Software Systems Evolution, Free Will and Hyperincursivity*

Juan Jesús Torres-Carbonell
Secretaría de Estado de Telecomunicaciones y para la Sociedad de la Información
Ministerio de Ciencia y Tecnología
Palacio de Comunicaciones, Plaza Cibeles s/n, 28071 Madrid, Spain
Fax: +34 91 346 15 52 - e-mail: jj.torres@setsi.mcyt.es

José Parets-Llorca
Departamento de Lenguajes y Sistemas Informáticos
E.T.S. Ingeniería Informática
Av. Andalucía, s/n, 18071 Granada, Spain
Fax: +34 958 24 31 79 - e-mail: jparets@ugr.es

Daniel M. Dubois
Centre for Hyperincursion and Anticipation in Ordered Systems,
CHAOS asbl, Institute of Mathematics, B37, University of Liège
Grande Traverse 12, B-4000 Liège 1, Belgium
Fax: +32 4 366 94 89 – e-mail: Daniel.Dubois@ulg.ac.be
http://www.ulg.ac.be/mathgen/CHAOS/CASYS.html

Abstract
During software evolution, the next state to be reached by the system, can be an unknown situation that could have unwillingness effects over the evolutionary process. Additionally, the modifications suffered by a Software System may affect the structure or the way of use of that structure. In order to avoid undesired effects, changes must be known by the system. The knowledge of the future states moves us to consider these systems as Incursive Discrete Strong Anticipatory Systems. As well, the evolution has hyperincursive characters because there exists several possibilities of modification, before selecting the next state.. As a consequence we have a non directed evolutionary process in order to obtain a correct and adequate evolutionary sequence for the system. This process is also not hazardous, but under the modeller free will.
Keywords: Hyperincursion, incursion, anticipation, software evolution, system.

1 Introduction

We can define the state vector S(t) of a Software System by three variables: *processing structure, entry structure and memory*. Besides, the Software System evolutionary framework that is supposed to be adequate cannot be random, but based in the knowledge of the future changes to be performed.

* This research is partially supported by R+D projects of the Spanish MCYT (TIC97-0593-C05-04 and TIC2000-1673-C06-04).

This implies that the state vector at time t+1 (after a modification) must be dependent on the following components:

1) The current state vector S(t) at time t,
2) The modification (the modeller knows what is the modification to be produced) and
3) The state vector S(t+1) at time t+1 .

According to this perspective, a Software System can be considered as an Incursive Discrete Anticipatory System, as have been pointed previously [Torres-Carbonell and Parets-Llorca , 2001].

In addition, the modeller is the manager of the software development process and consequently is the decision-maker in the evolutionary process. This capacity to make decisions means the capacity to choose amongst multiple potential future states of the Software System from current state. This capacity of selection is a very important characteristic of the modelling process and implies a potentiality of the modeller which can be considered as free will. Also, according to the characteristics of the system and the modeller the system could reach several possible states during the evolutionary process. All these possibilities collapse on one unique state that will be the next state of the system. This multiplicity of possibilities and the selection of one particular state amongst the set of potential states define Hyperincursive Anticipatory System.

Furthermore, Software Systems is unpredictable before the selection, due to the multiplicity of future possible states.

Finally, all these modifications are performed by the modeller. Additionally the modeller influences at each step of the evolution of the system, in order to direct, redirect, change or eliminate any evolutionary action yet performed or to be performed. The role of the modeller in the modelling system is always very important even if we do not mention it continuously in further sections.

The paper is organized as follows. First of all in Section 2 introduce our concept of System definition and in Section 3 system hierarchy is presented. Section 4 discusses briefly Software System evolution and presents its formal elements. In Section 5 we present the incursive character of Software Systems evolution defined in terms of the previous formal elements. A presentation of the hyperincursive character of Software System is introduced in Section 6, together with the introduction of the concepts of Unpredictable Hyperincursive Anticipation and Alternative Hyperincursion. Finally, Section 7 gather the conclusion and future work.

2 System Definition

For us, following [Parets-Llorca, 1995] and [Parets et al., 1994A, 1994B, 1996], a Software System (SS) is a set of processors that interact between them and with the environment, in such a way that the whole Software System could be viewed as a processor from the functional point of view. Furthermore, the functionality of a Software System as an object is reached by the processors that belong to it, through the activation of theirs actions.

Parets-Llorca [1995] proposes a representation of Software Systems using concepts from the General System Theory [Le Moigne, 1977] processors, environments,

systems, actions, events and decisions. This structure use a historical representation of the functioning (System Functional History) and a historical representation of the structure (System Structural History). [Parets-Llorca, 1995] introduces the concept of Software Metasystem (MS): a System that allows the interaction between the Software System and the Development System, and he propose the definition of the Software System from three different points of view: structural, functional and evolutionary.

2.1 Structural Definition of a Software System

The elements of the basic structure of a Software System are the followings (Figure 1):

Definition. Structure of a Software System. The structural composition of a Software System is:

SOFTWARE SYSTEM = (IA, IE, Pi, H, SD, SG)

1. *Action Interface (AI)*: Using this interface messages and functional actions pass to and from de System (the SS and the MS). This is the way to communicate with the System's *functional* environment.
2. *Evolution Interface (EI)*: Through this interface structural actions and messages related to them pass to and from the System. Actions and messages come from the Metasystem to the Software System, and from the Modeller to the Metasystem (the *development* environment).
3. *Processing Structure*: Set of processors that work within the system, performing actions.
4. *System Functional History (SFH)*: Memory of the functional actions, represented by functional events, performed by the System or its processors.
5. *System Structural History (SSH)*: Memory of the structural actions developed over the system, represented by structural events.
6. *System Decisional History (SSH)*: Memory of the decisional actions developed over the system, represented by decisional events.
7. *Decision Subsystem (DS)*: Processors that take decisions about the actions performed by the System
8. *Genetic Subsystem (GS)*: Processor that take decision about the evolutionary actions.

