A biological-like synthesis framework for software engineering environment

Peraphon Sophatsathit
speraphon@gmail.com

Abstract
Background: Software architecture consists of many artifacts that encompass their own architectures. These architectures evolve over the software lifetime. Some are replaced by new versions, others are obsolete and disposed of. At any rate, they suffer from complexities and interoperability. The human body, on the contrary, is a natural wonder that works seamlessly and intelligently. By imitating the biology of a uni-cellular life form and the body's building blocks or DNA, this prospectus will furnish an autonomous software system that is independent of its working environment.
Methods: In this paper, we propose a Biological-like Architecture for Software Systems (BASS) that mimic the simplicity of a uni-cellular life form. The basic construct consists of fixed size components holding their attributes and operations arranged in a one-dimension array akin to DNA strings. This permits self-execution without external support. Since uni-cellular life form is short-live, so are the components modeled to undergo a three-stage life cycle, namely, creation, sustainment, and cessation that must be completed within a predefined Time-To-Live (TTL) limit. In the meantime, a new component is cloned to replace the ceasing one in situ. Since there is no comparable architecture to benchmark the proposed novel architecture, it is simulated to gauge the performance statistics of BASS.
Results: The combinations of fixed size, direct access, and linearly arranged like DNA string are purposely planned to advocate hardware implementation. The results show that memory occupation remains relatively low by virtue of this organization and replacement in situ scheme.
Conclusions: The contribution of the proposed software architecture is an autonomous system that serves as an efficient portable environment and can lessen software systems resource utilization considerably.
Keywords: autonomous system, fixed size, short-live, creation, sustainment, cessation, replacement in situ.

1. Introduction

Traditional software architecture presets many constituent components that possess a number of design attributes. For example, an architectural construct of a utility set could be a temporary or permanent attribute, while the functional characteristics might be specifically derived from some basic requirements. An operating system (OS) is a comprehensive case to echo the above argument. The temporary components are numerous, namely, caches, buffers, processes, pipes and filters, etc., whereas the permanent ones are files, directories, bootstrap program, devices, etc. The functional characteristics can range from general utilities such as shell commands, editor, to special programs such as text formatting programs, network APIs, device drivers, etc. Fowler [1] outlines software architecture as follows

> "There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime."

The above arguments represent man-made complex artifacts that are theoretically well-structured, configurable, and can run indefinitely. Unfortunately, one of the shortfalls of this construct is its limitations induced by its own creating principles such as time and space complexities. This study looks into the basic building blocks of software architecture if there are alternatives to build software systems in a way mimicking natural artifacts, thereby the theory-ridden design process can be avoided. From the oriental belief, the human body is made up of four substances, namely, earth, water, air, and fire. What makes it work is the soul that controls this living contraption. The hardware and software systems are analogous to the body as depicted in Figure 1.

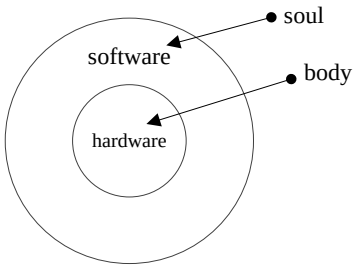

Figure 1. The human body VS the computer system.

The intricacy of architectural relationships to be drawn from this analogy begins at the highest level of body abstraction. The basic five senses, namely, sight, smell, touch, taste, and hearing, functioned by the input organs are the natural marvel that no artificially man-made devices can substitute. An even more intricacy is their integrated operations that consolidate all forms of input senses for instantaneous processing by the brain, subconscious mind, and instinct. The result is then interpreted, responded, or acted on accordingly. All these activities are completed in splits of a second. A further research deep into the structural construct of these organs unveils one common building block, i.e., the Deoxyribonucleic Acid (DNA) which is fundamentally composed of Adenine (A), Thymine (T), Cytosine (C), and Guanine (G) nucleotides. The functional aspect of each organ has long been discovered and established. From Figure 1, hardware and software exhibit *simple* design principles that map directly from 0 and 1 to AND, OR, NOT gates and software type hierarchy (see Figure 3). No qubit and superposition of quantum computing [21] or full library support of DNA computing [22] constructs to complicate the structure, function, and behavior of the proposed system. However, Adleman [23] proposed one good DNA computing prospect that could encode the Hamiltonian path to solve it in a single molecule.

The objective of study is to propose a Biological-like Architecture for Software Systems (BASS) that will attain the following four characterizations: (1) simple and straightforward linear structure and algorithm design for technology transfer to hardware, (2) fixed size to entail fast access and retrieval, (3) simple in situ replacement for space efficiency, and (4) no complex theoretical imposition.

This paper is organized as follows. Section 2 recounts some relevant researches that will be further exploited. Section 3 describes the methods for BASS architectural propositions in three aspects, namely, structure, function, and behavior. A concise verification of both propositions is also presented. Section 4 explains the simulation experiment. Section 5 explains some practical applications of this novel prospectus, limitations, and research questions. Some final thoughts are given in Section 6.

2. Related work

Race and ethnicity have long been used but were controversial and misunderstood of human classification. Mayr [2] was critical of the use of races. Foster et al. [5]

explained the multi-dimensional of genetic variations by genomic resources in human populations. Whatever the classifications might be, biological heterogeneity offers a range of human genetic compositions. Such constructs, complicated as they appear, are made up of the aforementioned DNA to carry the genetic code of living cells. This notion, though has not yet been adopted in software systems, can be drawn as an analogy to software architecture that has varieties of patterns composing the software systems.

One of the well-established architectural constructs is modularization. Parnas [4] discussed related criteria using Keyword in Context (KWIC) as an example that consisted of (1) the task which was to build a contextualized index for the text, (2) input which was a set of lines of text, and (3) Output which was the set of all circular shifts of all lines in alphabetical order. These in turn entailed several flexible software architectural designs such as function-based, action-based, pipe and filter, event-condition-action, and implicit invocation.

