High-level Open Evolvable Systems Design By Process-oriented Modeling: Application to DNA Replication Mechanism

Behzad Bastani
Computer Laboratory
University of Cambridge
Cambridge, UK
Behzad.Bastani@cl.cam.ac.uk

Hoda Bastani
College of Medicine
Pennsylvania State University
Hershey, PA
Hoda@psu.edu

ABSTRACT

Open Evolvable Systems' design requires a methodological [1] and conceptual paradigm different from the conventional software design. Evolvable Systems' research [2, 6, 16, and 17] has established itself as a new research field, but the content is more domain-oriented than universal. Consequently, major contributions toward substantiation of that universal methodological and conceptual paradigm are yet to come. In this paper we present a new perspective and method for the general-purpose design of Evolvable Systems. The paper presents the attributes of the Evolvable Systems and discusses the distinction between Evolvable Systems' and conventional software design as well as the methodological ramifications. We pose and address the question of what is an efficient methodology for designing a system for which we do not know the boundaries? We present our version of Process-oriented Modeling as the key method in the high-level design of Evolvable Systems and show its utilization in implementation of one modeling case of a complex Evolvable System, the DNA replication process. We also present the dynamic aspects of the design process management and pre-code verifications in the framework of Quantified Controls and Simulations.

CATEGORIES AND SUBJECT DESCRIPTORS

D.2.10 [Software Engineering]: Design - Methodologies, representation

GENERAL TERMS

Software Engineering, Evolvable Systems, Modeling, Design, Method, Framework.

KEYWORDS

Process-oriented Modeling, Evolvable Systems, Requirements Analysis, Requirements and Architecture modeling, Specification, Design, Design Patterns, Methodologies, Abstractions.

1. INTRODUCTION

The design of complex open evolvable systems poses serious challenges to Software Engineering [2]. The history of Software Engineering has witnessed an evolution of methods from Functional Programming to Object Orientation and Model-based strategies [3, 4]. Yet the methodological advances have essentially referenced systems that we choose to call "finite systems", conceptually being an extension of finite state machines [5] in architecture. A *finite system* in our definition is a system whose boundaries are known to the designer at the design time. Being clear on what an architect or designer intends to build, s/he would use any of those conventional methods, organizing the activities from inception to transition [7] of the system. The lifecycle starts with Requirements inception, then analysis of the system, followed by the general architecture and detailed design, implementation, testing and deployment [7]. At this point the system is complete and finalized, and only needs maintenance and fixes. Any substantial change in the requirements of the system necessitates the reconceptualization [8], redesign and reimplementation of the system. The old system needs to be removed from the deployment platform and the new system installed.

In light of this life-cycle reality, the research question is whether we could design systems that can evolve over the time and adapt to new requirements. There is no doubt that it would be very desirable to have software architectures [24, 27] that could be modified or redesigned [25, 28] while the system is in use without infringing the integrity of the system and while having a consistent running system [26]. The core question then boils down to how we design a line of systems that can architecturally anticipate all possible uses in future as well as environmental changes by being open to continuous modification or redesign while in use.

Research in evolvable systems can be described as mixed, non-uniform in assumptions and premises, domain oriented or subject oriented [8, 9, 10, 11, 12, 13, 16, and 17]. Most of the existing research literature is based on assumptions that limit them to the systems in a specific domain, or have chosen to deal with certain issues that make the results most suitable just for the subject area of the specific issue dealt with [8]. In contrast, we try to look at the subject matter in Software Engineering terms, hence our focus will be on *generic* evolvable systems design. By generic, we mean that the principles and choices are not elected given the requirements of any specific domain or application, although specific domains might need to extend the architecture to respond to their specific system needs or to advance their system performance.

One other issue under consideration was choosing an appropriate

testbed for the implementation of evolvable systems ideas. Although the usual domains like business or command and control modeling could be used, yet the issue of their evolvability is in fact a matter of time realization, something that only the passage of time will disclose. In other words, the issue of evolvability can be true about every domain, however when we model, for instance, a business application, we are normally clear about the dimensions of the system, although we might artificially hold on to some aspects as candidates for evolutionary additions in trial phases. In such a scenario, we are in fact aware of the evolutionary extensions and this might influence our initial modeling considerations. Therefore, we finally reached the conclusion that probably the best testbed for evolvable systems would be biological systems modeling, and that was for two reasons. First, although we might not be able to say biological systems evolve in front of our eyes, however, the science of the complex biological systems is constantly evolving, hence posing the field as a practical evolutionary domain right before us. The second reason was the extreme complexity of the biological entities and their organismic relationship. This extreme complexity creates a situation that deprives us from having a definite baseline for the system. As a result, we almost do not have a clear starting point for modeling the system, recognized as a point from which the system starts. In a theoretical model of a bio-system, almost every point can be the start of a range of larger systems, and also the end of a slew of smaller systems known as member micro-systems. Therefore, whichever point we pick as the start of our modeling activity, the system is subject to expansion in two logical directions simultaneously, the macro direction and the micro direction, as well as engagement of a multidimensional web of dynamic relations in a live 3D model structure. On the other hand, the architectural and operational details of these multilaterally related organisms are so overwhelming that it is truly hard to imagine even one small area of the system with all its details at any one time. This setting satisfies the condition we were looking for initially: starting to model a system whose dimensions were unclear at the time of design. The situation poses the challenge of evolvability from the very beginning, as the modeling methods and practices chosen will have early influence on the advancement of the architectural formation. Presenting only the first set of these methods and practices is the subject of this paper. The paper also presents an implementation path within its scope, which addresses means for architectural management issues and pre-coding verification methods.

By exploring the issues of Open Evolvable Systems' design, this paper addresses two interrelated research questions, one on methodology and the other on architecture:

1. What is an efficient methodology for designing a system where we do not know its boundaries?

2. How could we manage architecturally *consistent* design and development of an open-ended system that is continuously changing?

This paper is presented to reflect partial results of the work on the Open Evolvable Systems research project, which is carried on under the umbrella Pebbles Project, at the Computer Laboratory of the University of Cambridge. The project is funded by the Cambridge-MIT Institute (CMI).