   1.- AS = Adaptation Subsystem.

   Processor of the Genetic Subsystem that make the decision about adaptive actions, either if this actions affect the structure or not

   2.- IS = Inheritance Subsystem.

   Processor of the Genetic Subsystem that make the decision about producing new Software Systems.

The main functional characteristics of this structure are [Parets-Llorca, 1995A]:

1. The actions follow a Stimulus/Elaboration/Response pattern based on the biological analogy. Software System is stimulated by a stimulus event that is stored in its SFH. This event will fire, through the Decision Subsystem, the actions of the involved processors.

2. Every functional events (mainly stimulus and response events) are stored in the SFH.
3. Decisions are made using stored events of the SFH, and having into account the history of previous decisions.
4. The processors of a system work in parallel.

Processors and systems are isomorphic, and this implies that a processor can be substituted by a Software System.

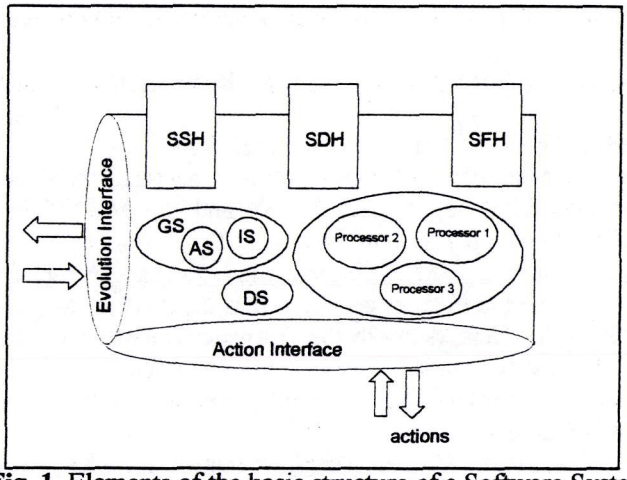


Fig. 1. Elements of the basic structure of a Software System

2.2 Functional Definition of a Software System

From our point of view, a Software System is a set of processors that interact between them and with the environment. The SS is also a processor from the functional point of view. So, the functionality of the SS as an object is attained by the contributions of the processors, through the activation of theirs actions, having into account the history of the system and de action conditions.

Actions are related to events that symbolise them. These events are symbols of the execution of actions and are recorded in the history of the System. [Anaya et al., 1996] proposed that the functionality of a System is determined by:

1. Spontaneous actions performed by the System's processors when the established conditions are satisfied according to the Decision Subsystem.
2. Execution of actions required by the environment through the Action Interface.
3. Execution of actions started by 1) and 2).

The functional structure of a Software System is the following:

Definition 1 System Functionality. The System Functionality is defined as SF =(PS, ES, M) where PS is the processing structure, ES is the entry structure and M is the system memory, s.t. (such that):

System Functionality: (PS, ES, M)

PS = {P | P is a processor for the System}

P = { AS | AS is an action in the System}

ES = {AS | AS is an action of the System Interface}

This definition allows that the System behaves as a processor, and so it can replace a processor. We could say that the ES of a System is a processor.

AS = (f_{ACT}, Conditions)

f_{ACT}: Activity Function.

Conditions: established over the activity function.

M = memory of the System: (SFH, SSH, SDH)

SFH = sequence of stimulus happened over the ES and events of the PS. Represents the sequence of actions performed through time, i.e. the *functional state* of the System.

SSH = sequence of structural events that correspond to functional events of the Metasystem. Represent the *structural state* of the System.

HDS = sequence of decision events previous, during and at the end of the actions. Represent the *decisional state* of the System.

Following [Parets-Llorca, 1995], we think that a Metasystem, which is part of the development system, exists. Its main function is to perform changes in the structure of the Software System. The Software System and the Metasystem are isomorphic, that is to say, they have both the same structure, and the Functional History of the Metasystem records its functional actions. This implies that the Structural History of the System is a subset of the Functional History of the Metasystem.

Both Software System and Metasystem can perform several adaptations following the same models. In the rest of the paper we will use the term *System* to include Software Systems and Metasystems, unless we make an explicit distinction.

2.3. Evolutionary Definition of a Software System

The evolutionary structure of a Software System, according to [Parets-Llorca, 1995] the following:

Definition. Evolutionary Definition of a Software System. The evolutionary description of a Software System is:

SOFTWARE SYSTEM: (SS, SG, IE, HS)

where:

i. *SS: a Software System,* as has been described.

ii. *GS: Genetic Subsystem*

A processor of the system with the following subsystems:

1.-IS: Inheritance Subsystem, *a processor of the system that carries the following actions:*

- Produces new SS.
- Produces adaptations with structural changes.

2.- *AS: Adaptation Subsystem*, a processor of the system that perform the following actions:

- Produces adaptations without structural changes.

iii. EI: an evolutionary interface.

Composed by the actions of the GS.

iv. SH: System History.

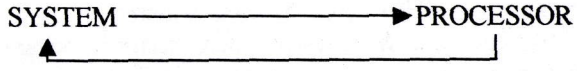
1. SSH: a structural history that records the events produced by the actions of the GS and the stimulus of these actions.
2. SFH: a functional history that records the events produced by the actions of the SS and the stimulus of these actions.
3. SDH: a decisional history that records events produced when making decisions of the SS and the stimulus related to these actions.

