

## Towards Autonomously Developed Software: A genetic Approach in Critical and Embedded Systems

<sup>1</sup>Djamel Meslati and <sup>2</sup>Saïd Ghoul

<sup>1</sup>LRI Laboratory, University of Annaba, BP 12, Annaba, Algeria

<sup>2</sup>Philadelphia University, Computer Science Department, Sweilah, PoBox 1101, Amman, Jordan

---

**Abstract:** Nowadays, we are still considering changes undergone by a software system as a sporadic phenomenon and we do not sufficiently anticipate future changes during the development phase. Consequently, many problems arise in the maintenance phase. In this study, we present an approach where all changes undergone by a software system are considered as its ontogenetic dimension. We represent this dimension by specific concepts as a continuous and well delimited process that is embedded in the software model of a system. Inspired by genetics, our approach proposes a model where anticipated and unanticipated changes are modeled by a collection of fine grained instructions called genes.

**Keywords:** Anticipated changes, evolution, gene, genome, unanticipated changes

---

### INTRODUCTION

Genetics has two aspects, ontogenesis and phylogenesis, that govern two processes in biological organisms. The ontogenesis governs all the developmental changes that shape an organism throughout its life by interpreting its genetic code. The phylogenesis governs evolution, it has no effect on the organism itself, but on maintenance and enhancement of species<sup>[1-3]</sup>. This aspect has been used in evolutionary algorithms, where each iteration consists of handling the genetic codes of individuals using crossover and mutation operators and then selecting the best individuals according to some adaptation function<sup>[4,5]</sup>. Evolutionary algorithms are widely and successfully used in many domains. Unfortunately, neither phylogenesis nor ontogenesis is really used in software engineering.

In this study we are concerned by the ontogenetic aspect of genetics. We exploit a genetic metaphor by considering that a critical and embedded software system has, in addition to structural and behavioral dimensions, a third dimension called ontogenesis that governs all the changes undergone by the two first ones. In our approach, the model of given software system consists of a phenotype and a genome. While the phenotype captures the structural and behavioral dimensions, the genome captures all changes that shape the system to keep it conform to the changing environment and requirements. The genome is composed of fine grained instructions called genes. Each gene achieves an elementary change on the model.

Figure 1 shows our vision of ontogenesis and phylogenesis. We can see that what we call the

modeling phase consists of creating the first genome of the software system (G0). The interpretation of G0 will produce the first phenotype (P1). The end of an important phase, called embryogenesis, is reached when the phenotype becomes able to run and interact with the real world. During the life cycle of the software system, the genome remains active and continuously shapes the phenotype. Unanticipated changes consist of deleting genes from, adding genes to, the genome. Notice that phylogenesis is achieved through a partial reuse of the genome.

To avoid ambiguities, we separate clearly the software development from the evolution. Biologists consider that any change undergone by an organism belongs to its developmental process, which is called ontogenesis and use the evolution or phylogenesis terms when studying species and comparing individuals with their ancestors<sup>[1]</sup>. According to this, it's inappropriate to use the evolution term as a synonym of changes undergone by a software system during its life cycle.

Our approach is based on four principles:

- \* Any change undergone by a software system belongs to its ontogenetic process.
- \* Ontogenesis is a continuous process. In current approaches, changes occur during the development and maintenance phases and are considered as a sporadic phenomenon<sup>[6]</sup>. In our approach, the genome is the kernel of the model that continuously achieves changes according to internal or external triggering conditions and events.
- \* Ontogenesis is an embedded dimension. i.e. the

software system develops autonomously. Anticipated changes are coded in the genome as genes and executed when triggering conditions and events are met. Anticipating and coding the future changes is what we call the modeling of ontogenesis.

- \* Unanticipated changes are coded as genes and added to the genome when needed.

The remaining part of this study is composed of five sections. Section 2 states clearly the motivations of our approach. Section 3 gives target applications where our approach can be used. Section 4 describes the main concepts and their use through an example. In section 5 we discuss related work, and in 6, we give a conclusion and some perspectives.

