline{t}(z_1 \ldots z_m).|[g_l z_1 \ldots z_m]|o) \]

Note that such an execution scheme is possible in an asynchronous environment only [3].

These rules outline the use of \( \pi \)-calculus as a mechanism underlying execution of our constructive processes. If all computations are performed by a single actor – executing the provided process is identical to ordinary reduction of a correspondent combinator. Thisings are different in heterogeneous environments. For example, if projection functions or subprocesses are executed in different systems.

The first benefit from implementing \( \pi \)-calculus semantics is implementation of complex dynamic routing of computational flows in a computational network. We face this problem of routing requests over the network when various nodes are able to execute several different types of projection functions or subprocesses. In that case the network has to be able to find the right (sub)set of nodes to execute the computational task in a correct and, and, preferably, optimal (from the resource consumption) way. Process execution model must deal with two main problems: selection of the target nodes to participate in the process execution and load-balancing. The second benefit from using \( \pi \)-calculus is runtime type checking for computations, hosted by different systems. These checks can be implemented by using typed \( \pi \)-calculus. This version of calculus assigns to every name of term some type constant and forbids interaction if types of names involved into interaction don’t suit each other.

We suggest the following changes in the \( \pi \)-calculus for implementing routing for different systems. Let \( a_i (\mathcal{X}) \cdot ([\pi^1_x \mathcal{X}] 1) \ldots ([\pi^n_x \mathcal{X}] m) \) be a process for projecting input data. This is a correct process when all projecting functions are hosted by single computation node, described by channel \( a \). Computation process doesn’t need routing in that case – all requests may be
safely sent via the channel $a$. In case of a distributed system (meaning that various parts of the system are separated, isolated from each other, not necessarily be means of a physical network) the previous scheme transforms into the following:

$$
(a_1, ..., i_n(x_1), \left(\prod_{j=1}^{i_1} \pi_{x_1}^{i_1} \right) i_1 \cdots \left(\prod_{j=1}^{i_n} \pi_{x_n}^{i_n} \right) i_n) \mid \cdots \mid (a_{i_m}, ..., i_p(x_m), \left(\prod_{j=1}^{i_m} \pi_{x_m}^{i_m} \right) i_m \cdots \left(\prod_{j=1}^{i_p} \pi_{x_p}^{i_p} \right) i_p)
$$

In order to manage this ‘zoo’ of channels $a_i$, we will introduce a centralized routing channel whose responsibility is to select valid targets for every particular message (which, obviously, involves some run-time type-checking, as was mentioned above). We define two rule for target selection. First, we use any metadata, supplied for the input data (basically, its type; note that nearly any kind of metadata may be fit into a data type, given the type system is strong enough, like that with dependent types; but this is far too complex a question to discuss here). We introduce a combinator $T x = [i_n, \ldots, i_k]$ that, given an object (and its type along it), produces a list of required ‘projections’. There is also another required combinator is $L$ which does the actual consistency checking, so that a given set of channels is able indeed to process the given set of subcomponents of a given input object. This model is implemented as such:

$$
r(x).ve(\left[\prod_{j=1}^{i_1} \pi_{y_1}^{i_1} \right] e, (y_1, \ldots, y_n) \left[\prod_{j=1}^{i_m} \pi_{y_m}^{i_m} \right] e, a_1, ..., a_{i_n}, a_{i_m}, \ldots a_{i_k})
$$

The second rule uses external knowledge about input objects and selects projection channels by the name of the passed input. This rule is applied when a computation can take inputs from a finite set of objects. It doesn’t require the computation performed by the router, but fails on objects not from the expected set. Implementation of the router uses construction of process on the case-by-case basis.

4. Computational environment design
The usage of various kinds and variations of service-oriented architectures (SOA) is beneficial in that it helps in addressing the problems we stated in the introductory section – and besides, improves software scalability, naturally supporting distributed software configurations. Note that in this paper we stick to a wider understanding of SOA which is not limited to “classical” web-services (usually hosted by web-servers and relying on SOAP and the likes of it for interaction) but includes other options as well. Services interaction may be carried out in various ways: from a “local” bus like DBus to standard technologies like WCF\(^4\) or RMI\(^5\) to custom TCP/IP-based protocols implementations.