The evolutionary description is fully active in the Metasystem, because its Inheritance Subsystem make decision about the production of new Software Systems and about the adaptations that produce structural modifications.

3. System Hierarchy

We can think on a Software System as a processor. So, every processor can be substituted by a system, and every system can perform as a processor of other system.

Using the graphical symbols from Morin, the relation between Software System and processor, can be represented in a circular way:



This relation implies the existence of an isomorphism between system and processors, and neither the system nor the processor is more complex.

The traditional architecture of a Software System establishes two different hierarchy:

1. The *control* hierarchy that establishes hierarchical dependency of the modules of the system.
2. The hierarchy of the *structure* and the *functionality*, that divide the functions to be performed hierarchically.

In our proposal, the interaction between processors allows the construction of systems in which the control is not hierarchical, but the functionality is hierarchical, that is, the action of a system is a construction made from the actions of the processor of that system.

In order to approach the hierarchical character of Software Systems. We can think about a system like the one of the Figure 2. In this system we can identify Systems/Processors that gather others Systems/Processor, that is to say, we could think

on P_i being a program that has a component, P_{ij}, that is a subprogram, and which also has P_{ijk} as sub-subprogram.

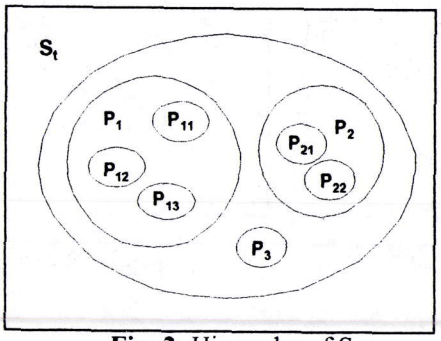


Fig. 2. Hierarchy of S_t.

This set of programs (P_i, P_{ij}, P_{ijk}) can undergo an evolutionary process in which there could be different modifications:

- Suppression of a component. In Figure 3 suppression of P_3.
- Creation of a component. In Figure 4 creation of P_4.
- Modification of a component. For example, modification of P_2 = Suppression of P_2 + Creation of new P_2.

Having into account this hierarchical relationships, the evolution of P_1 could be, for example, the following:

$P_1^t \longrightarrow P_1^{t+1}$ This modification implies:

$P_{11}^t \longrightarrow P_{11}^{t+1}$ (modification of P_{11})

$P_{12}^t \longrightarrow P_{12}^{t+1}$ (modification of P_{12})

$P_{13}^t \longrightarrow P_{13}^{t+1}$ (modification of P_{13})

These modifications suppose that inside each sub-program modification, that is to say, moving from P_{ij}^t to P_{ij}^{t+1}, there are creation, suppression and modification of instructions, moving from the current set of instructions to the new one:

$$\{I^t\} \longrightarrow \{I^{t+1}\}$$

The consequence of the hierarchy is the following:

- The modeller defines first the *global* next step of the Software System:
  $(P_i^t \longrightarrow P_i^{t+1})$.
  To go ahead with the modifications he will look then to the sub-programs:
  $(P_{ij}^t \longrightarrow P_{ij}^{t+1})$
  Then he look into the instructions, what is a *local* approach:
  $(\{I^t\} \longrightarrow \{I^{t+1}\})$.
- The modifications are made in the reverse sense, that is, from *local* (instructions) to *global* (Software System).

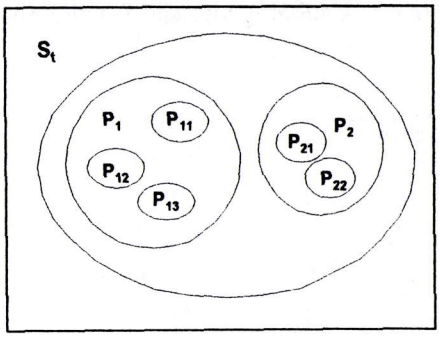


Fig. 3. Suppression of P_3.

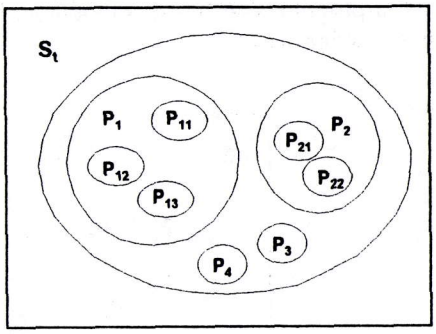


Fig. 4. Creation of P_4.

4 Software Systems Evolution

Software Systems are complex conceptual artefacts that gather together sets, relationships and compound components. These systems evolve, exists within an evolutionary context, and maintain a complex interaction with it, in such a way that their behaviour is not absolutely and certainly set up.

The evolution of Software Systems, consequence of their interaction with the environment (both at the developing and the functioning stages of the process), could be seen as the performed modifications of the system at one of the following lifecycle phases:

1. Modifications during the development process.
   From the conception of a Software System to its implementation, a series of changes are produced. They are needed in order to obtain a software product that works under the proposed restrictions and specifications. These modifications are approached using methods and development tools that allow the modeller to enjoy a certain degree of flexibility in order to assimilate the continuous changes, see for example [Boehm, 1986; Jacobson, Booch and Rumbaugh, 1999].
2. Modifications during the life of the system.

2.1. Software maintenance.
   These changes are traditionally considered as a consequence of three different requirements: 1) System adaptation to changing environmental conditions; 2) Addition of new capacities not envisioned in the system development phase; 3) Repairing of weaknesses, deficiencies or errors detected during the functioning of the system. Usually, these changes are approached using software developing techniques that allow modularity and reuse as [Booch, Rumbaugh and Jacobson, 1999].