Meierk et al. [3] explained some key architectural designs, for instance, built to change instead of building to last models to analyze and reduce risks, used model and visualization, as a communication and collaboration tool, and identified key engineering decisions. Both suggestions pointed to flexible design considerations and decisions for a well-planned system architecture. Research endeavors on Software Engineering Environment (SEE) support [11] attempted to establish a framework of reference model (RM) for architectural standardization process, encompassing platform suppliers, environment suppliers, tool suppliers, and users. The underlying RM provided service groups as follows: (1) object management services, such as metadata, storage, persistence, archive, backup, relationship, name, distribution, location, replication, transaction, concurrency, synchronization, process support, access control, common schema, composite object, data interchange; (2) process management services, such as development, enactment, resource, monitoring, transaction; (3) communication services, such as data sharing, inter-process communication, network, message, event notification; (4) operating system services, such as synchronization, input and output, file storage, memory management; (5) user interface services, such as metadata, session, dialog, presentation, security, internationalization; (6) policy enforcement, such as identification and authentication, mandatory and discretionary access control, audit; and (7) framework administration services, such as registration, metrication, sub-environment, license management.

From hardware support standpoint, the contents of software are stored in memory waiting for execution. BASS is set out to simplify hardware constructs that support the software system operations, in particular, memory management systems. Saulsbury et al. [9] proposed the process and memory integration to solve the memory wall problem. Rixner et al. [10] introduced a memory access scheduler consisting of a comprehensive memory access scheduling architecture that supposedly alleviated memory bandwidth bottleneck. Current research and development efforts have established various techniques to efficiently handle such outgrown storage problems, leading to contention for shared resources on distributed environments such as non-uniform memory access (NUMA) [24]. However, the insatiable demands for massive storage do not dwindle down, cluttering the memory pool, and demanding more computational power of multiple tightly-coupled computer systems. Lergchinnaboot et al. [18] simulated the biological-like memory architecture using modified FIFO scheme and TTL as a limiting factor to gauge how the scheme stacked up against well-established memory allocation algorithms such as pure FIFO, shortest remaining time first (SRTF), and round-robin (RR). The prospect proved to out-pace all schemes except

pure FIFO due to context switch problem. Moreover, the modified FIFO scheme was starvation free and suitable for hardware implementable without any supporting complex allocation algorithms as oppose to other comparable schemes.

Recent development of DNA computing [14, 15, 23] shed some lights to the adoption of biological approach for solving classical computing problems such as NP problems. A number of intrinsically complex problems that could not be efficiently solved by conventional electronic computers were handled by DNA computing. Adleman [15, 23] exploited the sequential code of 0's and 1's data in the computer to be adapted on the DNA computing storage scheme. The revolutionary hindsight instills more research endeavors on biological adaptation of efficient and effective computing capability. All these prior works constitute the framework for the design of BASS. They are:

H1) component adheres to uni-cellular structure, that is, autonomy;
H2) component splits to create new ones (creation), undergoes some activities (sustainment), and finally goes away (cessation);
H3) component lasts only TTL duration just as uni-cellular life form has limited short life span. A renewal is allowed only in a long processing situation;
H4) basic types are the mandatory composition of component;
H5) component is linearly modeled after nature cellular structure, that is, nucleotide, codon, chromosome and cell; and
H6) nature is simple, so should BASS. No complex structures and algorithms are adopted.

In order to satisfy the above objectives of study, some questions on how to accomplish the four characterizations of BASS arise:

Q1) What are the design principles to map such simple and straightforward linear structure and algorithm to hardware?
Q2) How can space be efficiently handled in ways that improves memory access?
Q3) How does BASS benefit over conventional software architectures?

3. Methods for the proposed reference architecture

This study proposes a reference architecture of BASS that mimics simplicity of the human body. Two propositions are established based on the observation from Figure 1 as follows:

Proposition 1: There are simple building blocks on which software components can be built to constitute the software systems.
Proposition 2: System components must maintain a direct reference to the basic building blocks.

From the human analogy, the body is made of smaller organs which in turn are made of cells. The basic building blocks of these cells are nucleotides. A uni-cell life form demonstrates how a single natural building block can survive and grow. By the same token, the basic building blocks of BASS are components which make up the systems as shown in Figure 2. The design principle is to keep all components as simple as the uni-cell life form, i.e., by adhering to H1 and H5 that will permit fast access and disposal just like cells.

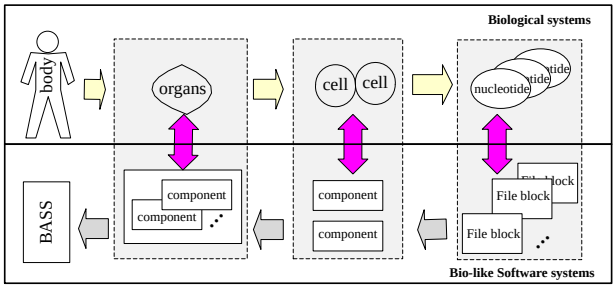

Figure 2. The proposed software systems architecture.

The propositions are elucidated descriptively to follow the above forerunners, namely, modular design [4], data and tools abstraction, software engineering environments and reference model integration [12], uniqueness of DNA string, and simple FIFO discipline. Details on how software components and their environments are laid out will be described in three aspects, namely, structure, function, and behavior.

3.1 Structural aspect of system architecture

As evident abound in archeology, many creatures were extinct while many evolved over the years. Conjectures on what the world centuries and millennia from now would look like have been attempted. One proposition that stood out from such predictions was the survival of cockroaches [7]. The fact that they are biologically simple renders them survive all adversaries as they could adapt to endure various environmental transformations. By the same token, the structural aspect of hardware and software can mimic such endurance by adhering to their simple building blocks, i.e., gates in hardware and bits in software. As such, they can maintain their close relationship to each other. In the meantime, the intermediate and high-level constructs fade in and out as computer system architecture evolves. For example, vacuum tubes, transistors, on the hardware side, and APL programming, HIPO chart, on the software side, became rarities and were replaced by integrated circuits and high-level languages or software design patterns. If one were to apply Proposition 1 to these progressive levels of abstraction, one could accumulatively create building blocks in the same manner as the biological DNA that makes up the cells, tissues, organs, and eventually the body. The creation process is subject to a set of basic premises that govern the configuration of the proposed architecture as follows:

1. Artifacts are countably finite and take the form of a component which represents the system construct of stored objects.
2. All sizes of component substructures are fixed to permit maximal provision for hardware level implementation.
3. Simple methods/algorithms are deployed to support the first two premises.

In order for the proposed architecture to support the above propositions, it is essential that design of the software systems be compatible with their environments. The basis of BASS component is characterized by its typing as shown in Figure 3.