## MOTIVATIONS

**Modeling the ontogenetic process:** This implies two aspects. First is the integration of the process in the software system itself and second is the modeling of the ontogenesis as a continuous process.

Today, the change process is seen as a human intervention on the software system that keeps it conform to the domain it represents and supporting evolving requirements of the users. The software system is considered as a collection of facts or passive entities. However, when modeling ontogenesis, we model the change process itself. This means investigating the future of a given domain to determine what it will be, when and under what conditions it will change, then adding some knowledge and mechanisms to the software system allowing it to change autonomously.

To illustrate our proposal, let's consider the biological example of Fig. 2, where an egg transforms into a butterfly. A static view of this reality, leads us to model it by static structures whose types and values change by external intervention. A dynamic view assigns to some behaviors the task of changing the structure values. However, not all the dynamicity is captured since the change of structures and behaviors themselves remains necessary.

The ontogenesis is a continuous process. If we consider the previous example: An egg undergoes a continuous change until it becomes a butterfly, even if we perceive only distinct phases. In a static view the changes of the attribute value are accumulated (for example a length of 6,5 cm), then the model is updated by an external intervention (for example the value 6,5 replaces 5). When adding the behavior `ChangeLength()`, the value of the attribute can be changed more frequently (5, 5.2, 5.3, ..., 7) according to a certain natural rule. In a similar way the maintenance accumulates several changes before

modifying a model. In the ontogenetic approach we use behaviors of higher order to continuously change the structural and behavioral properties.

Reducing the interactions with the real world. Interactions that involve human being are generally error prone. Therefore, their reduction is worthwhile. We distinguish two kinds of interactions. First are interactions that aim to change the model. They are effectively reduced in our approach as changes are anticipated and encoded in the genome. However, the genome extension constitutes unavoidable interactions corresponding to unanticipated changes.

Second are interactions which derive from the use of the model's functionalities: inputs, outputs and perception of external stimuli. Although, those interactions are design depend, meaningless interactions can be avoided when the model always conforms to the requirements. For example, when an object field is no longer needed, its deletion avoids unneeded input of its value.

**Reducing the maintenance effort:** Although an autonomously updated system needs less external interactions to change, this is not what reduces the maintenance effort. In deed we must take into account that, initially, we need an important effort to describe the ontogenesis.

The main reason behind this reduction is rather due to the fact that when we anticipate changes, using a systematic approach, we can avoid any inappropriate ones and the corresponding feed backs. For example, when an investigation leads to the conclusion that in some phase of an object life, a field type will become a real, we can avoid intermediary changes that this type subsumes such as integer.

**Enhancing performances:** It is a consequence of the specialization and the dynamic adaptation of programs<sup>[7,8]</sup>. To understand this, we need to compare a system, modeled using our approach, with what we call a stable system, i.e. a utopist system that deals with anticipated changes without any maintenance and that can run on several platforms. For example, one can imagine a class that regroups all properties along with conditional instructions that avoid erroneous use (for example the caterpillar cannot fly). Dealing with all incompatibilities between properties, makes the stable system less efficient, complex and entangled. We advocate an approach where a software system is dynamically updated and a general exception mechanism (similar to the Java one) is used to deal with objects in different stages of their life. In the same way, it is possible to dynamically adapt a system to its execution platform.