2.2. Software function self-adaptation
   Changes derived from foreseen modifications of the Software System as a consequence of the functioning activity. Such modifications are internal to

the system and involve the improvement of operation by means of algorithm adaptation and parameter modifications. These changes are usually approached using local adaptation techniques, such as neural networks or genetic programming [Koza, 1992; Holland, 1975].

This classification of Software System changes, which would be agreed by most software engineers, highlights two important questions not considered in traditional Software Engineering research:

1. The *ad-hoc* treatment of each kind of change by *ad-hoc* tools and approaches. In fact, the fields of knowledge and research involved in each type of change have no relationships. Consider, for instance, that modifications incurred in the development process are separated from software maintenance, and that self-adaptation is not a usual topic in Software Engineering.
2. The absence of global models of evolution that unify different types of change in a unique and comprehensive framework.

4.1 Needed Evolution

With these necessities in mind, we identified the kind of evolution necessary for Software Systems. As we have exposed in [Torres-Carbonell and Parets-Llorca , 2001], Software Systems evolution can not rely on hazardous processes. This impossibility comes from the necessity of the system to assure the modifications to be performed during the evolutionary process. These modifications should be conducted in order to preserve certain invariants which avoid the problems that could come out as a consequence of the non adaptive character of most random changes.

- Design of new functionality by the Modeller.
- Taking into account the necessities.
- The Modeller knows the modifications to be performed.

This proposal tries to establish an evolutionary framework in which all the changes produced are necessary, in such a way that there is no useless effort. This kind of useless effort could appear when hazardous changes are allowed, because most of these changes are not necessary or not absolutely suitable.

Additionally, and in order to perform adequate modifications, the following knowledge is necessary:

- Former states of the system. This knowledge of the history of the Software System is necessary to act accurately and to avoid the repetition of errors.
- The future state of the system, that is to say, the result of the evolutionary modification.

- The parameters to be modified, that reflect in detail what the changes to be performed are.

4.2 Formal Elements of Software Systems Evolution

In order to approach Software Systems evolution, we propose the following formal elements:

- S_t:
  Current structure of the Software System, composed by:
  PS_t = Processing Structure of S_t.
  ES_t = Entry Structure of S_t.
  M_t = Memory of the system S_t.
- $S_{t|t\_k}$:
  Stage k of the structure S_t.
- S_{t+1}:
  Next structure of the Software System, composed by:
  PS_{t+1} = Processing Structure of S_{t+1}.
  ES_{t+1} = Entry Structure of S_{t+1}.
  M_{t+1} = Memory of the System S_{t+1}.
- $OS_{t\_k|t\_k+1}$:
  Operator that changes the stage (structure plus way of use of that structure) $S_{t|t\_k}$ into the stage $S_{t|t\_k+1}$, without modification of the structure S_t of the Software System.
- OS_t^{t+1}:
  Operator that modifies the structure S_t of the Software System, yielding the new structure S_{t+1}.
  The relationships between these elements are shown in Figure 5.

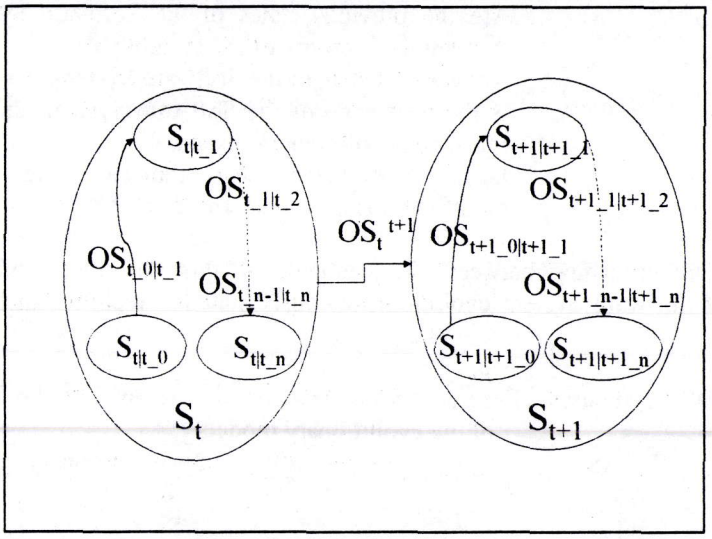
This formalisation establishes the existence, during the life of the Software System, of different structures and different uses of those structures, consequence of the application of operators that modify the structure (OS_t^{t+1}) and operators that modify the stage ($OS_{t\_k|t\_k+1}$). So, the evolution of the Software System could imply:

1. The modification of the current structure S_t, , producing the new structure S_{t+1}, after OS_t^{t+1} is applied.
2. The modification of the current stage $S_{t|t\_k}$, producing the new stage $S_{t|t\_k}$, after the operator $OS_{t\_k|t\_k+1}$ is applied, without structural modifications.

According to this proposal, an evolutionary process can exist within the current structure, just moving from one stage to another, that is to say, by the modification of the use of that structure.

5 Incursion and Formal Elements

As stated in a previous work [Torres-Carbonell and Parets-Llorca, 2001], we consider Software Systems as Incursive Discrete Strong Anticipatory Systems (IDSAS), according to the reasons explained below.