Section 2 presents some fundamental concepts around evolvable systems and our definitions or requirements. Section 3 presents Process-oriented Modeling as we perceive, define and use it, the analytical and logical premises for such adoption, the context of implementation, and the methodological relationship between our version of Process-oriented Modeling and our previously published open-systems modeling and design framework, the Abstraction-oriented Frames [1].

Section 4 presents a system implementation of the Process-oriented modeling within the scope of a DNA replication model as a proof of concept and the elaboration of the method as well as the theoretical issues involved.

Sections 5 and 6 offer the conclusion and the future work.

2. CONCEPTS AND DEFINITIONS

We suggest that there are two distinct concepts of evolution or adaptation for a system to consider: *user adaptation* and *environmental adaptation*. Having *user adaptation* means the flexibility of the system to adapt to the individual user requirements and preferences. This flexibility has structural connotations, meaning the system should be capable of accepting modifications at all levels. However, the insertion point of the modifications is the architectural level so that the system's integrity can be preserved. Along the same concept, the expression of the user choices is expected to be done at a high level without a necessity to employ conventional coding procedures. This assumption and requirement leads to development of systems in which writing the system and running the system would practically be the same undertaking. Environmental adaptation, on the other hand, means the capability of responding to changing environmental and context requirements that might be dynamically presented to the system. This includes the changes resulting from organismic relationships between and among the sub-entities. These two concepts create an architectural loop that continuously interacts with the use of the system, as shown in Figure 1. Yet the loop is not closed but moves on along the time and creates an overall spiral movement, with the activities of figure 1 regenerating continuously.

The assumption for our purposes, contrary to SER [8], is that the amount of change or extending of the system should be unlimited, suggesting a true concept of an Open System. Here it might be helpful to present a definition of Open Systems and Closed Systems as combined with and related to the evolvable systems, as well as some other system concepts related to our analysis.

2.1 OPEN EVOLVABLE SYSTEMS:

An Open System is a functional entity that its design and implementation is not rigidly controlled by a central authority. An Evolvable System is a *consistent entity* that is not confined by pre-set boundaries and can continuously grow and extend to address the environmental changes and/or users' new requirements. From

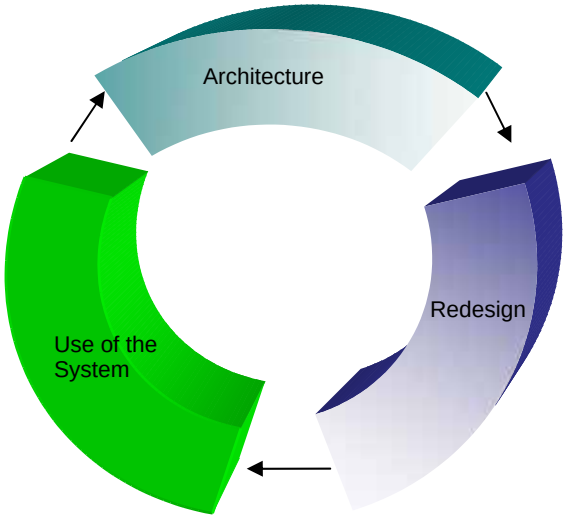

Figure 1: Representing one slice of the system adaptation in an evolvable system

a conceptual point of view, the two sets of definitions we presented are inherently related. An Open Evolvable System is one compliant with both sets of presented definitions.

A key theoretical characteristic of open evolvable systems is having the capability to address not only unanticipated run-time issues [14, 15], but unanticipated concepts at their design time, and ideally at a certain level of maturity in future, without any human intervention, or practically with very little manual intervention. The latter is in fact a "first class design activity" performed by the system on itself [8, 10], but given the present state of such systems, its realization might be considered a somewhat far-fetched goal. As Nardi [18] puts: "*As has been shown time and again, no matter how much designers and programmers try to anticipate and provide for what users will need, the effort always falls short because it is impossible to know in advance what may be needed... End users should have the ability to create customizations, extensions, and applications.... {p. 3}*" Obviously this will require a totally new architecture and methodology.

A challenge of designing evolvable systems is to ascertain that evolutionary extensions would not mean – necessarily – software development. Not all the users are software developers, but it should be possible for them to participate in evolving their systems as their requirements dictate [11]. In such systems, the user has an opportunity to do programming without doing software development in the sense of conventional coding. Although at the beginning a good amount of such activity might be manual, an *automated artifact generation facility* or mechanism should be considered in due course as an advanced requirement for evolvable systems. It should be noted that the automation concept in software systems is practically a moving target, always flying ahead of some level of automation that might be available at any specific point of time.

2.2 CLOSED SYSTEMS:

Closed systems represent the idea of manufacturing finished products that symbolizes the assumption of existence of a rigid boundary between the formation phase of the system and usage phase the system. One of the characteristics of closed systems is that the scope and functionality of the system is essentially limited to the concepts conceived at the design time, therefore new concepts cannot find room in the fixed system. Adding functionality normally is possible through enhancement of the system to a new version [19]. The extent of the change and associated cost depends on how radically the new concept is different from the ones underlying the set system.

2.3 ETERNAL SYSTEMS:

The nature of activities in certain environments requires the software and hardware systems in those environments to be up and running eternally [16] without any downtime or reboot. Examples of such systems are an air traffic control system, a spaceship command and control system, or even at a much less mission critical level, the command and control system in an automated house that should guarantee the ongoing operation of all the systems such as security system and temperature control. Although eternality of operations could probably be obtained in many different ways, including provision of redundant systems for regular operations, and the use of backup systems during software or hardware upgrade times to reduce the downtime to close to zero [20], still these are in fact workaround techniques to give the impression of eternality at the cutoff junctures – an approach similar to wrappers that match legacy systems to newer systems. We believe Evolvable Systems provide the most indigenous and natural foundation for design of eternal systems, as practically there is no need to shift from one set-system to other set-system to require a shutdown or shift of some sort.