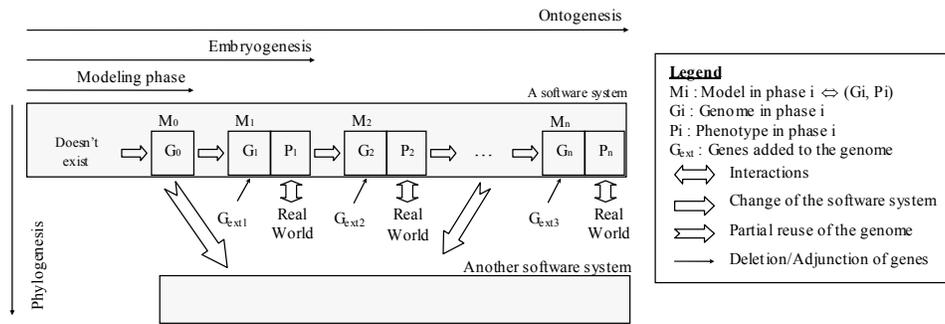


Fig. 1: Ontogenesis and phylogenesis in the proposed approach

Phases	 Egg	 Caterpillar	 Chrysalis	 Winged adult
Static modeling	Relief : Streaky Color : Green Length : 0,3 mm	True legs : 6 Pro-legs : 10 Length : 5 cm SkinShedNumber : 5	Volume : 8 cm <sup>3</sup> Length : 6,5 cm	Legs : 6 ; Wings : 4 Wingspread : 10 cm Length : 7 cm Wings beating : 35 per second
Dynamic modeling	idem static + Method Hatch()	idem static + ProduceSilk(), Breathe(), ShedSkin(), ChangeLength()	idem static + Breathe()	idem static + Fly(), SuckFloralNectar(), ChangeLength()

Fig. 2: Modeling of a butterfly at different phases of its life

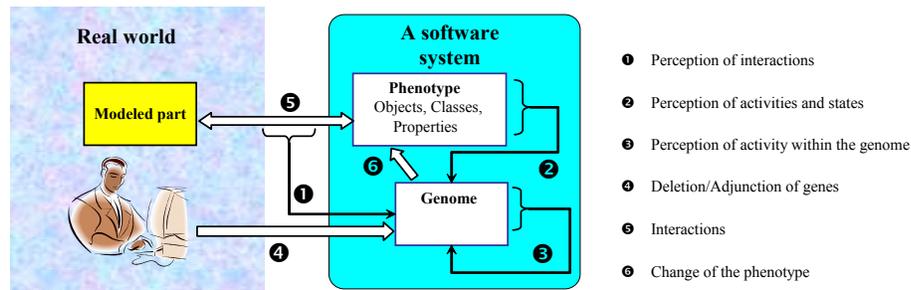


Fig. 3: The concepts

### TARGET APPLICATIONS

Our approach has three features that deal with the change process: modeling of the process itself, modeling of changes as a continuous process and autonomy. Each feature is a response to a requirement of practical applications<sup>[9]</sup>.

Our approach aims at modeling applications having intensive and deep changes. While intensive refers to the frequency of changes, deep refers to the scope of changes on the software system. In most cases, a software system controlling a system that is variable is as effective as itself is adaptable. Modeling and embedding of ontogenesis as a continuous process within a software system aims at modeling applications such as simulation of complex

systems and applications where a gradual change is a requirement. For example, the study of biological systems requires the simulation of their continuous development<sup>[4,9]</sup>. The gradual change is an important characteristic of human-machine interfaces<sup>[10]</sup>. Indeed, progressively changing a software system won't cause a hindrance to users accustomed to that system.

Autonomously updating a system is a good feature in adaptive systems that deal with different execution environments<sup>[7]</sup>. The autonomy implies the dynamic updating of a system (i.e. during its execution). It is an important feature in mission critical systems that must provide continuous and uninterrupted services such as air-traffic control, telephone switches, the financial transaction processors and power plants management<sup>[11,12]</sup>.