- S_t: Previous structure of the Software System.
- $S_{t|t\_0}$: Initial stage of the structure S_t.
- $OS_{t\_o|t\_1}$: Operator that transform the stage $S_{t|t\_0}$ into $S_{t|t\_1}$.
- $S_{t|t\_1}$: Stage t_1 of the structure S_t.
- $OS_{t\_1|t\_2}$: Operator that transform the stage $S_{t|t\_1}$ into $S_{t|t\_2}$.
- $OS_{t\_n-1|t\_n}$: Operator that transform the stage $S_{t|t\_n-1}$ into $S_{t|t\_n}$.
- $S_{t|t\_n}$: Stage t_n of the structure S_t.
- OS_t^{t+1}: Operator that transform the structure S_t into S_{t+1}.
- S_{t+1}: Next structure of the Software System.
- $S_{t+1|t+1\_0}$: Initial stage of the structure S_{t+1}.
- $OS_{t+1\_o|t+1\_1}$: Operator that transform the stage $S_{t+1|t+1\_0}$ into $S_{t+1|t+1\_1}$.
- $S_{t+1|t+1\_1}$: Stage t+1_1 of the structure S_{t+1}.
- $OS_{t+1\_1|t+1\_2}$: Operator that transform the stage $S_{t+1|t+1\_1}$ into $S_{t+1|t+1\_2}$.
- $OS_{t+1\_n-1|t+1\_n}$: Operator that transform the stage $S_{t+1|t+1\_n-1}$ into $S_{t+1|t+1\_n}$.
- $S_{t+1|t+1\_n}$: Stage t+1_n of the structure S_{t+1}.

Fig. 5: Formal Elements of Software System Evolution

Following Dubois [2000], we adopt his definition of an Incursive Discrete Strong Anticipatory System: an incursive discrete system is a system which computes its current state at time t, as a function of its states at past times, ..., t-3, t-2, t-1, present time, t, and even its states at future times t+1, t+2, t+3, ..., that is to say:

$$x(t+1) = A(..., x(t-2), x(t-1), x(t), x(t+1); p) \quad (1)$$

In the framework of Software System, we can similarly write:

$$S_{t+1} = M(..., S_{t-2}, S_{t-1}, S_t, S_{t+1}; OS) \quad (2)$$

$..., S_{t-2},$	Are the previous states of the Software System known through the Memory of the system (M).
S_t	Is the actual state of the Software System.
S_{t+1}	Is the next state of the Software System, the state at the following evolutionary instant.
OS	Is the parameter that indicates the evolutionary modifications to be performed.

The relationships between the elements of this definition and the formal elements of Software System evolution previously stated, are shown in the following table.

Table 1: IDSAS, Incursive Discrete Strong Anticipatory System, and the formalization of the evolutionary models

IDSAS	Formalization of the evolutionary models
x(0)	S_0
x(1)	S_1
...	...
x(t-1)	S_{t-1}
x(t)	S_t
x(t+1)	S_{t+1}
p	OS_t^{t+1}:

Additionally, we think that an evolutionary Software System has deterministic characteristics but not predictable. This irresolution implies serious limitations in the availability of future situation knowledge. These limitations refer to the states t+2, t+3, ... (not to state t+1) because they are *a priori* unknown states, and therefore it is impossible to foreseen the evolutionary course of the Software System.

The knowledge of the state t+1 depends on the evolutionary modification that must be performed according to the desire of the modeller. Then the evolutionary course will be influenced, and usually conducted, by the design of the modeller and the necessities of the environment.

Having these concepts in mind, we can identify the following evolutionary influences (Figure 6):

- The modeller acts on (has influence over) the evolutionary development of the Software System.
- The new system characteristics, x(t+1), produced using x(t) are influenced by the system history (M) that helps to know former states and their characteristics (x(t-1), x(t-2), ...)
- The new state is also influenced by the characteristics of its new configuration. This characteristics are known by the modeller and are a consequence of the knowledge of the necessary modification, p, to be performed.

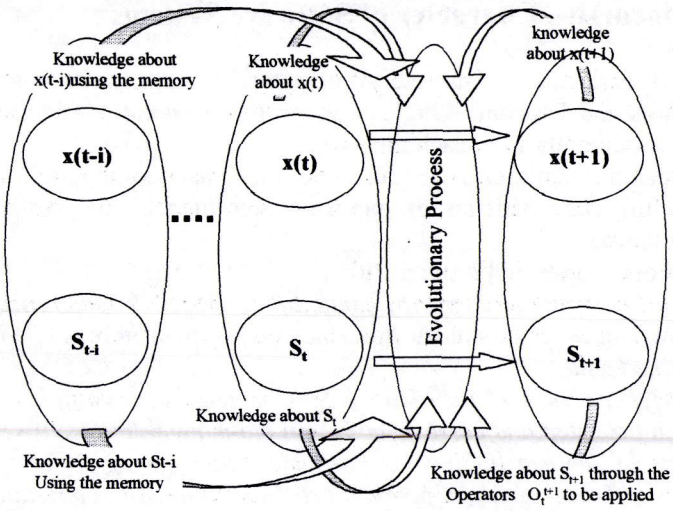


Fig. 6: IDSAS and the formalization of evolutionary models

Consequently, a Software System can be considered as an Incursive Discrete Strong Anticipatory System, subjected to the following considerations:

1. The initial situation of a Software System (state x(0), structure S_0), during its development, can be considered the first idea of the modeller or the user about the system, the starting point of the development. During the functional life of the system, the initial state can be considered as the moment when the system is released to the environment.
2. The modeller and its environment have an important role in the request and achievement of the modifications.
3. The necessary modifications, given by the parameter p, are also known during each step of the evolutionary process. Because the modeller knows the necessary change, it depends on his design.
4. The new situation of the Software System is a consequence of the modification that will be performed. This new situation is the following state of the structure, S_{t+1}, after a structural change from S_t to S_{t+1}, in the evolutionary process.
5. The new situation represents the future Software System. This new situation is the final state on each evolutionary step. Only the next situation is known, because the conditions of the environment and the design of the modeller are subjected to changes during the life of the system, and will change in the future.
6. At each evolutionary step the following situation of the system is the only one that is possible according to the system behaviour and the modeller design. Then, this is the unique final state known within each evolutionary step.
7. The state vector at time t+1, S_{t+1}, will be the set composed by PS_{t+1}, ES_{t+1}, M_{t+1} at that moment of the time. Its composition depends upon the characteristics acquired during the life of the system.