2.4 DESIGN ENVIRONMENT VS. DESIGNED SYSTEM:

Conventional systems are designed and delivered to the users, either as a fixed system or with certain reconfiguration and programming capabilities. Normally the design environment and tools are not delivered to the user along with the system [8, 10]. The idea of evolvable systems necessitates that the design environment and tools are also delivered or at least made available to the user, in addition to the system itself. To get users out of the "couch potatoes" [11] mentality, there should be provision of some kind of motivation for the user to develop a designer mindset [8] and feel capable and responsible to contribute toward evolution of their systems outside the tight control of the experts.

2.5 DECENTRALIZED EVOLUTION:

The idea of evolvable systems logically goes hand in hand with the idea of decentralized evolution. A centrally mandated change inherently means pre-programming which is defeating the idea of evolvable systems. The idea of decentralized evolution presents fundamental difficulties to the software architecture of such systems [10, 43]. Can decentralized be interpreted as not having even a centralized authoritative architecture or design, as it is the case with Open Source Software, to regulate the trend of development and change? One of the questions is how could we

keep track of all evolutionary trends and design instances or patterns in a decentralized environment and how could we use such knowledge for naturalizing the evolutionary environment? Whether there is some kind of central control idea or absolutely none, there should also be contemplation about causal elements for a paradigm shift in the evolutionary environment as well as consideration of how the current systems might adapt to such paradigm shift.

2.6 INTER-COMPATIBILITY OF EVOLVABLE SYSTEMS:

When a collection of evolvable systems undergo the evolutionary changes in an environment, there should be a requirement that these systems do not grow incompatible, but certain level of compatibility should be maintained by the evolutionary process.

2.7 DYNAMIC DEPENDENCY MAINTENANCE MECHANISM:

Dependency relationships are part of the natural attributes of a system. The core question is what would be the boundaries of the system. Do we need a centralized mechanism to keep tack of the dependencies, or can we define the system as local units of performance, and communications between them? In any case, it seems there should be some kind of a Dynamic Dependency Maintenance Mechanism to keep track of the volatile dependencies that appear and disappear in the system [12, 21, 22], based on the relative definition of the system and it boundaries.

2.8 SERVICE DISCOVERY MODEL AND FRAMEWORK:

As services are the front view of the system to the user, there should be a friendly model and framework for discovering services [21, 23]. Still this model and framework should be deep enough to describe the architectural and operational aspects of the design of evolvable systems.

2.9 DEFINING ADAPTATION OR EVOLUTION TRIGGER MECHANISM: [30]

Adaptation or evolution is and should be an analog or continuous process by nature. Yet there are major joints in the evolutionary process of a system that should be recognized. In order to be prepared, there should be an understanding of under what circumstances the system should look to adapt or evolve into what can be called a new or improved system. This might be based on a Goals Evaluation [29] System that determines if the current goals are generally satisfied in the most efficient manner, and whether new goals are introduced into the environment.

2.10 ARCHITECTURAL EVOLUTION AND ADAPTABILITY VS. SYSTEM MANAGEMENT AND RECONFIGURATION:

In a number of research papers about evolvable systems [31, 32, 33], proposed views and remedies are of contents that we believe are best categorized under the topic of "System Management and Reconfiguration" not architectural evolution and adaptability [16, 34]. These two concepts should be carefully distinguished, as their conceptual proximity makes them prone to be mixed up easily. Some concrete examples clarify this issue.

2.10.1 CONTROL ENGINEERING concepts are suggested in some literature [15, 49, 50] as means of making evolvable systems. Examples are *feed-forward*, which is feeding specification of the software and its expected behavior into newly designed modules, and *feedback*, meaning gathering and measuring software environmental attributes. We believe proper utilization of control engineering concepts can help the design of evolvable systems, but mere use of them in a system does not categorize the system as an evolvable system, as most of control engineering concepts are designed to help with the reconfiguration of *closed systems*. Reconfiguration is "conceptually minor" or "tactical adaptation" of the software with the changes in the environment, while evolution is the capability of a system for strategic adaptation to new environmental concepts.

2.10.2 DYNAMIC INSTRUMENTATION makes use of instruments such as gauges, probes and monitors (as used in conventional engineering) [51] that can be dynamically attached to application components at runtime (and removed as required) to measure specific runtime parameters and monitor their behavior. Obviously the results of such operations help the reconfiguration of the system, but the capability of such reconfiguration is not enough to qualify a *closed system* as an *evolvable system*.

3. PROCESS-ORIENTED MODELING

The conventional technology for building systems is inclined toward building huge software objects with either multi-layer hierarchical inheritances or multiple-inheritance. Normally these huge and complex objects are used as components of some ontologically predetermined system configurations. We believe so much determinism both at the level of components and system architecture is contrary to the idea of evolvable systems, as application of any change would be so complex and costly with unknown repercussions which would be either impractical or infeasible.

We believe for a system to be evolvable, it needs to be fundamentally as disintegrated as possible. In other terms, it needs to be composed from some modules that are independently accessible and modifiable. The emphasis here is on building systems through "Composition" as opposed to "Inheritance". Logically, a pool of the maximum number of fine grained independent modules would provide the best condition for the maximum number of diverse configurations at any size. From a system point of view, we can categorize the structure of such systems as *loosely coupled*. This should not be interpreted as a ban on the use of the technique of inheritance in such systems. Designers still can make inheritance-based classes and objects for satisfying their system needs, but the architecture or foundation of the system is not established on a web of inheritance-based classes that constrains the flexibility of the system for evolutionary purposes.

As an architectural configuration for Evolvable Systems, we propose a system of "Nuclear Process-Units in an Unspecified Open Chaining Configuration" so that the system could be a composition of such nuclear modules in any desired configuration.

Early in section 2 we defined a requirement that an Evolvable System should be capable of accepting modifications at all levels, however, the insertion point of the modifications should be the architectural level so that the system's integrity could be preserved. Now with this configuration presented, we can offer an extra clarification that the insertion point for modifications would be at the architectural level of each single Nuclear Process-Unit.