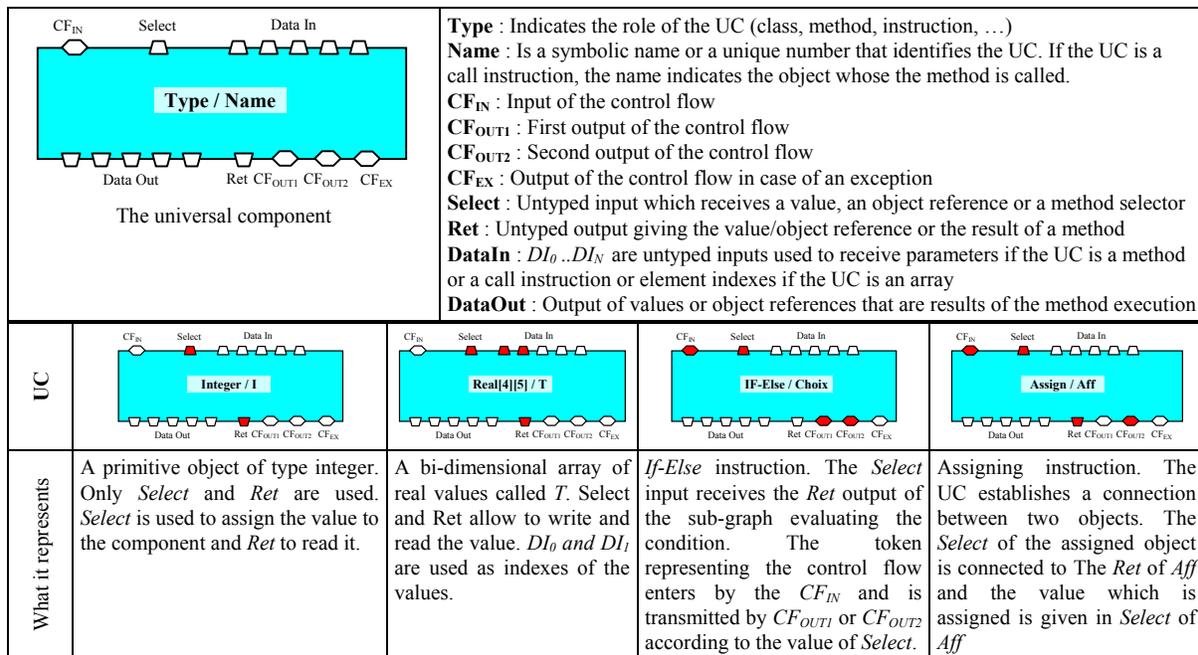


Fig. 4: Modeling structures and statements using the UC

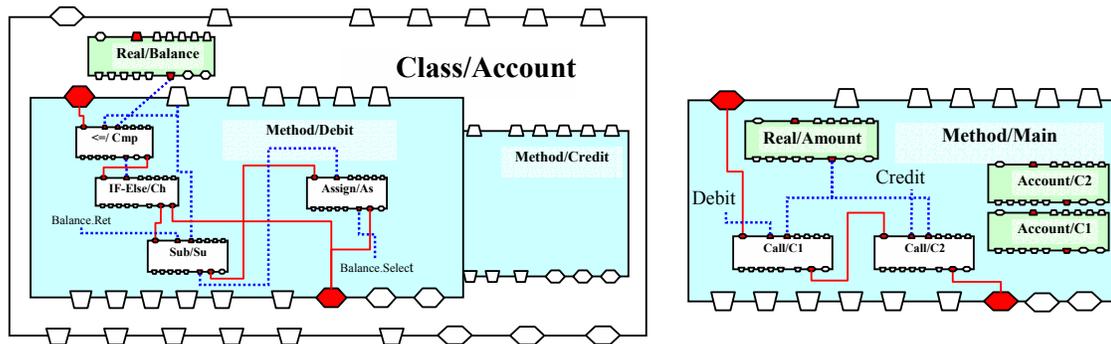


Fig. 5: Example of a phenotype

**THE APPROACH**

The proposed model is based on four concepts: phenotype, genome, interaction and stimuli Fig. 3. We describe the role of each.

As previously suggested, a software system consists of a phenotype and a genome. The genome is the kernel of the software system that initially creates and then continuously shapes the phenotype. For this purpose it perceives its self state, activities and states of the phenotype as well as interactions between the phenotype and the real world. In a previous work, we have considered the phenotype as a set of classes and objects along with various properties, methods and links<sup>[13]</sup>. However, we found this approach difficult when describing genes to affect the phenotype. In this study, we have made a projection of the phenotype

space on a space where we use only one kind of construct called universal component (UC). The phenotype is thus a graph consisting of a multitude of simple or composed UC that are interconnected to form what corresponds to a classical object system. As in<sup>[14]</sup>, the graph of UCs combines data and control flows at the same time. This has the effect of limiting the kind of changes that a gene can achieve.

The UC, we propose, is an abstract element that can represent all the components in a software system program: a graph, a sub-graph, a class, a method, a primitive object, an object of a class, an array. Figure 4 shows the UC and its use in various situations.