6 Hyperincursive Character of Software Systems

A Hyperincursive Discrete Anticipatory System, according to the definition from [Dubois and Resconi, 1992], is an Incursive Discrete Anticipatory System that produce several iterations at each time step.

Under this definition we can understand that the condition of hyperincursion consist on the characteristics of incursion presented by the system, although with multiple solutions.

Dubois exposes in [2000, p. 29]:

"Karl Pribarm asked me (by email, after CASYS'99 conference):
How can an anticipatory hyperincursive system be modelled without a future defined goal?.
A Hyperincursive Anticipatory System generates multiple potential states at each time step and corresponds to one-to-many relations. A selection parameter must be defined to select a particular state amongst these multiple potential states. These multiple potential states collapse to one state (amongst these states) which becomes the actual state".

This point of view from Dubois, could be understood as the existence of a "double step" (Figure 7) in order to select the actual state:

1. The first one implies the production of multiple possible states at each incursion, that is, at each time moment the future state of the system is anticipated and there are many possibilities. So, a Hyperincursive Discrete Anticipatory System produces a series of potential states.
2. These multiple states will take part in a "second turn" that deals with the selection of the next state at which the system will be between all the potential states. The selection parameter allows the selection of one of those potential states.

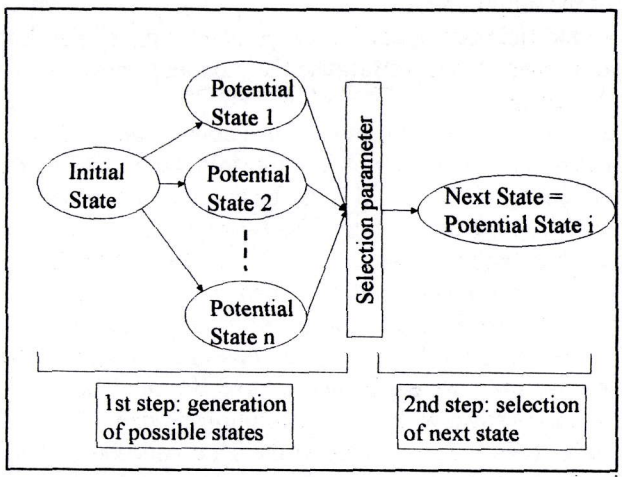


Fig. 7: Software System as HDAS. The actual state selection

When referring to Software System, it must be noticed that the future state, S_{t+1}, depends on past states, present state and the future state, because this state S_{t+1} is one of the set of potential $PS\_S_{t+1\_i}$ (Figure 4).

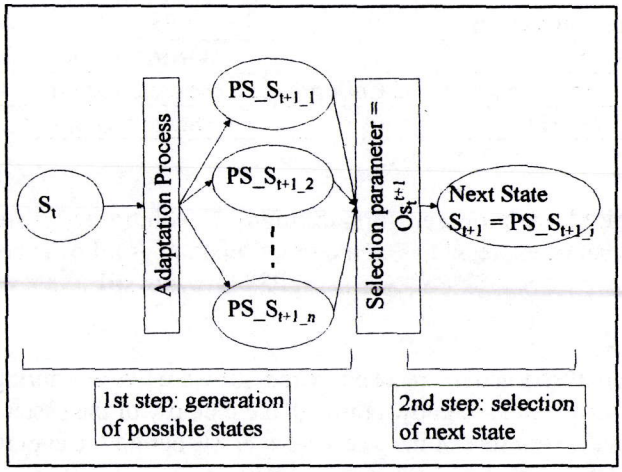


Fig. 8: Behaviour of a Software System as HDAS

From this point of view, this Software System behaves like an Hyperincursive Discrete Anticipatory System (Figure 8), with the following characteristics:

1. From the Software System initial state, S_t, a set of potential states is obtained for the following instant of the evolutionary process ($PS\_S_{t+1\_1}$, $PS\_S_{t+1\_2}$, ..., $PS\_S_{t+1\_n}$), according to the set of selection parameters that must be taken into account.
   The concrete evolutionary action that will be applied, producing a structural change, is not known *a priori*. Then, multiple potential adaptations and structural changes can be produced.
2. Through the selection parameter (operator OS_t^{t+1}) the following evolutionary state is chosen, it will be the next state S_{t+1}.
   This suppose that the system behaves as a Hyperincursive Discrete Anticipatory System, because all the possible adaptations collapse in only one that will produce the selected next state.
   Additionally, the collapse produce the stage $S_{t+1|t+1\_0}$ after the structural modification from S_t to S_{t+1}.