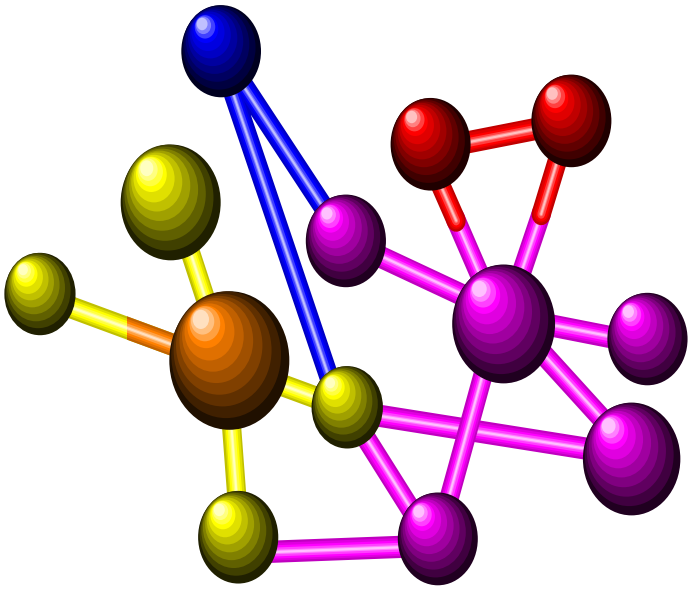

Figure 2: A sample logical view of the overall system with each chain ring representing the Nuclear Process-Unit in an openly connecting or disconnecting pattern, with protected semi-closed systems (indicated by color coding).

Of course, the broader general computing environment is practically divided into protected semi-closed systems like companies, homes, airplanes, human bodies and bacteria, each case of which follows its own individual policies, both as a type and as an individual instance, in terms of providing services, accessibility and security. Hence, the configuration above defines the logic of the architecture and is not a representation of objects.

In this representation, each Nuclear Unit is a logical "Control Station" (CS) abstracting a meaningful process at some level of abstraction, while acting as a fairly autonomous command and control station. Nuclear Process-Unit should not be considered an indivisible unit in the sense of bonding and rigidity of the structure. Given the need for very flexible structures for evolvable systems, the Nuclear Process-Unit is considered to be a unit of optimal benefit, while many such modules can be removed or replaced depending on their role in the system and still have a technically (vs. functionally) working system. By highlighting this attribute, we mean the system will not be technically crippled by removal of a module, although it might not be able to deliver the intended functionality due to missing the functionality of that module. That means there can be some modules in the system that because of their critical role, their removal can cripple the system even technically.

One example of a CS is a laptop or PC that has access to component resources in the back end and presents a user interface in the front end. Such a CS can be used to exert control on a variety of entities in a dynamic environment based on developing network connections. Another example of a light weight CS is a software interface which might be in the form of a programming menu on a digital camera, VCR or any finite state machine display. Along the same concept, we might find a main logical CS in a house with more than one interface, or more than one type of interface, distributed around the house, exerting controls over other devices in its locale and being responsible for the overall computational operations, networking, configuration management and monitoring operations. Yet, it should be emphasized that the control issue should not be viewed only from our perspective, but from a systemic-role perspective as well. A biological pacemaker in the heart that controls the contractions of the heart muscles is an example of a CS from an organismic point of view without us directly operating it. Obviously this type of CS cannot be removed from the system and still have a technically working system, unless it is replaced by an artificial pacemaker that performs an equivalent systemic role. Our theory of evolvable systems will try to give system definitions, roles and implementation paths to all these concepts.

To realize the *user adaptation* and *environmental adaptation* capabilities as defined, we presented three conceptual requirements for the Open Evolvable Systems:

1. The system capability of receiving modifications at all times without the need for complete redesign;

2. The capability of applying high-level architectural choices without a need for conventional coding;

3. The operational synonymity of *writing* the software system and *running* it.

Obviously this requires a roadmap that is completely different from the conventional paradigm of "design and code the system". The critical question is what type of architectural framework would best facilitate realization of these requirements. We propose that Process-oriented Modeling would create the necessary context for design and development of the evolvable systems. It should be noted that in the context of complex systems, Process-oriented Modeling might mean different things to different designers, or might be utilized in quite diverse manners [35, 36, 37 and 38] although the fundamentals might very well be shared. Our perspective on Process-oriented Modeling is explained through our analysis and implementation.

In complex and dynamically changing systems, processes are the most noticeable modules of the system when observed from the outside. This can be portrayed using an example from biological systems. DNA replication is one of the main processes occurring in the cell nucleus. Gregor Mendel initially discovered the process of inheritance in 1866 by observing the transmission of specific traits in pea plants from one generation to the next [44]. However, Mendel did not know which component of the cell was responsible for this transmission, and it was not until the experiments of Avery, MacCleod, and McCarty in 1944 that the cellular component responsible for genetic transmission was found to be

DNA [45]. It was also theorized that two other cellular components, RNA and protein, played roles in DNA replication and genetic transmission, but the exact interrelationship between these three components was only discovered fourteen years later by Francis Crick [46]. Even the basic structural details of the process of DNA replication were not fully understood until the early 1960s when John Cairns demonstrated the process of DNA replication using radioactively-labeled chromosomes that could be seen replicating under an electron microscope [47]. In brief, looking from outside at the complex entity of the cell, scientists could initially only confirm that there was a process which caused DNA to replicate itself. At this level there was no clarity as to the objects involved in this process, including the internal structures of those objects, their interfaces and their interrelationships. The object-level clarifications came as the result of probing the *processes'* traits and variations.

Since we are going to present the DNA replication model in the next section as an application environment for our Process-oriented Modeling, we deem it necessary to first give an overview of the subprocesses involved in the DNA replication process in order to assure better understanding of the developed model. DNA exists as a helical double-stranded structure where each strand consists of a chain of sugar molecules and phosphate molecules interlinked. Also linked to each sugar molecule is one of the following four bases: adenine (A), thymine (T), cytosine (C), and guanine (G). A unit of sugar, phosphate and base together is called a nucleotide. The bases from each of the two DNA strands bond together in pairs, as shown in Figure 3, in a complementary fashion (A and T always pair together, and G and C always pair together). The sequence of bases in a strand of DNA is what makes that DNA strand unique from all others and forms its genetic code.