Figure 5 shows an example of a phenotype which consists of a class called Account containing two methods (Debit() and Credit()) and a method called

Main() that transfers an amount from one account C1 to another C2.

The genome is composed of a collection of fine grained instructions called genes. They can achieve a development action, a control action or a functional action. Genes are grouped in higher level structures called chromosomes according to the type of their actions:

**Chromosome D** contains constructor or developer genes that change structure and behavior of all entities

**Chromosome F** consists of genes that ensure functionalities such as controlling values or periodically triggering methods

**Chromosome C** consists of genes that control the other by activating and deactivating them

Genes are objects having a structure composed of four parts: Action type, Activation state, Triggering condition and Information part.

The action type indicates the role of the gene. We have identified various actions such as: CreateUC, DeleteUC, DuplicateUC, IntegrateUC, AssignType, AssignName, Activate, Deactivate, Run, Connect, Lock, Unlock, NoAction, ReplaceUC.

The activation state can be Activated or Deactivated. It indicates if the gene can be run. When the activation state is activated, the gene first checks for the triggering condition before executing the assigned action. The triggering condition allows the gene to perceive stimuli such as the existence of an object or some property in an object, activation or deactivation of other genes, comparison of objects values, etc. An active gene, will continuously verify if its triggering condition is satisfied, in such case the assigned action is executed. When the condition is about the other genes (their state or their existence), it allows the introduction of a dependence that expresses a relationship or a mutual exclusion between genes (i.e. gene G1 cannot run until gene G2 is activated/deactivated<sup>[1]</sup>). The information part supplies the necessary information to allow the execution of the gene. The condition part allows the gene to perceive four types of stimuli:

- \* Factual stimuli reflecting the state of a component in the phenotype or the genome (i.e. existence of UCs, existence of genes, connections ...).
- \* Activity stimuli reflecting that currently an activity is executing/started/finished within the phenotype or the genome.
- \* Interaction stimuli reflecting an interaction between the phenotype and the real world.
- \* Temporal stimuli that allow genes to execute independently of the structure of the software system but according to a temporal reference. References can be associated to properties,

objects, classes or to the whole system. This feature allows us to express situations like creating an object O2 after 10 chronons of the creation of object O1. (chronon being a unit of time<sup>[15]</sup>).

The separation between activation state and triggering condition is necessary because some genes are executed only once. For example, the genes that create a class are not used once the class has been created. In the opposite functional genes are deactivated when the class is being created and activated after that. Let's note that if the gene is deactivated; the evaluation of the triggering condition won't take place. This avoids a mistimed execution of genes if the condition remains true and also avoids a repetitive evaluation when we know that in some situation the condition will remain false. The change of the activation state of genes is under the control of genes in chromosome C. All genes are deactivated once their assigned actions achieved unless a clause KeepActive is given in the information part.

The following Table partially shows the genome that creates the phenotype in Fig. 5. Genes are identified using their indexes within the chromosome.

**Computational models:** The approach uses two different computational models. For the phenotype we use a Java-like concurrent computational model while genes are executed like guarded commands, with the command being the action type of the gene and the guard, the triggering condition. What follows gives the semantic of the UC:

- \* UC representing data don't use the control flow input and outputs. Its possible to introduce a value in the component using Select or to read what is stored in, using Ret.
- \* UCs representing methods or statements use the control and data flows. Action assigned to the UC runs when a token representing the control flow arrives at  $CF_{IN}$  and the necessary data are present at  $DI_0..DI_N$ . When the computation is achieved, the control token is transmitted on  $CF_{OUT1}$  (or  $CF_{OUT2}$  if the UC is of type IF-ELSE and Select receives the value false) and results on Ret and  $DO_0..DO_N$ . All incompatibility of data will force the UC to produce an exception that consists in transmitting the token on the  $CF_{EX}$  which is connected to a UC that calls a method to handle the exception.