Schematically, the relationship that exists between the concepts of Hyperincursive Discrete Anticipatory System and the proposed formal models could be gathered as follows (Table 2):

Table 2: HDAS and the formalization of the evolutionary models

HDAS	Formalization of the evolutionary models
Initial State	S_t
PotentialState_1 ... PotentialState_n	$PS\_S_{t+1\_1}$ ... $PS\_S_{t+1\_n}$
p	Operators O_t^{t+1}
sp	Collapses to a concrete state: the stage $S_{t+1\|t+1\_0}$
x(t+1)	Stage $S_{t+1\|t+1\_0}$

Using the above proposed formalization of evolutionary systems, the reached next state, S_{t+1}, can be expressed as an incursive function as follows:

$$S_{t+1} = F(...,S_{t-2}, S_{t-1}, S_t; PS\_S_{t+1\_1}, \ldots, PS\_S_{t+1\_n}; p; sp) \quad (2)$$

where:

..., S_{t-2}, S_{t-1}
: are the structures reached by the Software System during its evolutionary history and are known through the memory of the system (M).

S_t : is the structure of the system that exists before the evolutionary process.

$PS\_S_{t+1\_1}, \ldots, PS\_S_{t+1\_n}$
: are the multiple reachable states, as initial stage when structure S_{t+1} will be reached.

p : are operators that modify the structure: O_t^{t+1}.

sp : collapses into a concrete state: the stage $S_{t+1|t+1\_0}$.

6.1 Unpredictable Hyperincursive Anticipation

The description of the Software System evolutionary characteristics as Hyperincursive Discrete Anticipatory Systems, gives the possibility of understanding hyperincursion as the fact that a system can have characteristics of incursion with multiples possible solutions, or multiple future possible states, what could be called Unpredictable Hyperincursive Anticipation.

Consequently, the multiplicity has a certain degree of unpredictability when the following state of the evolutionary sequence have to be established. The new state produced is not the unique possible state. Nevertheless, after the modification it will be considered the current stage ($S_{t+1|t+1\_0}$) of the next evolutionary step after the change of structure from S_t to S_{t+1}.

This unpredictability of the evolutionary phenomenon is produced by the impossibility of establishing *a priori* what modification will be necessary at each moment. This modification will depend on the design of the modeller, the conditions of the environment, the new requirements, etc. On the other hand, it has also been pointed out that when a structural modifications happens, Software System moves from

structure S_t to structure S_{t+1} and specifically to stage $S_{t+1|t+1\_0}$, which is, by definition, the default initial stage when the use of a new structure begins.

The fact that the stage of a new structure (that is, structure and the way of use) is determined by a default way of use, does not imply that the evolutionary phenomenon is predictable: the next modification is not known neither the next stage of the system.

6.2 Alternative Hyperincursivity or Double Hyperincursivity

At the moment, we have only mentioned that the evolutionary process consists on moving from state x(t) to state x(t+1), and this change is equivalent to the transformation from structure S_t to structure S_{t+1}.

Besides, the Software System evolutionary framework considers:

- On the one hand, the existence of Adaptation by Mutation-Differentiation, that modifies the structure of the Software System.
- On the other hand, the existence of Adaptation by Accommodation-Learning, that doesn't modify the structure but the way the structure is used.

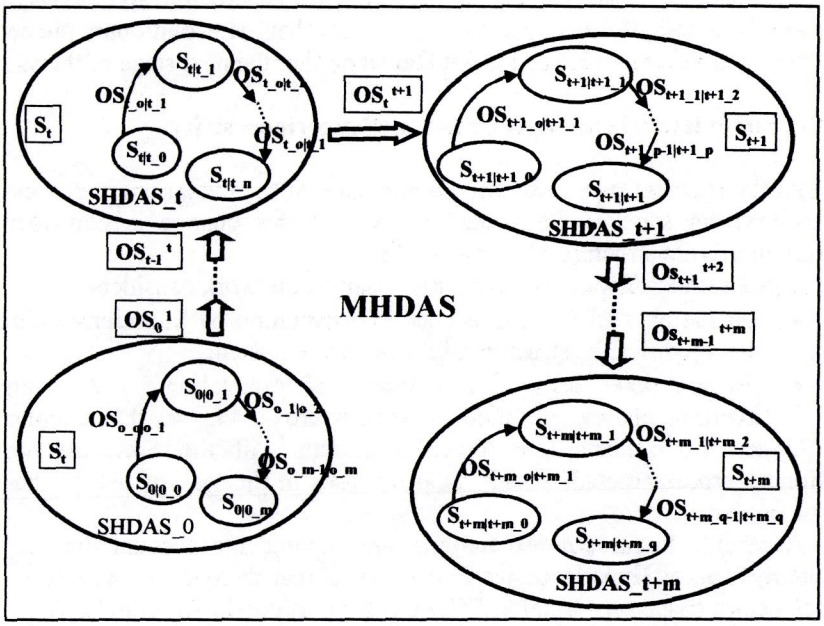
Under this schema, and given a structure of the Software System, an evolutionary process including only modifications in the way of use of the structure could be produced.

According to these considerations, and having into account that we consider Software Systems as Hyperincursive Discrete Anticipatory Systems, we have:

1. Each action based on Mutation-Differentiation implies a structural change, *a priori* unknown, from S_t to S_{t+1}. In this case the system behaves as a Hyperincursive Discrete Anticipatory System, because the structural modification produced is one between the set of possible adaptations.
2. Besides the structural changes, the existence of adaptations by Accommodation-Learning imply changes in the way of use of the structure, for example from $S_{t|t\_i}$ to $S_{t+1|t\_i+1}$. In this case the system behaves again as a Hyperincursive Discrete Anticipatory System, neither in this case is possible to foreseen at each moment which will be the next way of use.
3. Then we can consider that a "double turn" is produced: Each new structure originated by Mutation-Differentiation, produce multiple evolutionary possibilities by Accommodation-Learning. We can think on the existence of "two nested circles" in the following way (Figure 9):
    a. There exist a ring that is the *Main* Hyperincursive Discrete Anticipatory System, that corresponds with a system's structural modifications sequence. This ring embraces several occurrences of:
    b. Secondary Hyperincursive Discrete Anticipatory System, that correspond with the sequence of different way of use of the structure.