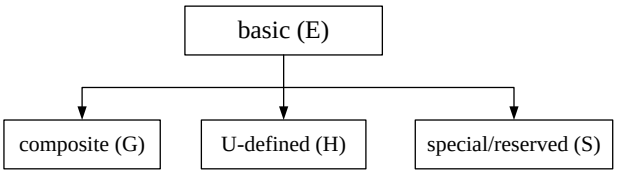

Figure 3. Component type hierarchy.

- basic types (**E**) such as int (A), binary (B), char (C), note (D) that denotes sound attributes such as intensity and frequency;
- composite type (**G**) which denotes combinations of the basic types;
- user defined type (**H**); and
- special/reserved type (**S**).

The difference between composite and user defined types is that the former builds a new component by combining several basic types, while the latter could be built from extending existing basic types or a new type defined by the user. For example, a composite type component could be composed of int (A), binary (B), and char (C). A U-defined component could be composed of extended binary type with a create-your-own method to manipulate the component or a new U-defined type for each 5-sense component representation.

The above set up only takes care of sight (text and image) and sound (audio) contents of file representations. The remaining three senses are more involved and complicated to represent. For example, touch includes texture, temperature, hardness; smell or scent includes odor, temperature; and taste includes temperature, flavor, texture, etc. These representations are left as future work.

Two rules are established for such transformation regulatory mandate: (1) all basic component types cannot be added or altered to prevent creation of new basic types, and (2) other component types can be freely disposed of, altered, redefined, or added. The first rule preserves the integrity of the entire "life form" to keep the basic construct simple and unchanged. Imagine if the DNA kept evolving and breeding new types of nucleotides other than ATCG, all human races on earth would flourish beyond anyone could imagine. The second rule supports cell growth or die. This allows components to be created as the system runs.

A BASS component is designed to be a fixed size linear array in accordance with the cell structure, i.e., from nucleotide to codon to chromosome and cell. Each hierarchy represents the characteristics of basic building blocks. The overview of a software component is depicted in Figure 4.

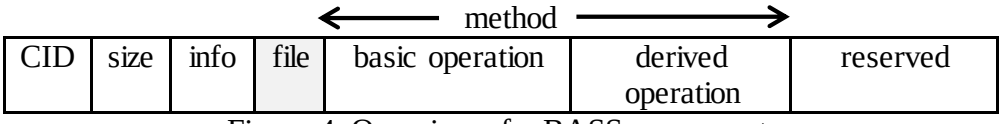

Figure 4. Overview of a BASS component.

- *componentID* (CID), to uniquely identify the component by incorporating timestamp generated by the global clock (which is analogous to the body clock) and information from the info field;
- *info*, to hold typical component information such as date of creation, last modification, owner, TTL which delimits the duration of component existence;
- *size*, to specify the (fixed) length of the method_set;
- *file*, to hold the contents of the component;
- *basic operation*, to hold operations that defines on the basic types;
- *derived operation*, to hold methods that are defined by the users.
- *reserved* for future extension.

The rationale for the above flat nested fixed size blocks is to simplify and increase data access and retrieval by hardware implementation. This construct also

permits replacement in situ of an expired component by a new one. Thus, memory wall and I/O-bound problems can be lessened.

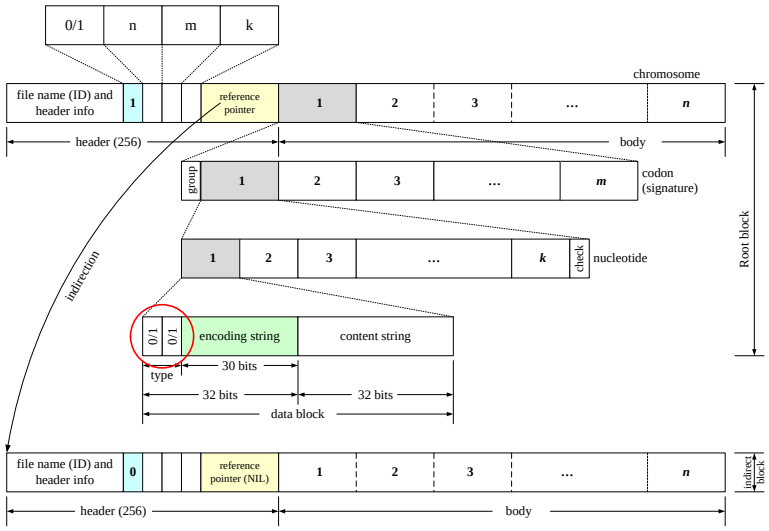

Figure 5. Structure of a file.

An overview of file structure is depicted in Figure 5. The first block is a directly accessible *root* block to permit hardware implementable access and retrieval operations. A file consists of two parts, namely, header and body. The header of size 256 bytes contains file name or ID and header information such as file location, owner, date of creation/modification. The next four values are set as follows:

- indirection bit (0/1), where 0 denotes direct, 1 denotes indirect reference of additional file blocks;
- multiplicity (n), whose values are either 1 (default value) or 2…n. These blocks analogously denote the chromosomes;
- system addressable size (m), whose value is divisible by *group*. This block size analogously denotes the codon. The value of group designates the number of nucleotides that constitutes units of file. The default value is 3 to conform to natural codon; and
- number of data blocks per process (k), whose value is power of 2, the default value is 4 blocks to mimic the natural A, T, C, G code. A *check* byte is appended at the end of each block as the sentinel value for its signature. These blocks analogously denote the nucleotides.

Note that the first two bits of every data block identify the component types, namely, 00, 01, 10, and 11 for type E, G, H, and S, respectively. The last value of header block is a reference pointer, which is set according to the indirection bit. If the indirection bit is 1, the reference pointer will hold the address of the next file block which is structurally identical to the root block. Otherwise, indirection bit is 0 and the reference pointer is set to NIL.

This layered multiplicity blocks stored in chromosome, codon, and nucleotide levels are made to hold component data that are working together. That means each component can execute several methods, while each method from different components interoperates with one another concurrently.

3.2 Functional aspect of system architecture

The most important functions of the proposed architecture are the global clock and type derivation. The global clock regulates all component activities and establishes the

limiting TTL to prevent their execution from FIFO infamous starvation problem. This is analogous to the heart that generates pluses to regulate all the body functions.

Table 1. Summary of type description.