At the genome level, there is no direct control flow. But an indirect control flow can be forced by control genes. At any moment, we have two pools: active genes pool and passive genes pool. An active gene passes from the active pool to the passive pool when the assigned action is achieved (unless keep active clause is present). A gene passes from the

passive pool to the active pool when a control gene sets its activation state to Activated. The execution system picks randomly a gene in the active pool and if its triggering condition evaluates to true, it executes the assigned action, then put it in the passive pool (in the active pool, if keep active is used).

**Segmentation of chromosomes:** Since modeling the ontogenesis is a complex task, the genome is decomposed in segments (not shown in the previous example) corresponding to classes. The whole software system is considered as a class that contains component classes and in turn those classes may contain other classes and so on. Chromosomes that create a class C1 are associated with the class containing C1. Chromosomes achieving tasks that crosscut many classes C1, C2, ..., Cn, are associated with the class containing C1, C2, ..., Cn.

**Unanticipated changes:** Most unanticipated changes are dealt with by adding genes to the genome (not by deleting them). This is easy to understand since when we want to delete a UC we must add genes that delete it not deleting the genes that have created it, which has no effect on the current phenotype. But in the case of a functional gene that periodically triggers a method; we need also to delete it.

Any change that will affect the software system is first analyzed then coded in the form of genes and added to the genome. At least one of the control genes added must be active to ensure that the needed change will be achieved.

**Implementation issues:** Recall that our approach uses two computational models and hence any implementation must deal with them both. The first implementation was achieved using AspectJ<sup>[16]</sup>. Broadly speaking, aspects in AspectJ are used to model the genome. Since aspects can not be added dynamically to a program, this approach deals only with anticipated changes. First the genome is analyzed to determine all possible classes and properties and then a corresponding AspectJ program is created. After that structures, called markers, are added to indicate for each class or property if it can be used in the current state of the execution of the phenotype. Markers are handled by aspects according to the triggering conditions<sup>[13]</sup>.

The second implementation approach is based on Java and its virtual machine. The main concepts used are proxies and class loaders. In this approach we have added an environment that considers genes as commands and executes them by handling object proxies and class loaders<sup>[17]</sup>.

Other approaches are possible such as creating two virtual machines, one for the phenotype and one

for the genome. While each implements one computational model, they interact in various ways since genome must perceive stimuli and affect the phenotype. Finally, notice that genes can lock or unlock objects to allow a coherent handling of the phenotype, however their use is in charge of the programmer.

## RELATED WORK

**Databases:** Evolution in databases can affect the schema or the instances of the database. There are many approaches that deal with evolution in databases such as triggers and management of versions<sup>[18]</sup>. However the evolution is considered as a sporadic phenomenon where specific environment helps the database users to evolve the database and its schema. In a nutshell, there is no modeling of the change process.

**Artificial life:** Evolution, autonomy and adaptability to the environment are key concepts in artificial life research domain<sup>[4]</sup>. However, we note a strong tendency toward an evolutionist approach where the phenotype of an individual is directly coded in the genome. Therefore ontogenesis is not considered. We found in<sup>[9]</sup> an interesting approach to generate autonomous agents using both phylogenesis and ontogenesis. Unfortunately, the developed agents are too simple and far from any practical use in software engineering. Our approach is comparable since it partially shares the same goal; but in the current state of our research we are concerned only by ontogenesis of practical systems. In our approach, the genome is not a direct mapping of the phenotype and the use of the UC allowed us to reduce the gap between the classical binary encoding of the genome and the phenotype functionalities. Thus, we let some extent to the emergence of functionalities when using phylogenesis. In<sup>[19]</sup>, we found the description of a project whose goal is to create hardware platforms that can develop and support evolutionist systems. Concretely speaking, the project is about self-reproducing and self-healing integrated circuits. It has a low-level approach that is not confronted to the same problems than our approach. Indeed, the phenotype of the obtained circuits has limited functionalities that are far from what we can get when using the UC.