This double nesting of a Software System in evolution express that each adaptation by Mutation-Differentiation allows the system to reach a unique stage between all the possibilities that exist before the Mutation-Differentiation happens. Many possible modifications exist and only one of them has happened. Later, within

that structure and starting at the initial reached stage, an evolutionary process without structural modifications could happen.



MHDAS:	Main Hyperincursive Discrete Anticipatory System.
SHDAS_I:	Secondary Hyperincursive Discrete Anticipatory System_ i.
S_t:	Structure S_t of the Software System.
$S_{t|t\_k}$:	State $S_{t|t\_k}$ of the Software System.

Fig. 9: Structure, state and stage of Formal Models

7 Conclusion

Our understanding of Software System Evolution is based on a no hazardous evolutionary process, but certain knowledge about the characteristics of the next state is needed. This knowledge comes from the modeller design as an expression of his free will and having into account the pressure of the environment. The characteristics of the evolutionary process followed by a Software System lead us to think on it as an Incursive Discrete Strong Anticipatory System. As IDSAS the next state is influenced by the present state and the next state, according to a parameter that indicates the evolutionary modifications to be performed. Furthermore, Software Systems are Hyperincursive Systems because during their evolution multiple possible states could be reached, although the selection process will collapse in a unique state. This multiplicity lead us to propose the characteristic of Unpredictable Hyperincursive Anticipation. Besides, we think on Software Systems as Alternative Hyperincursive or Double

Hyperincursive systems with two nested evolutionary process. This abstract construction ought to be mapped into concrete specification languages in order to obtain some profit from it, a work which is being carried out and was initially outlined in [Torres-Carbonell and Parets-Llorca, 1999].

References

Anaya, A; Rodríguez-Fortíz, M.J.; Paderewski, P.; Parets-Llorca, J. (1996). "Time in the Evolution and Functioning of Information Systems". *I Jornadas de Trabajo en Ingeniería y Desarrollo de Software*. Sevilla, 14-15 de noviembre de 1996. JIS'96.

Boehm, B.W. (1986). 'A Spiral Model of Software Development and Enhancement'. *ACM SIGSOFT Software Engineering Notes*. August 11. P. 14-24.

Booch, G.; Rumbaugh, J.; Jabobson, I. (1999). 'The Unified Modelling Language User Guide'. Addison Wesley Longman Inc.

Dubois, D. M. (2000). 'Review of Incursive, Hyperincursive and Anticipatory Systems – Foundation of Anticipation in Electromagnetism'. Computing Anticipatory Systems: CASYS'99 – Third International Conference. Edited by Daniel M. Dubois, Published by The American Institute of Physics, AIP Conference Proceedings 517, pp. 3-30.

Dubois, D., Resconi, G. (1992). '*Hyperincursivity: a new Mathematical Theory*'. Presses Universitaires de Liège.

Holland, J. H. (1975, 1992). '*Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*'. CAMBRIDGE. MA. The MIT Press.

Jacobson, I.; Booch, G.; Rumbaugh, J. (1999). 'The Unified Software Development Process'. Addison Wesley Longman Inc.

Koza, J.R. (1992). '*Genetic Programming*'. CAMBRIDGE. MA. The MIT Press.

Le Moigne, J.L. (1977, 1983, 1990). "*La théorie du système général. Théorie de la modélisation*". Paris. Presse Universitaires de France.

Parets-Llorca, J. (1995). 'Reflexiones sobre el proceso de concepción de sistemas complejos. MEDES: un método de especificación, desarrolo y evolución de sistemas software'. (Ph.D. Thesis). Granada. Dpto. de Lenguajes y Sistemas Informáticos. Universidad de Granada.

Parets-Llorca, J. ; Anaya,A. ; Rodríguez,M.J. ; Paderewski,P. (1994A). 'A Representation of Software Systems Evolution Based on the Theory of the General System' in: 'Computer Aided Systems Theory - EUROCAST'93'. Pichler,F. ; Moreno-Díaz,R. (Eds.). Springer-Verlag. LNCS 763. 96- 109.

Parets-Llorca, J.; Rodríguez, M.J.; Paderewsky, P.; Anaya, A. (1994B). 'HEDES: A System Theory based tool to support evolutionary Software Systems". EUROCAST'99. 69-71. Viena. September 1999.

Parets-Llorca,J.; Torres,J.C (1996) "Software Maintenance versus Software Evolution: An Approach to Software Systems Evolution". IEEE Conference and Workshop on Computer Based Systems (ECBS'96). Friedrichafen. March 1996. pp. 134-141.

Torres-Carbonell, J.J. y Parets-Llorca, J. (2000). '*A Formalisation of Evolution of Software Systems*'. In Pichler, F.R.; Moreno-Díaz, R.; Kopacek, P. (Eds.) 'Computer Aided System Theory – EUROCAST'99', Springer-Verlag. LNCS 1798. pp 439-449.

Torres-Carbonell, J.J., Parets-Llorca, J. (2001). 'Software Evolution. What kind of Evolution?'. Computing Anticipatory Systems: CASYS2000 – Fourth International Conference. Edited by Daniel M. Dubois, Published by the American Institute of Physics, AIP Conference Proceedings 573, pp. 412-421.