Type	OpCode	Operation
	E	
int (A)	ADD, SUB	add, subtract
binary (B)	AND, OR, NOT, COM, SHT, XOR, BDD	bitwise and, or, not, complement, shift, exclusive-OR, binary add
char (C)	ORD	lexical order
note (D)	ITS, FRQ	intensity, frequency
	G	
A,B,C,D	-	composition of type **E**
	H	
U-defined	-	user define type
	S	
reserved	-	-

Table 1 summarizes each software component type and its associated operations to function as a unified entity. Thus, an integer component can operate with another integer component directly via ADD/SUB operations. Operations of composite and U-defined types can be constructed from basic types and combined to form complex components in the same manner as organs that are built on cells. These add-on constructs enhance the characteristics of Proposition 2 that will be established subsequently in Note 1 and 2.

3.3 Behavioral aspect of system architecture

An important behavioral dynamicity is software component life cycle. Conventional systems will perform a "context switch" to preserve all process states and contents before swapping this completing/expiring process out. BASS performs component reproduction (creation) of new components to handle whatever functions are required to continue processing (sustainment) while its TTL diminishes. The expiring component is then disposed of (cessation). The new components perform *replacement in situ* to the expired ones. No context switch, storage reclamation, and house-keeping tasks are required. This is similar to old skin cells are replaced by the new cells as shown in Figure 6. In an event when the component is engaging in some lengthy processing, the expiring TTL can be renewed by setting a new TTL limit so that the component is operating afresh and autonomously. Consequently, minimal overhead is involved in the system dynamicity.

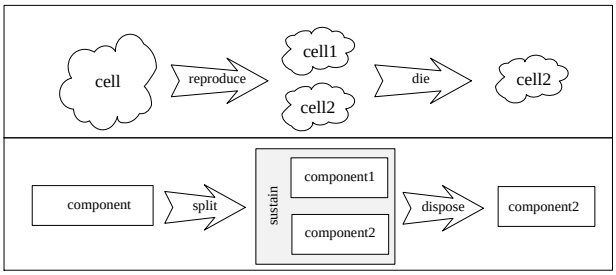

Figure 6. Cell (component) create (split), sustain, and cease (dispose).

Figure 7 illustrates the reproduction process of a new component, where the original component splits to form two components, namely, ORIG and SPLIT_1,

having CID and CID' as their ComponentID, respectively. The new CID' is derived from the original CID plus timestamp information to keep it unique.

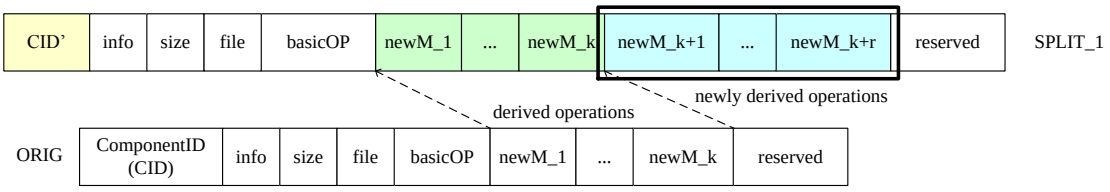

Figure 7. Component reproduction to become two components.

Note that SPLIT_1 can be enhanced with additional derived operations to accommodate new capabilities. The provision of *n, m, k* can readily support system expansion with less reconfiguration or modification effort.

One arguable issue when taking structural and functional considerations into account is the way components grow in the same manner as cells do. As DNA induces run-on strings of A, T, C, and G, identification or matching is tedious and time-consuming. Consequently, the DNA strands can be excessively long which somehow offset their uniqueness of identification advantage. On the contrary, identification of component under BASS scheme is confined to the structural architecture: fixed size, finite number of components, and well-defined methods. Thus, processing is fast and enumerable to terminate.

The second proposition conveys one very important architectural design principle, that is, component autonomy. It implies that each software component is self-contained. Yet its behavioral capability depends on how it is designated to function. For example, graphic components will be equipped with basic type operations, such as A, B, and C to support the sight sensory. On the other hand, communication components will contain protocol related operations encompassing E, G, H, and S to support different operations. Since each software component must operate to practically "survive" on its own, loose integration (coupling) is the basis for the design principle. They essentially have virtually no relation except a direct reference that ties them to the software system building blocks. Such independent constructs make homo-function software component interchangeable. By the same token, related or hetero-function software components interoperate to exchange or share results required by the sub-system to which they belong. For example, the print method that accepts text inputs, namely, int (A) and char (C) and prints them out on the paper must invoke text component to open and retrieve the data content. This can be summarized in two enhanced constructional and operational characteristics of software components.

Note 1: All components are interchangeable and interoperable.

Note 2: Software engineering components can be composed from the basic building blocks in such a way that they comply with the underlying architecture.

Proofs of both notes will be postponed until Section 3.5. At any rate, the distinctions for architectural design of BASS from object-oriented (OO) paradigm are summarized in Table 2.

Table 2. Summary of BASS vs Objected-Oriented architectural differences.

BASS (X)	O-O paradigm (Y)	Remark (denoted by X and Y for brevity)
creation, sustainment, cessation	Abstraction	main aspect of X, Y depends on disciplinary design by programmer
type	attribute	X is confined to fixed hierarchy, but Y must obey the language construct
component	class	X is predefined, but Y depends on programmer
component	sub-class	X reproduces component, Y uses language support
group of components	nested classes	X combines components, Y combines classes
file	object	instance and expire (X), instance but explicit deletion (Y)
simplicity	complexity	goal of X, Y depends on programmer
component	inheritance, polymorphism	X is autonomous, Y uses language support
component	encapsulation, modularity	X is mandatory, Y depends on programmer
component	information hiding	X is mandatory, Y is done by programmer
reproduction	instantiation	X splits, Y creates object from class
-	association, overloading	X is autonomous, Y uses language construct
-	concurrency	X is autonomous, Y uses language construct
-	IDE	X is self-contained, Y requires its support
TTL	persistence	X is self-limiting, Y uses language construct

The main distinction of BASS from OO paradigm is the three-stage cycle that sets BASS artifacts to be autonomous. This design feature lessens the overhead burden of system support, while existing design paradigms encompass heavy overhead such as compiler, libraries, application program interfaces (APIs), and integrated development environment (IDE).