**Separation of concerns:** Separation of concerns approaches share the same principle, that of separating aspects such as synchronization, optimization and security from the software functionalities<sup>[16]</sup>. In this work, we consider ontogenesis as an aspect and we describe it separately. Even if there is strong dependency between the genome and the phenotype, the separation is clear when considering the dedicated

Chromosome C (genes 1 to 6)

1	Activated	True	Activate	D[1..4], D[8..11], C[2]
2	Deactivated	Exists Account	Activate	C[3..4]
3	Deactivated	Exists balance & Dedit & Credit	Activate	D[5..7]
4	Deactivated	Exists Debit	Activate	C[5]
5	Deactivated	Exists Cmp & Ch & Su & As	Activate	D[12..15], C[6]
6	Deactivated	Deactivated D[12..15]	Activate	D[16..23]

Chromosome D (genes 1 to 23)

1	Deactivated	True	CreateUC	Type Class, Name Account
2	Deactivated	True	CreateUC	Type Real, Name Balance
3	Deactivated	True	CreateUC	Type Method, Name Debit
4	Deactivated	True	CreateUC	Type Method, Name Credit
5	Deactivated	True	IntegrateUC	Balance In Account
6	Deactivated	True	IntegrateUC	Debit In Account
7	Deactivated	True	IntegrateUC	Credit In Account
8	Deactivated	True	CreateUC	Type <=, Name Cmp
9	Deactivated	True	CreateUC	Type If-Else, Name Ch
10	Deactivated	True	CreateUC	Type Sub, Name Su
11	Deactivated	True	CreateUC	Type Assign, Name As
12	Deactivated	True	IntegrateUC	Cmp In Debit
13	Deactivated	True	IntegrateUC	Ch In Debit
14	Deactivated	True	IntegrateUC	Su In Debit
15	Deactivated	True	IntegrateUC	As In Debit
16	Deactivated	True	Connect	Debit CFIN with Cmp.CFIN
17	Deactivated	True	Connect	Cmp.CFOUT1 with Ch.CFIN
18	Deactivated	True	Connect	Ch.CFOUT1 with Su.CFIN
19	Deactivated	True	Connect	Ch.CFOUT2 with Debit.CFOUT1
20	Deactivated	True	Connect	Su.CFOUT1 with As.CFIN
21	Deactivated	True	Connect	As.CFOUT1 with Debit.CFOUT1
22	Deactivated	True	Connect	Debit.Select with Cmp.Select
23	Deactivated	True	Connect	Cmp.Ret with Ch.Select

concepts and computational model. We use a stimuli model that is richer than the join point model of aspectJ<sup>[16]</sup>. Another feature of separation of concerns approaches is their reflective capability which preserves information about the source program and makes it available during execution<sup>[20]</sup>. Our approach cannot be implemented without using such capability.

**Dynamic updating:** Updating a software system during its execution is now becoming an attractive research area<sup>[11,21]</sup>. Dynamic updating approaches are dedicated mainly to unanticipated changes. Although they use various dynamic linking mechanisms, they don't consider the modeling of ontogenesis as an objective.

**Component based approaches:** Those approaches aim at providing suitable solutions to problems such as reuse, component deployment, interoperability and so on<sup>[22,23]</sup>. Our model doesn't have the same goal; however the universal component is partially inspired by the Fractal component model<sup>[23]</sup>, where component can be recursively composed. The UC abstracts the software system entities and gives them a uniform appearance. To this purpose, the UC is universal, simple or complex and freely composed and connected.

## CONCLUSION

In this study, we proposed a new model where a software system includes the structural, behavioral and ontogenetic dimensions. Inspired by the biological development, our approach proposes a radical view of the change process and its modeling. To enforce this point of view, we have considered that, initially, the phenotype does not exist, but begins to exist as a result of a continuous activity of the genome. The phenotype is uniformly described using the universal component while the genome is described using three types of genes, each with a specific role. In addition to its naturalness, our approach deals uniformly with anticipated and unanticipated changes.

Modeling ontogenesis remains a challenging task for which proposing suitable concepts is important but not sufficient to master all the subtle problems involved. Indeed our experience with the object model shows us that the methodological issues, such as those dealt with in UML, are very important. For example, how to analyze the future of a real world domain? How to extract changes that must be achieved on a giving software system? ... Before investigating such methodological issues we will first consider future work such as finding ontogenesis patterns.