We assume that applications are build, in a sense, in a ‘task-oriented’ manner, meaning that every application has a definite list of (formally written) tasks that it should perform. Such an application contains a data and meta-data object model (basically, a list of entities found in the subject area, their attributes and relationships) and computational processes model. An advantage of ACSs is that they provide a uniform framework suitable to represent both kind of models. We see a task analogous to a request to a database written in relational calculus: one defines what is it exactly that he or she wants to get, leaving to the system to figure out the best way to do it. In case of a relational database, such a request is translated to a computational process consisting of elementary relational operations (and, perhaps, additional operations as well).

\(^4\) Windows Communication Foundations is .NET technology to allow interaction between objects located in different, including remote, processes. Its main feature is that it offers developers high-level interfaces that abstract away many details of working with low-level network protocols.

\(^5\) Remote Method Invocation is, roughly speaking, the Java analogue for WCF.
Consequently, the environment in question must primarily contain a number of elementary
domain-specific processes, each equipped with some meta-information as to the kind of
transformation the process does. At the very least, it should be a list of ‘key words’ – references to
concepts from the subject domain model. As an option, a more interesting one though, seemingly,
yet poorly explored, this meta-information is to be included in the process type (using Curry-
Howard isomorphism).

When a user invokes a function in an application, a ‘request’, which
is basically a term in λ-calculus (or CL) either assembled on-the-fly or defined and stored in
advance, is sent to the environment for execution. This term is translated into another one
that uses only those elementary computational processes that are defined in the environment.
The environment then acts as an abstract machine and reduces this term, not unlike how the
conventional abstract machines reduce terms representing programs in functional languages.

The environment is a set of nodes forming a connected graph (each pair of connected nodes
may be connected either with undirected edges in case of a symmetric protocol, like TCP, or
with two directed arcs in case of use of asymmetric protocols, as may be the case with nodes
hosted via web services). The basic functionality of all nodes is only that they can establish
connection, interchange with messages of various kinds and break the connection. On top of this,
two mechanisms are to be implemented: task execution and event propagation. Task execution
consists in selecting a set node of nodes capable of executing a specific task, choosing one (usually)
or several (e.g. for service tasks) nodes, sending the task to these nodes and then ensuring that
the result is produced and stored for further use. Event propagation is a mechanism to inform
an intended set of nodes that a particular situation has occurred (e.g. a specific node has gone
offline, which might mean that a particular application or part of its functionality is no longer
available). Above that, a node must run one or more loadable modules (plugins, of sorts) and
according to this play one or more roles in the network, of which the main are:

- transport: routing messages throughout the network, ensuring that a message may be
delivered to any given node, including plotting optimal paths;
- decomposer/composer: a task is an object constructed from atoms using TFFs and the
same applies to the result of computations, so executing a task means, roughly speaking,
invoking the initial computing objects with some parameters in some order and combining
the partial results;
- computational: the node actually hosts a number of initial computing processes;

The set of transport nodes should form a connected graph and is called the transport system
of the network. On top of this functionality implemented are: task and result sending. Besides
these three roles, a number of other roles may be identified, important as they may be in practice,
they are secondary to the environment’s functioning, such as:

- load-balancing between node;
- gateway to enable interaction with external systems;
- caching;
- support to allow remote control over nodes, self-diagnostics, and other useful features
concerning internal management.

Another important notion is that of a resource. Resource is an object with a ‘mobility policy’
assigned to it that determines whether this object may or may not be copied, or moved to, or
accessed from another node. There are two basic applications to this. First, we may or may
not transfer data between two nodes, be it due security reasons or simply availability of storage
space to accommodate the data on the target node. The other application is that a computing
object, which is basically a piece of (compiled) executable code, may or may not be transferred
to another node – again, security considerations and code dependencies availability may be
concerned. The basic framework for formal reasoning about resources is to be found in [7], but the actual formal system is a matter of future work.

5. Conclusion
There is a number of approaches to model business/information/computational process. Many are based on network, state-diagram (e.g. Petri Coloured Nets), event-oriented models, and often either lack proper formal basis or this basis is simply ignored, the being used more for visualizing purposes. These models may be useful for subject areas analysts, but they are of littly help when it comes to software design, complicating architecture development and increasing the probability of design errors (which are most dangerous to software development). To counter this problem, a number of process algebras [2] were developed, covering a range of aspects of process semantics execution issue, including temporality.