The process of DNA replication begins at locations on the DNA strands known as origins of replication. It is first necessary to unwind the helical structure of the DNA strands, a process achieved by the enzyme helicase. Once the DNA is unwound and the two strands are separated, exposing the bases on each strand (Figure 4), the DNA is ready to be replicated. The primary enzymes that form the "replication fork," or the site of DNA replication, are the DNA polymerase enzymes (there are five such enzymes named alphabetically from α to ε).

The DNA polymerases scan the original strand of DNA and for every nucleotide that is scanned, a nucleotide with a complementary base is bonded to it, thus creating a new strand of DNA identical to the original opposing strand. Consequently, the two DNA molecules that result from the DNA replication process will each consist of one original strand and one newly created strand (Figure 5).

Considering that in our real-world conceptualization, processes appear as the most noticeable modules of the complex systems, it would be logical to consider processes as the top-level architectural units of modularization in the system modeling, and hence our adoption of a Process-oriented Modeling approach. From a methodological point of view, this is equivalent to Michael Jackson's system modeling theory of the world outside for the

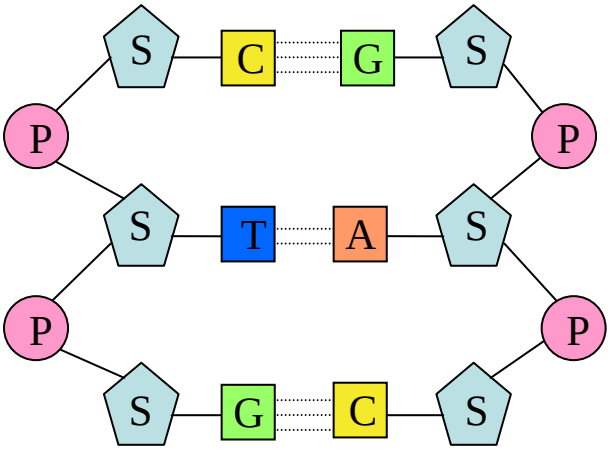

Figure 3: Double-stranded structure of DNA with sugar-phosphate backbone and paired bases. A=Adenine, C=Cytosine, G= Guanine, P= Phosphate, S= Sugar, T= Thymine.

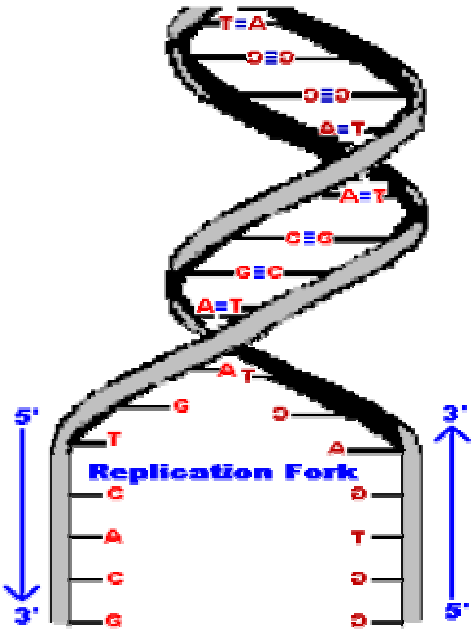

Figure 4: Exposure of the bases after separation of the DNA strands

world of computers: "As software developers, we too, need not aim to disclose the real essence of the phenomena. We can deal with the phenomena as we experience them, as they appear to us … Each method supports, encourages, or enforces a specific way of seeing the world. A phenomenology, a way of seeing the world, is embodied in a language: the language is adapted to express what we see; and our seeing is conditioned by the concepts familiar from our language… Most methods for solving problems that fit the Simple IS frame rely on the technique of making a MODEL of the *Real World* and embodying that model in the system. In effect, the system becomes the simulation of the real world, and derives

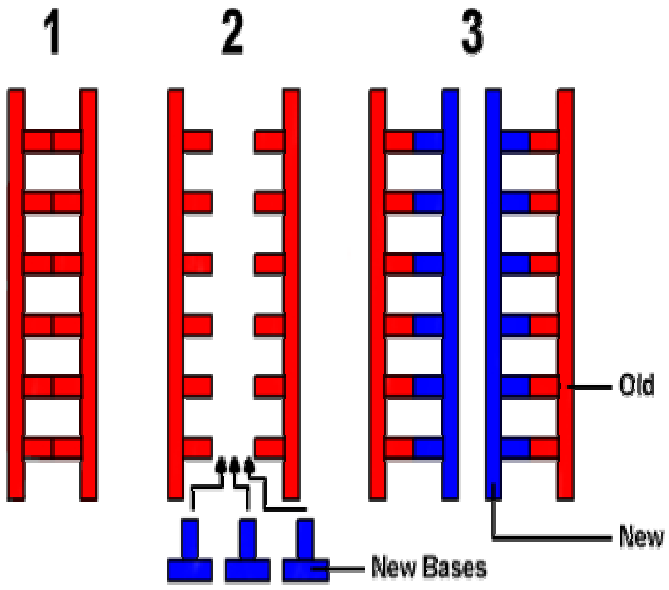

Figure 5: A very simplistic representation of the overall DNA replication

its information directly from its model, and only indirectly from the real world itself. This kind of modelling technique is at the core of JSD method ….” [39].

Although the starting point of the Jackson’s argument appears to be at the expense of the ontological approach, keeping everything in context of his broader discussion our understanding is that what he is practically talking about is the methodological validity of an approach in which the modeler models as the world unfolds to her in a discovery process and expands the system design thereof to more elaborate layers. The bearing for our argument is that probing into complex systems such as biological systems would unfold processes as the most prominent modules of conceptualization and therefore methodologically we start our architectural activity with process-oriented modeling.