3.4 Verification of Proposition 1

A corroboration of formal verification can be demonstrated as follows. Let $e \in E$, $g \in G$, $h \in H$, and $s \in S$ denote basic, composite, U-defined, and special/reserved component types and their corresponding sets, respectively. The relation $\rightarrow$ denotes *composed of*, i.e., $n \rightarrow e|g|h|s|\{g, h, s\}$ designates a component n to be composed of either pure e, g, h, s, or mixture of g, h, s, where {..} denotes mixture of types. By the component type hierarchy in Figure 3, $\forall g, h, s$, $g = \bigcup_{i=1}^{x} e_i$, $h = \bigcup_{i=1}^{y} e_i$, and $s = \bigcup_{i=1}^{z} e_i$, where x, y, z are arbitrary typing counts such that for any component n, the architectural construct must satisfy the first condition (i)* plus one of the following conditions:

(i)*	$n \cap e \neq \varnothing$			/* basic types */
(ii)	$n \cap g = \varnothing$	or	$n \cap g \neq \varnothing$	/* composite */
(iii)	$n \cap h = \varnothing$	or	$n \cap h \neq \varnothing$	/* U-defined */
(iv)	$n \cap s = \varnothing$	or	$n \cap s \neq \varnothing$	/* special/reserved */
(I)	$n \cap (g \cap h) = \varnothing$	or	$n \cap (g \cap h) \neq \varnothing$	/* composite/U-def */
(II)	$n \cap (g \cap s) = \varnothing$	or	$n \cap (g \cap s) \neq \varnothing$	/* composite/reserved */
(III)	$n \cap (h \cap s) = \varnothing$	or	$n \cap (h \cap s) \neq \varnothing$	/* U-def/reserved */
(IV)	$n \cap (g \cap h \cap s) = \varnothing$	or	$n \cap (g \cap h \cap s) \neq \varnothing$	/* compo/U-def/reserved */

The first condition (i)* is mandatory for all components constructed since they must comply with the proposed scheme. This is in concert with the A, T, C, G nucleotides that make up the DNA. One of the remaining conditions could hold if the component so constructed is made up of additional types. That is to say, conditions (ii) to (iv) represent pure composite, U-defined, or special/reserved type, while conditions (I) to (IV) represent composite/U-defined type, composite/reserved type, U-defined/reserved type, and composite/U-defined/reserved type, respectively. For example, define a composite component type (n_1) → (A, C) and a user define component type (n_2) → (extended-int, B, C). In this case, the counts of n_1 and n_2 become $x = 2$ and $y = 3$, respectively, and $n_1 \cap n_2 \neq \varnothing$ which is equal to char (C). Hence, the component having $n_1 \cap n_2$ as its typing basis satisfies conditions (i)* and (I). On the other hand, if n_3 → (extended-int, B, D), then $n_1 \cap n_3 = \varnothing$. This means that the component having $n_1 \cap n_3$ as its typing basis also satisfies conditions (i)* and (I).

Now suppose a new component type (m) is created by combining int (A), binary (B), and (n_1 and n_2), or m → (A, B, n_1, n_2) as a special type having the count $z = 4$, there are two possible formulation conditions to be deployed:

- conditions (i)* and (I), i.e., consider A and B separately satisfying (i)* and the n_1 and n_2 satisfying (I), that is, $m \cap A \cap B \cap (n_1 \cap n_2) \neq \varnothing$, or
- conditions (ii) and (I), i.e., consider A and B as a composite type ($A \cup B$) satisfying (ii) which implicitly satisfies (i)*, and the entire group satisfies (IV), that is, $m \cap (A \cup B) \cap (n_1 \cap n_2) \neq \varnothing$.

It can be inferred from the above formulation that the simple building blocks make up larger components under the proposed scheme.

3.5 Verification of Proposition 2 and its Notes

Let l denote the height of the component type hierarchy measured from basic type which serves as the root level. Define $l = 0$ for the basic types. Since all component types are derived directly from the basic types which is one level away, the height becomes $\max(l) = 1$.

As for Note 1 goes, given p is a newly created component. There are two scenarios to consider. Firstly, $p \in$ G|H|S, i.e., p is either a composite, U-defined, or special/reserved type. According to Proposition 1, the condition (i)* must hold for all components, i.e., the portion of p that is made up by basic types is known by all other components. The remaining portion that is not pure basic types must either be pure composite, U-defined, special/reserved type, or mixture of composite/U-defined, reserved/composite, reserved/U-defined, reserved/composite/U-defined. In either case, the conditions (ii) to (iv) or (I) to (IV) apply. Thus, data exchange and interaction among them are straightforward. Secondly, $p \notin$ G|H|S. Only the basic portion applies to p. This implies that the remaining portion of p does not reuse any existing types and

methods and is foreign to other components. In order for *p* to exchange or interact with others, *p* must deposit its type and corresponding method to the shared component repository. As a consequent, Note 1 holds.

Proof of Note 2 also follows in a similar manner. From Proposition 1, all software components must be created having at least condition (i)*, and so is the underlying architecture. Otherwise, Proposition 1 will not hold and the system cannot operate. Thus, the smallest composition of software artifact complies with the proofs and is able to operate properly. As these software components grow, the newly added constituents of type {G, H, S}, as well as their corresponding methods, must be declared and deposited in the shared component repository. At which point, each added artifact can be accessed and interoperated with other existing components. Consequently, Note 2 holds.

4. Simulation

A simulation module was devised to exercise the performance of BASS since it is a novel prospectus having no forerunner to compare. Thus, the simulation would focus on memory utilization since it is one of the most important operations for any support environments.

The simulation began by manually creating the first component to initiate the process. This component contained necessary contents, namely, CID, info, size, file, and DEFAULT number of core operations but no derived operation (r=0). Then the reproduction process began. The first off-spring had no derived operation because the first reproduction (cell split) yielded an identical copy to the first component. The immediate step randomized the value of r for the new generation off-spring to mimic cell evolution process, wherein derived operations came into play for subsequent simulation runs, that is, created CID', process component, and randomized r. The simulation procedure was laid out as follows:

```
Procedure      sim_test
      create FIRST component                    /* handcraft the FIRST component */
      set LIMIT                                 /* set no. of simulation runs */
      X ← FIRST, r ← 0
      reproduce:                                /* recursively create until LIMIT */
      repeat def_op (newM_w)                    /* w = basic operations */
         create CID' from CID + timestamp
         init_defaults                          /* all defaults initialization */
         process_component (X)                  /* r = 0, no derived operation */
         assign_component (X)                   /* deposit in FIFO memory */
         run_service()
         TTL_check()                            /* expiration */
      until w > DEFAULT
end Procedure
```

Table 3 summarizes the estimation of parameter setup. Some parameters deployed in the simulation were experimentally tried and predicted using exponential decay (Eq 1) and the 37% Method [20] to simulate cell reproduction process, that is,

$$N(t) = N_0 e^{(-t/\tau)} \quad (1)$$

where $\tau = 1/\lambda$, λ denotes the creation rate.