## REFERENCES

1. Gilbert, S.F., 2003. *Developmental Biology*. Seventh Edn. Sinauer Associates Inc. Publishers.
2. Lewin, B., 1999. *Genes VII*. Oxford University Press.
3. Ridley, M., 1996. *Evolution*. Second Edn. Blackwell Scientific Publications Ltd. Oxford.
4. Adami, C., 1998. *Introduction to Artificial Life*. Springer-Verlag, New York, Inc.
5. Forrest, S., 1996. Genetic algorithms. *ACM Computing Surveys*, 28: 77-80.
6. Mätzel, K.U. and W.R. Bischofberger, 1997. Designing Object Systems for Evolution. *Theory and Practice of Object Systems*, 3: 265-283.
7. Kistler, T. and M. Franz, 2003. Continuous program optimization: A case study. *ACM Trans. Programming Languages and Systems*, 25: 500-548.
8. Shultz, U.P. *et al.*, 2003. Automatic Program Specialization for Java. *ACM Trans. Programming Languages and Systems*, 25: 452-499.
9. Dellaert, F. and R.D. Beer, 1996. A Developmental Model for the Evolution of Complete Autonomous Agents. In P. Maes, M. Mataric, J. Meyer, J. Pollack and S. Wilson (Eds.), *From Animals to Animals 4: Proc. Fourth Intl. Conf. Simulation of Adaptive Behavior*, pp: 393-406.
10. Calvary, G. *et al.*, 2001. Supporting Context Changes for Plastic User Interfaces: A Process and A Mechanism. *Proc. HCI-IHM*, Blandford, A., Vanderdonckt, J., Gray, P. Eds, BCS Conf. Series, Springer Publications, pp: 349-363.
11. Bierman, G. *et al.*, 2003. Formalising Dynamic Software Updating. *Second Intl. Workshop on Unanticipated Software Evolution*, Warsaw, Poland <http://joint.org/use/2003/>
12. Ronström, M., 2000. On-line Schema Update for a Telecom Database. *Proc. 16th Intl. Conf. Data Eng.*, pp: 329.
13. Meslati, D. *et al.*, 2003. L'Auto-Evolution par MAGE: Une Approche Génétique Orientée Aspects, ISPS'2003. 6th Intl. Symp. Programming and Systems, Algiers (in French).
14. Honitrianiela, R. and I. Bertrand 200. Seamless Integration of Control Flow and Data Flow. 16th Intl. Conf. Computers and Their Applications (CATA- 2001), Washington USA, pp: 28-30.
15. Ozsoyoglu, G. and R.T. Snodgrass, 1995. Temporal and Real-Time Databases: A survey. *IEEE Trans. Knowledge and Data Eng.*
16. Kiczales, G. *et al.*, 1997. Aspect-Oriented Programming. In *Proc. ECOOP'97, Eur. Conf. Object-Oriented Programming*, Springer-verlag.
17. Boutbicha, M.R. and M.A. Bouzidi, 2004. JVM: Implementation of Mage using Java. Master Project, University of Annaba.
18. Li, X., 1999. A Survey of Schema Evolution in Object-Oriented Databases. *Proc. 31st Intl. Conf. Technol. of Object-Oriented Language and Systems*, pp: 362-371.
19. Tempesti, G. *et al.*, 2002. A POEtic Architecture for Bio-Inspired Hardware. *Proc. 8th Intl. Conf. Simulation and Synthesis of Living Systems (Artificial Life VIII)*, Sydney, Australia.
20. Cazzola, W. *et al.*, 2000. *Reflection and Software Engineering*, LNCS 1826, Springer-verlag.
21. Vandewoude, Y. and Y. Berbers, 2002. An Overview and Assessment of Dynamic Update Methods for Component-oriented Embedded Systems. In *Proc. Intl. Conf. Software Engineering Research and Practice*, Las Vegas, USA.
22. Brereton, P. and D. Budgen, 2000. Component-Based Systems: A Classification of Issues. *Computer*, 33: 54-62.
23. The objectWeb Consortium, <http://fractal.objectweb.org>