Using Process-oriented Modeling as the core architectural style, one can model a dynamically changing system at the highest level – recognizable system processes and their interaction patterns – and preserve the consistency of the software system over the unfolding of evolutionary changes. The reality of complex systems is that human perception continues to have ambiguities about their details and inner structures, either from a scientific perspective or from a software design perspective, and these ambiguities lessen only in a discovery process. System initiation by means of modeling from a process level would enable us to focus on unambiguous parts for defining the system as a whole and preserving its consistency over the evolutionary path. In this paper we will present the details of this proposition through Process-oriented modeling of DNA replication.

DNA replication is one of many processes occurring in a cell. If one decides to model a human cell, the first question would likely be where to start from. As the basic unit of life and the building block of the human body, a cell in itself is an example of a complex system. Each cell can contain hundreds to thousands of energy production factories called mitochondria. Cells have complex membrane systems that function to produce necessary materials and package and ship them out if necessary. There are cell components that are equipped to degrade and dispose of waste within the cell. It is therefore clear that aside from DNA replication, there are many other processes occurring within the cell. The DNA replication process in itself has several subprocesses that are described further below, and is also interrelated with other cell processes. Thus, a structure like this requires a flexible system design method in which each process can be randomly taken, designed, tested with regards to some operational criteria and then later connected to other processes in order for the system to grow in an open-ended manner.

Following our stated requirement of “the capability of applying high-level architectural choices without a need for conventional coding”, we studied a number of implementation environments to determine the best pathway for attaining our objectives. As expected, we realized that there is no one single platform or implementation environment that can respond to all our needs in the design of evolvable systems. As a result we need to carefully combine different environments to create an equivalent of a manufacturing assembly line for generic evolvable systems as we defined. For this first phase and in realization of our Process-oriented modeling, we chose IBM WebSphere Business Modeler as a suitable implementation environment. This platform provides some of the necessary facilities for designing process models [40], although a good amount of fine-granularity manipulation tools are still missing in version 6.0.2.

Regardless of the platform used for implementation, a *Process Model* abstracts the dynamic nature, operations or activities of an organization of any type, and includes *Tasks*, *Resources*, *Services*, *Repositories* and *Processes.* In this paper we present the application of our perspective on process-oriented modeling to a high-level architectural modeling of DNA replication using a framework that can be geared towards two of our principles, process-oriented modeling and expression of high-level architectural choices.

A Process Model for our purposes is an abstraction of an evolvable system process’ real-time dynamics and contains individual task units and their interaction control patterns. The control patterns specify the conditions that initiate certain behavior. Resources are part of the Process Model and indicate instrumental or consumable entities that are necessary for the execution of the process or improving its performance.

A Process Model of a biological system which is equivalent to our Problem Model [1] can act as a foundation for understanding the intricacies of such extremely complex multidimensional systems by registering and exposing their inner interaction patterns in a quantitative, measurable and repeatable manner. Biological systems and their physiology are marked by webs of default processes, yet at the same time each process might have alternate processes dynamically unfolding due to diversification of environmental conditions or disease circumstances. Modeling the main processes and alternate processes along with their trigger

conditions and causes creates an unprecedented level of control over the manipulation of the rule-based nature of such systems as well as detection of the exception cases, for subsequent discovery of the reasons underlying them. Furthermore, such a manipulable Problem Model serves as the base model for defining the subsequent Requirements Model as defined in the Integrated Triple Sequence Model [1] that for such systems would be expressions of the process-confined desired changes that might be necessary for therapeutic or preventive operations. Last but not least, the Requirements Model leads to a Specification Model [1] that contains accurately manageable and quantitatively controllable therapeutic solutions. Such a model in its theoretically ultimate form can integrate monitoring devices that can attach to a patient and interact with their counterpart processes in the model.

4. SYSTEM IMPLEMENTATION

In this section we present the application of Process-oriented Modeling to the DNA replication mechanism. The application of this method to the DNA replication mechanism is meant to be a partial proof of concept (within the scope of Problem Model phase) and an elaboration of the method in practice. Although we are focusing on one specific process which has several subprocesses, the idea is not to demonstrate how the DNA replication process has evolved in the nature. Our attempt is to present that in a context where numerous processes are in operation with some known or unknown forms of organismic relationships, Process-oriented Modeling as we have defined theoretically and applied practically would make it possible for complex system designers to tackle the issue of complexity in a manageable framework, discover the pieces of the system in a random manner, and design discrete modules of a functional computer model at different levels of abstraction. These modules can then be related to each other in a hierarchical or horizontal manner as dictated by the nature of the system in question. Looking from a different perspective, two scientists can run their research independently at two different parts of the world and even in two different areas of expertise. Suppose a scientist has discovered a pattern of gene mutation under certain circumstances. Unaware of this research, another scientist has a new discovery on a specific chemical reaction in certain circumstances similar to the one in the first research. By using Process-oriented Modeling as a response to Evolvable Systems' design, these two sets of results can easily be integrated in a Process-oriented computer model that can still be open for some yet to be discovered organismic connections. Although this model might be incomplete, it can still function based on the current information. Furthermore, quantification of the model elements can independently help the advancement of the research on both sides of the proverbial pond.

In the Integrated Triple Sequence Model [1] we considered the formation of the Problem Model as the first step in the Integrated Triple Sequence Model. The Problem Model was defined as a model of the concerned area of the real world without any insertion of solutions or requirements, and as an entity belonging to the problem domain. Consequently, the Problem Model is an exact simulation of the operational model of the real world entities involved. The Problem Model would later be used as the foundation for Requirements Model. The Requirements Model as described in the Integrated Triple Sequence Model [1] reflects the amendments that the stakeholders wish to see in the systems under question. The Requirements Models for the biological systems might reflect amendments such as details of therapeutic, curing or just preventive objectives.