Table 3. Simulation parameter setup.

Parameter	Setup value	Remark
mean life time (τ)	500	475 rounded up
initial component (N_0)	1	first component
simulation run (t)	2000	round up
processing repetition (N)	10000	arbitrary selected

The proposed design was written in C running on Intel® Core i5, CPU M 520 @2.1 GHz x 4, Ubuntu 16.04 LTS system to simulate BASS memory allocation scheme. The execution performed basic arithmetic and modulo computations of 10,000 repetitions. Components were created, sustained (executed), and disposed of (written to disk) very rapidly. For the first hundred repetitions, memory allocation calls in C were performed repeatedly in user mode. This inevitably induced high overhead to set up the component and file structures which sporadically took several clock ticks during each simulation run, i.e., 3, 5, 10, 12, 13, 17, 18, 20. An extended looping of 500,000 repetitions was then conducted in each run to observe whether any steady states would result. It could be seen that more repetitions induced the OS to prolong the process in core so that execution would be completed faster. In other words, memory blocks were replaced in situ and fewer swaps of the running process out, hence faster process completion. The total execution time of subsequent calls was negligible. This is shown in Figure 8, where the tail flattens down considerably.

Time measurements were done by system clock ticks via *struct tms*. The variables duration denotes the time of simulation run (sim_test), DFutime denotes component cycle time (step 1-7), and DFstime denotes the time spent on instructions of the user calling process (memory allocation calls in C).

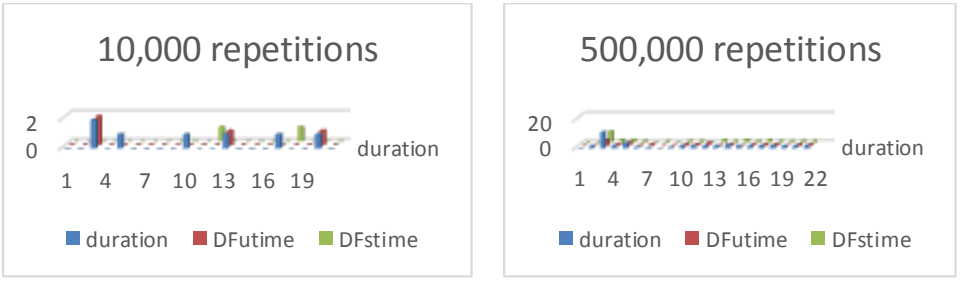

Figure 8. Memory allocation looping of 10k and 500k after 21 runs.

5. Discussions

The proposed BASS architecture establishes an unconventional basis for building software systems. The design framework based on H1-H6 made it easy for software resource utilization in an integrated development environment.

5.1 Practical applications

Conventional software systems from the highest level of abstraction such as architecture, model, software system design, computing representations, and program organization, are all transpired to 0 and 1 to run on hardware. BASS is no different but takes the biological-like mapping of bodily architecture and cell model fundaments to synthesize the components into three stages, construction (*creation*), operation (*sustainment*), and disposal (*cessation*) processes. A suitable snapshot of testing these novelty mixes is to measure the concrete existence of components, i.e., memory allocation, deallocation, and process execution of components. Figure 9 shows the transformation from conceptual model to BASS model whose image of execution

resides and transforms in memory. The reasons are two folds. First, it is a concrete evident to objectively measure BASS performance in software systems; and second, the framework can be exercised in real use to mitigate the infamous "*memory wall*" problem. Under H1 and H6, the memory test scheme will not make use of NUMA to complicate the access and retrieval processing since there is no shared resource to begin with.

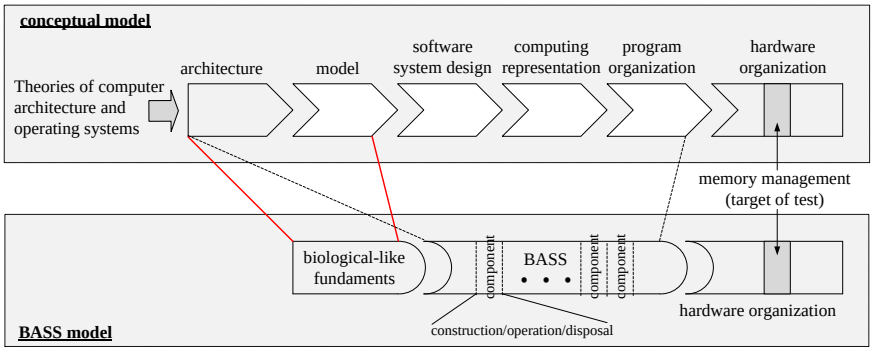

Figure 9. A mix of architectural constructs to hardware realization of BASS.

As part of the on-going investigations, the simulation results [25] have shown that TTL-FIFO and replacement in situ schemes saved 21% cost over the Least Recently Used (LRU) built-in algorithm of the memory chipset. Consequently, the novelty of BASS could, at present, be empirically gauged by how well it manages the memory in comparison with the state-of-the-practice memory management methods.

Another area of application that increasingly gains research interest is computing structure. Many data gathering techniques are developed to handle massive data storage and computations such as data warehouse and big data. Rack scale architecture [12], co-locating data storage and processing [6], computing virtualization [13], scalable synchronization on shared memory multiprocessor [16], and transient servers [17], are a few prominent examples. Unfortunately, these classical paradigms rest on stored program architecture that requires increasing memory, computing power, and complex algorithms that eventually create the memory wall. BASS, on the contrary, offers a pool of short-lived resources to handle data processing in the similar manner as the human analogy. Figure 10 depicts the conceptual framework of BASS data pipeline that echoes the above multimedia processing argument.

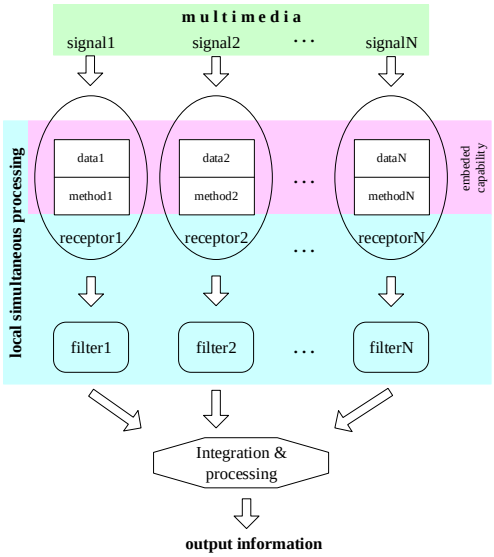

Figure 10. Data pipeline: signal reception by specific input receptors.