Using ACSs offers two major advantages: it is a very natural tool for objects structure studying and representation, and it offers a conceptual basis that is much easier to translate to software architecture and then to program code, thus being a very promising option for filling the two big gaps: the one between subject area model and architecture, and the one between the architecture and the source code.

In this paper we described an approach to processes structure and, to an extent, execution semantics modeling. For process structure representation we suggested using an approach based on an applicative computational systems extension that explicitly deals with objects constructors – TFFs. This approach has an additional advantage that, using it, an ‘environment algebra’ may be developed [10], which may be used to address various problems of semantic interoperability between different applications. To counter the problem of (possibly, remote) process interaction we suggested to use the \(\pi\)-calculus and showed how ACSs terms are represented and executed in it.

We suggested an approach and an architecture outline for an environment in which applications may be continuously constructed, debugged and executed. The environment is scalable and may equipped with many useful features, including fault-tolerance. The environment is (distributed) heterogeneous computational network, nodes playing one or more roles; the network’s functioning consists in solving ‘tasks’ which are decomposed into atomic components, distributed over the nodes suitable for each atomic task, and the partial results are then aggregated into the ‘answer’ of the original task. Compared to the previous version, described in [9], the architecture has been slightly changed to increase overall consistency.

An experimental prototype is presently under development, and we hope in some near future to be able to publish experimental results.

References
[1] Barendregt, H., 2012. The Lambda Calculus, its Syntax and Semantics. Studies in Logic. Mathematical Logic and Foundations, vol. 40.
[2] Bergstra J.A., Ponse A., Smolka S.A., 2001. Handbook of Process Algebra. (Elsevier Science)
[3] Boudol G, 1992. Asynchrony and the \(\pi\)-calculus. Rapport de Recherche RR-1702, INRIA Sophia Antipolis.
[4] Milner R., 1992. Functions as processes. Mathematical structures in computer science, vol. 2, no. 02, pp. 119-141
[5] Milner R, 1993. The polyadic \(\pi\)-calculus: a tutorial. (Springer)
[6] Milner R., Parrow J., Walker D, 1992. A Calculus of Mobile Processes. Information and computation, vol. 100, no. 1, pp. 140
[7] Murphy T, Crary K, Harper R, Pfenning F, 2004. A Symmetric Modal Lambda Calculus for Distributed Computing. (Carnegie Mellon University, Computer Science Department. Paper 465. Url: http://repository.cmu.edu/compsci/465)
[8] Roslovtsev, V.V., Luchin, A.E., 2009. Concept of Higher-Order Applicative Computational Environment. Proc. of the 11th Int. Workshop on Computer Science and Information Technologies CSIT’2009, pp 48-53.
[9] Roslovtsev V., Shumsky L., Wolfengagen V., Bohulenkov A., Sakhatskiy A., 2013. Applicative Approach to Information Processes Modeling. Towards a Constructive Information Theory. Proc. of the 15th Int. Conf. on Enterprise Information Systems ICEIS2013 (Anger, France) pp 221-227.

[10] Roslovtsev, V.V., 2013. Applicative Computational Environments Construction. SWorld: Conf. Series, Issue 2, Volume 6 (also available at http://sworld.com.ua/index.php/ru/technical-sciences-213/informatics-computer-science-and-automation-213/18159-213-745)

[11] Scott D., 1971. The Lattice of Flow Diagrams. Lecture Notes in Mathematics, Vol. 188. (Springer) pp 311-366

[12] Wolfengagen V.E., 2003. Combinatory logic in programming. Computations with objects through examples and exercises. 2nd ed. (Moscow, Russia: “Center JurInfoR” Ltd)

[13] Wolfengagen W.E., 2004. Methods and Means for Computations with Objects. Applicative Computational Systems. (Moscow, Russia: “Center JurInfoR”) [In Russian]

[14] Wolfengagen V.E., 2010. Applicative computing. Its quarks, atoms and molecules. Ed. Dr. L.Yu. Ismailova. (Moscow, Russia: “Center JurInfoR”)