This model is a Problem Model as described above. There are two logical initiation points contained in this stage. One is the methodological initiating point or the answer to the question of how should we start methodologically. As we proposed in this paper, Process-oriented modeling would be the first step for the design of any evolvable system or subsystem. The second is a random initiation point, which is the selection of a random part of an evolvable system for design and implementation activity. Although we started from DNA in this case, the random initiation point could be any entity, like the cell nucleus (the container of the DNA) or the cell itself. That is because, as we argued, in the case of biological systems, for all practical purposes, there is almost no concrete base-line for the system to start the design thereof. The system, basically, at each random point is exposed to expansion at both macro-direction and micro-direction. Entities or components are intrinsically related to each other and are affected by each other both hierarchically and on a peer-to-peer basis. Any object that might be selected as the starting point for modeling, most possibly is composed of other micro-objects who have a determinant role in the behavior of that object. Unless one decides to start from the atomic level, and then one will realize that probably the String Theory will challenge her in determining the object behavior even at that level. Of course you could never start modeling the biological systems from the atomic or molecular level because you will be entering yourself into an impossible proposition in terms of ever being able to deliver one version of the system.

It is important to keep in mind that the Process-oriented Modeling defined in this paper covers only a partial scope of our design landscape, as at this level it is confined to the system activities performed at the Problem Model [1] phase. The Problem Model phase is responsible for abstracting the world-outside-computer of an evolvable system domain and representing it as an operational computer model. This model will be followed up in future work by its corresponding Requirements Model and then Specification Model [1] which will be elaborated on their own rights. In this case we chose our evolvable system domain to be bio-systems, which is analog and vague in nature, and our imperfect understanding of that needs to undergo a great transformation before it can be represented as a manipulable computer model. Our specific assertion in this paper is that this transformation from an analog, vague, complex multidimensional phenomenon in nature to an operational discrete computer model can only be achieved by Process-oriented Analysis and Modeling as we have defined it, as opposed to the conventional Object-oriented method for building systems. There are two main reasons for this:

1. There is no recognized baseline point for the system to start the system design from, as we explained, and practically speaking, there are no absolute base-objects whose behavior is independent from any other entity.

2. The essential relativity of the knowledge we have with regards to this complex bio-systems domain, and the persistent discoveries that keep our understanding and information constantly changing, create a fundamentally volatile computer system design context that should be capable of incessantly changing at all levels without much of a bearing on other parts of the system.

In continuation of this research, we will extend Process-oriented Analysis and Modeling at the Problem Model level – which is the scope of this paper – to a "Process Class" entity at the Specification Model level. We define a process class as an abstraction of one modulated behavior of a system represented by an atomic process which can reference one or more actual entities.

In choosing DNA replication, in fact, we took just the first step in our evolvable system design which is "the identification of the processes as the most prominent high-level expressions of complex systems". Identification of processes can conceptually be compared with that of use cases. A use case is a case of use of the system as it happens in the non-computer world. Detailing a use case in the context of Object-oriented programming has an ultimate goal of discovering the objects involved in the structure of the system. However, looking carefully, the concept of a use case comes from an environment of business applications, like the use case of a customer placing an order through an online catalog, or using the university registration system by a student to register for courses. Although detailing such scenarios will expose certain objects needed for implementation, writing similar use cases for biological systems is impractical and will not expose underlying micro-objects that are involved in the structure of such systems. In a complex biological system like the human body, as a case of an evolvable system as we discussed, there might be hundreds of micro-processes that are not exposed in a way to yield themselves to use cases modeling. In contrast we use Process-oriented Modeling not only to look at a complex system as a set of interrelated processes, but to effectively utilize an identified process to discover and modulate the *process elements* in the architectural expression of the system. A *process element* is a distinct, self-contained unit of a process model [40] such as a task, a processed item (called a business item in business applications), a resource, or a connection. Figure 6 shows the top level DNA replication *process model* designed in IBM WebSphere Business Modeler Advanced Edition. A *process model* is an abstraction of a real-time system process that in this case includes three local *tasks*, a global *repository* and one local *subprocess*. As in the non-computer world, a process might consist of any number of subprocesses, and itself might be recognized as a subprocess of another process. A *task* is the basic building block of a process model and is an abstraction of an atomic activity performed in a process. Conversely every process should eventually boil down to a set of underlying tasks with explicitly defined relationships. There is a concept of *reuse* associated with the *task*. A task defined in a parent process might be reused in child processes, or a task might be defined as a global element and reused in many processes. The process model specifies the input and output for each task as well as resources required for completion of the task. For complex sets of inputs, we will define input criteria. In the multi-input scenarios, different subsets of the input that can initiate a task, element or process constitute discrete sets of input criterion. Input criteria create a great deal of control on the process design. For example, we might define input criteria constraints [41] on certain sets of input so that if the constraints are not satisfied, the input criterion will not cause the element or process to run. On the other hand, we might define element preconditions so that if they are not satisfied, the element will not run but will generate exceptions. The first task in the DNA replication process (Figure 6) is "Get Replication Origin". The *replication origin* or *origin of replication* is a specific DNA sub-code which is recognized as the starting point of the replication process. There are many *replication origins* on each strand of DNA. Thus, when replication starts, many semi-parallel replication processes start, each from a specific origin of replication on the DNA strand. However, in Figure 6 we are focusing on only one of these processes starting from one *replication origin*. Therefore this first task is responsible for recognizing the specific sub-code on the DNA strand that signals a *replication origin* code. The second task is "Unwind Helix" which is responsible for unwinding the two strands of DNA. In its default state, a DNA molecule is in the shape of a double-stranded helix (Figure 7).