For example, receptor1 could denote the eyes to receive image (signal1), while receptor2 could denote the ears to receive sound (signal2), and so on. This in turn is filtered by the respective method (filter1/filter2) to extract useful information from the brain archive for proper integration and processing.

The important issues for the filtering process are two folds, *coverage* and *tracking*. Coverage ensures that the system could provide proper receptors to accommodate different input signals. There will be no mismatch as to receptor1 takes signal2 and receptor2 takes signal1 since their corresponding filter1 and filter2 are unable to cover incorrect input data type.

Tracking, on the other hand, must ensure that the input data are recognizable or failed otherwise. For instance, a Latin voice signal might be handled by the coverage of receptor2. However, tracking could not retrieve related information from the brain (as the person did not learn Latin) to recognize it and thus failed to understand what the input voice meant.

The last example reiterates the compatibility with both propositions, namely, (1) no typing mismatch since the underlying architecture separates receptors for different input types and operations process them accordingly, (2) retrieval without conversion since each receptor is built to handle a specific input type, (3) simple building blocks and direct reference, and (4) simultaneous integration and processing as all inputs are processed to yield an immediate outcome.

5.2 Limitations

To operate this novel proposed architecture by way of Figure 2, memory allocation for BASS components on the underlying hardware may be set up in accordance with bodily priority. That is, dedication of *system space* for urgent functions as oppose to *user space* for mundane functions. Consider the scenario shown in Figure 11.

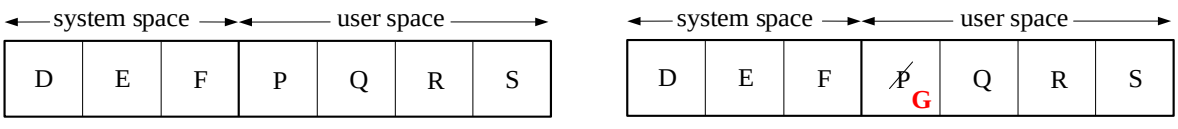

Figure 11. memory allocation for system and user components.

Suppose the memory blocks are allocated for system and user spaces in fixed-size blocks D, E, F and P, Q, R, S, respectively. All component method execution is regulated by the global clock in a First-In-First-Out (FIFO) queueing discipline. Suppose block D is assigned to the heart process, E to kidneys process, and F to lungs and respiratory process, while P, Q, R, and S are assigned to jogging, hand-swinging, listening to music (from wearable device), and singing along processes, respectively. Suddenly, the body gets wind-headed which is another system process (called G) that calls for an immediate service. Process G will be put to execute in memory by overriding user process P in the FIFO fashion since G cannot override D, E, and F processes. Thus, the remaining mutual dependent Q, R, and S processes to P also get terminated. This is similar to real body handles the situation as it can no longer sustain the sudden severe condition and must stop the jogging (P), hand-swinging (Q), listening to music (R), and singing along (S) activities.

The above practical applications and limitations open the horizon of SEE to encompass the four characterizations that help answer the above questions as follows:

Q1: The above simulation and the simulation results obtained by Lergchinnaboot et al. [25] reaffirmed the technology transfer at memory level. The importance of H6 excludes the use of logical level complexity such as NUMA, DNA and quantum computing. These architectural constructs could not be mapped directly to simple gate level implementation. On the contrary, the fixed size linear configuration of BASS components will structurally map to hardware implementation, while the simple TTL-FIFO memory access will functionally run in the memory chipset with replacement in situ scheme. As a consequence, the four characterizations of this study are satisfied.

Q2: By virtue of component and file designs, BASS expands its capacity when execution context grows by creating a new component to handle and shrinks when execution completes by ceasing the expiring component that is regulated by a TTL limit. Thus, memory occupation by system environment is efficiently arranged with little software overhead since every BASS component run autonomously.

Q3: The benefits advocate new system construction by several folds:

1. Fast access and retrieval to component configuration since its structure is fixed size and arranged linearly. This setup makes it ideal for table lookup or base-register and offsetting implementation at hardware level.
2. Components are governed by predetermined TTL. Thus, all components come and go. There is virtually no burden left over for garbage collection.
3. Components are self-contained and operationally autonomous. It is thus portable to different support environments.
4. The needs for large, preset, and long-term availability of resources such as storage space or processing units are unnecessary. By virtue of the first and second benefits above, these resource requirements can be established on demand and released once execution is completed. Thus, energy conservation can be substantial.
5. Lessen the burden of release engineering as new components can be tailored to fit a specific locale or "ecosystem" as Rossi put it [8] without having to maintain continuous delivery across the board. This flexibility is similar to organ transplant that is performed on a case-by-case basis as deemed necessary.

6. Conclusion

The biological-like architecture for software systems (BASS) were proposed to revolutionize how software systems would operate. By imitating cell life cycle, the proposed architecture creates components that live, work, reproduce, and die as natural as they can be. The prospectus was justified by three architectural constructs, namely, structural, functional, and behavioral. Two propositions were precipitated to govern the architecture of BASS. The most fruitful result of BASS was an autonomous entity that was self-reliance and independent among themselves.

Therefore, BASS and its components could grow and die in the same way as the body with little overhead. As a man was born and passed away, so was BASS components created and deleted. This very notion would maintain the dynamicity of computer applications in the same manner as the human society, but less congested to make room for new application generations to live and run for the existence of the computer systems.

An important consideration on future work is the design quality of BASS components, in particular, Measure of Aggregation (MoA) [19]. This metric will be deployed to measure how composite and U-def types will permit flexibility and effectiveness of attributes to support component autonomy.

Finally, one of the important problems is autonomous method execution. Since researchers still cannot uncover the secrets of DNA that would shed lights on these issues, how different methods of a designated artifact can run in different environments where it resides still remains to be reckoned with. The hardware solution at cache level where the basic types *live* is under investigation. This viable solution will hopefully bring about BASS to a new level of ubiquitous architecture for any operating environments.

Declarations

1. List of abbreviations is provided below.

Abbreviation	Full text
BASS	Biological-like Architecture for Software Systems
TTL	Time-To-Live
FIFO	First-In First-Out
DNA	Deoxyribonucleic Acid
SEE	Software Engineering Environment
E	Basic type
G	Composite type
H	User defined type
S	Special/reserved type
NUMA	non-uniform memory access

2. Availability of data and materials
   Not applicable.

3. Competing interests
   Not applicable.

4. Funding
   Not applicable.

5. Authors' contributions
   Not applicable.

6. Acknowledgements
Not applicable.

7. Authors' information
Peraphon Sophatsathit received the B.E. degree in industrial engineering from Chulalongkorn University. Bangkok, Thailand, M.S. degrees in industrial engineering and computer science from University of Texas at Arlington, USA, and Ph.D. degree in computer science from Arizona State University, USA. He worked at the National Electronics and Computer Technology Center (NECTEC), Bangkok, Thailand. At present, he is working at Advanced Virtual and Intelligent Computing (AVIC) Center, Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University. His major research interests are software engineering, operating systems, and distributed & parallel computing.

6. References

[1] Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.

[2] Ernst Mayr, The Growth of Biological Thought—Diversity, Evolution, and Inheritance. The Belknap Press of Harvard University Press, 1982.

[3] J.D. Meierk, David Hill, Alex Homer, Jason Taylor, Prashant Bansode, Lonnie Wall, Rob Boucher Jr., and Akshay Bogawat, Microsoft Application Architecture Guide, Patterns & Practices, 2nd Edition, 2009.

[4] D. Parnas, On the Criteria to be used in Decomposing Systems into Modules, Communications of the ACM, December 1972, vol. 15, no 12, pp. 1053-1058.

[5] Morris Foster and Richard Sharp, Race, Ethnicity, and Genomics: Social Classifications as Proxies of Biological Heterogeneity, Cold Spring Harbor Laboratory Press, 12: 2002, pp. 844-850.

[6] Aisling O'Driscoll, Jurate Daugelaite, and Roy Sleator, 'Big data' Hadoop and cloud computing in genomics, Journal of Biomedical Informatics, 46 (2013), pp. 774–781.

[7] Cockroaches Survive Nuclear Explosion, mythbusters database, http://dsc.discovery.com/tv-shows/mythbusters/mythbusters-database/cockroaches-survive-nuclear-explosion.htm, accessed on October 29, 2018.

[8] Bram Adams, Stephany Bellomo, Christian Bird, Tamara Marsheall-Keim, Foutse Khomh, and Kim Moir, The Practice and Future of Release Engineering, A Roundtable with Three Release Engineers, IEEE Software, March/April 2015, pp. 42-49.

[9] Ashley Saulsbury, Fong Pong, Andreas Nowatzyk, Missing the Memory Wall: The Case for Processor/Memory Integration, Proceedings of the 23rd annual International Symposium on Computer Architecture (1996), Philadelphia, USA, May 22 - 24, 1996, pp. 90-101.

[10] Scott Rixner, William Dally, Ujval Kapasi, Peter Mattson, and John Owens, Memory Access Scheduling, Proceedings of the 27th Annual International Symposium on Computer Architecture (2000), Vancouver, Canada, pp. 128-138.

[11] Reference Model for Framework of Software Engineering Environments, Edition 3 of Technical Report ECMA TR/55, NIST Special Publication 500-211, August, 1993.

[12] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash, R2C2: A Network Stack for Rack-scale Computers, SIGCOMM '15, August 17 - 21, 2015, London, pp. 551-564.
[13] Magnus Heitzler, Jürgen Hackl, Bryan Adey, Ionut Iosifescu-Enescu, Juan Carlos Lam, and Lorenz Hurni, A method to visualize the evolution of multiple interacting spatial systems, Journal of Photogrammetry and Remote Sensing, 117 (2016), pp. 217–226.
[14] Junzo Watada, Rohani binti abu Bakar, DNA Computing and Its Applications, Eighth International Conference on Intelligent Systems Design and Applications, 2008, pp. 288-294.
[15] Leonard Adleman, Molecular Computation of Solutions to Combinatorial Problems, Science, New Series, vol. 266, no. 5187, November 11, 1994, pp. 1021-1024.
[16] John Mellor_Crummey and Michael Scotty, Algorithms for Scalable Synchronization on Shared Memory Multiprocessors, ACM Transactions on Computer Systems, Vol. 9, No. 1, February 1991, pp. 21-65.
[17] R. Singh, P. Sharma, D. Irwin, P. Shenoy, and K. Ramakrishnan, Here Today, Gone Tomorrow: Exploiting Transient Servers in Datacenters, IEEE Internet Computing, July/August 2014, pp. 22-29.
[18] G. Lergchinnaboot and P. Sophatsathit, A Biological-like Memory Allocation Scheme using Simulation, Proceedings of the 2nd International Conferences on Information Technology on Information Technology, Information Systems and Electrical Engineering (2017), November 1-3, 2017, Yogyakarta, Indonesia, pp. 425-428.
[19] Jagdish Bansiya and Carl Davis, A Hierarchical Model for Object-Oriented Design Quality Assessment, IEEE Transactions on Software Engineering, Vol. 28, No. 1, January 2002, pp. 4-17.
[20] Leo Breiman, Randomizing Outputs to Increase Prediction Accuracy in Machine Learning, vol. 40, 2000, pp. 229–242.
[21] Neil Gershenfeld and Isaac Chuang, Quantum Computing with Molecules, Scientific American, June 1998, pp. 66-71.
[22] Ravinderjit Braich, Nickolas Chelyapov, Cliff Johnson, Paul Rothemund, and Leonard Adleman, Solution of a 20-Variable 3-SAT Problem on a DNA Computer, SCIENCE, vol 296, 19 Aril 2002, pp. 499-502.
[23] Leonard Adleman, Computing with DNA, Scientific American, August 1998, pp. 54-61.
[24] Sergey Blagodurov, Alexandra Fedorova, Sergey Zhuravlev, and Ali Kamali, A case for NUMA-aware contention management on multicore systems, Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, Sept 11-15, 2010, Vienna, Austria.
[25] G. Lergchinnaboot, P. Sophatsathit, and S. Maneeroj, In Situ Caching using Combined TTL-FIFO Algorithm, 2019 IEEE 5th International Conference on Computer and Communications, Chengdu, China, 2019, pp. 437-441.