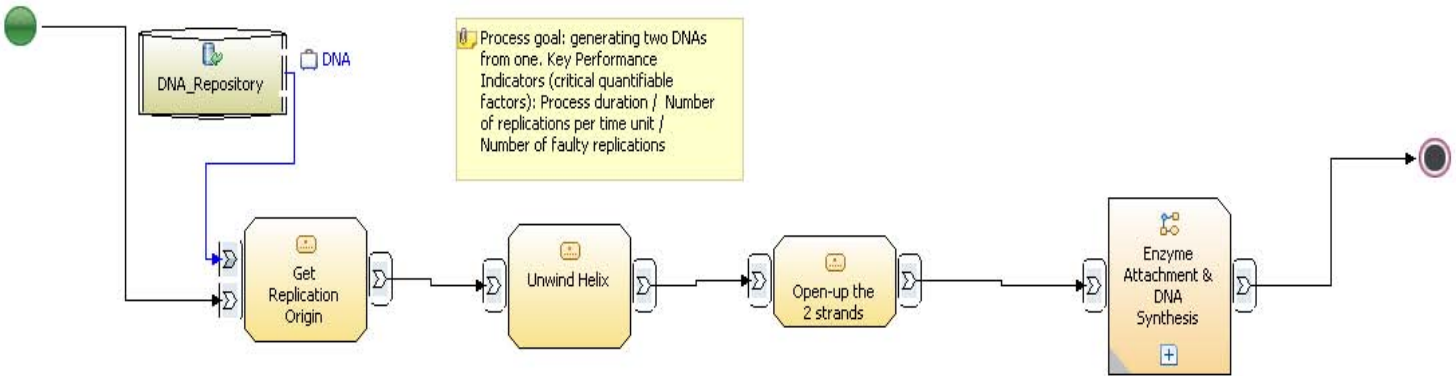

Figure 6: Top level DNA replication Process Model

Figure 7: DNA **double helix**

For the replication process to start, it is necessary that at a replication origin, the two strands of the DNA first unwind and then separate to allow the DNA polymerase enzymes access to the inside of each strand. Therefore the next task (Figure 6) is opening up the two strands so that the internal bases are exposed (Figure 8) for enzyme attachment and synthesis process advancement.

The last unit of operation in Figure 6 is itself another complex multi-level process which appears as a subprocess here called "Enzyme Attachment & DNA Synthesis". The top-level process duly shows the *start* and the *end* of the process delimiting this modulated unit of abstraction which could later appear as contained in a larger entity. While the process is under focus for the design of its ingredient modules, we are forced to think about the inputs and outputs to the tasks and subprocesses and their possible combinations and exceptions, as well as the transformation of the process model elements. This design process would reveal, for the purposes of architectural configuration, the involved *model elements* like the DNA, RNA, enzyme and base. However, to make the model more consistent and to have it reuse-oriented, we do not design the elements directly but we start with the templates for most model elements. The partial list of templates at this top level includes BaseTemplate, DNATemplate, EnzymeShapeTemplate, EnzymeTemplate, PhosphateTemplate and SugarTemplate as the ingredients of DNA (Figure 9).

We also discover the *resources* and design them normally in a three-step iteration by designing *Resource definition template*, *Resource definition*, and finally the *resource instance*. A resource is an entity that is instrumental in the realization of a task or process. Resources might be consumable or nonperishable. Creating *Resource definition templates* considerably improves the consistency across the system for many resources that are essentially similar. Such resource definitions inherit the essential properties from the corresponding template, but they might represent additional attributes. Resources are then defined as instances of their corresponding resource definitions. While analyzing the subprocess "Enzyme Attachment & DNA Synthesis" (Figure 6), we came across the design of an enzyme called RNA primase. Since enzymes act as catalysts in the formation of processes, by definition we can model them as resources. However, in an evolvable system design we have to put it in the framework of the larger picture. Although enzymes are operationally used as resources, they are manufactured by the body. Therefore in future software iterations we need to incorporate the enzyme manufacturing process into the system. Hence we do not categorize it under the resources at this phase to prevent further complications. Since there are many different enzymes in the body we need to start with an *element template* called EnzymeTemplate. According to the Lock-and-Key theory of enzyme specificity, the shape of an enzyme is the most important factor contributing to enzyme function [48]. Therefore we start with EnzymeShapeTemplate to be a contributing complex type toward the definition of EnzymeTemplate which in turn will be the parent template for Enzyme *definition* (Figure 9). The "RNA primase" as an *element* then inherits the EnzymeTemplate *definition*. "RNA primase" is the enzyme that attaches to DNA to

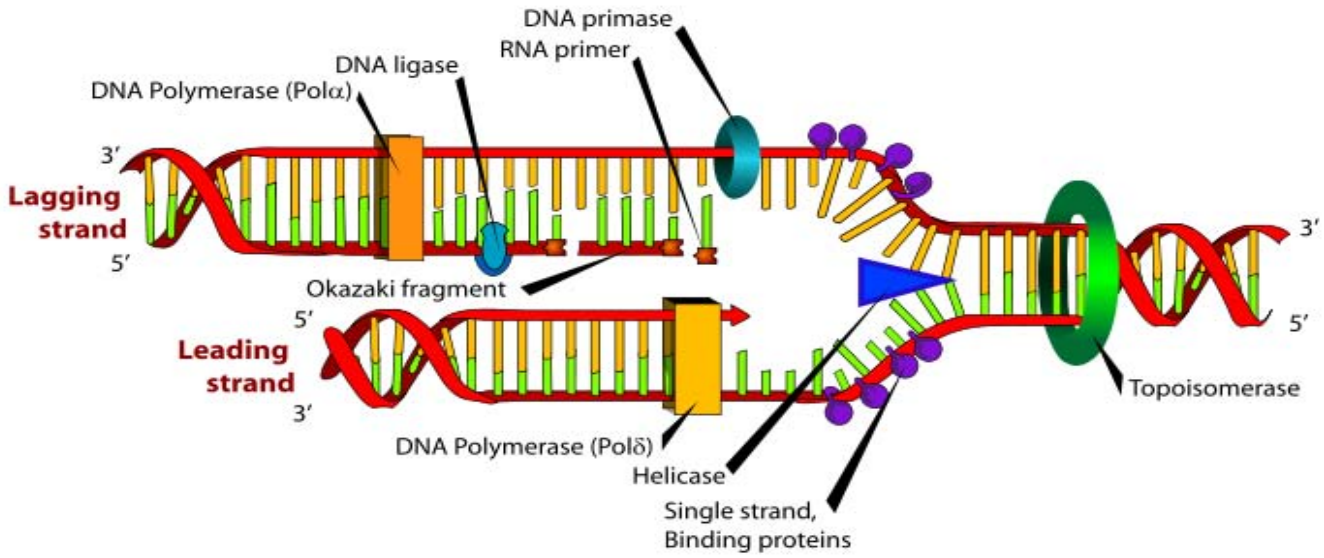

Figure 8: Exposure of the internal bases for enzyme attachment

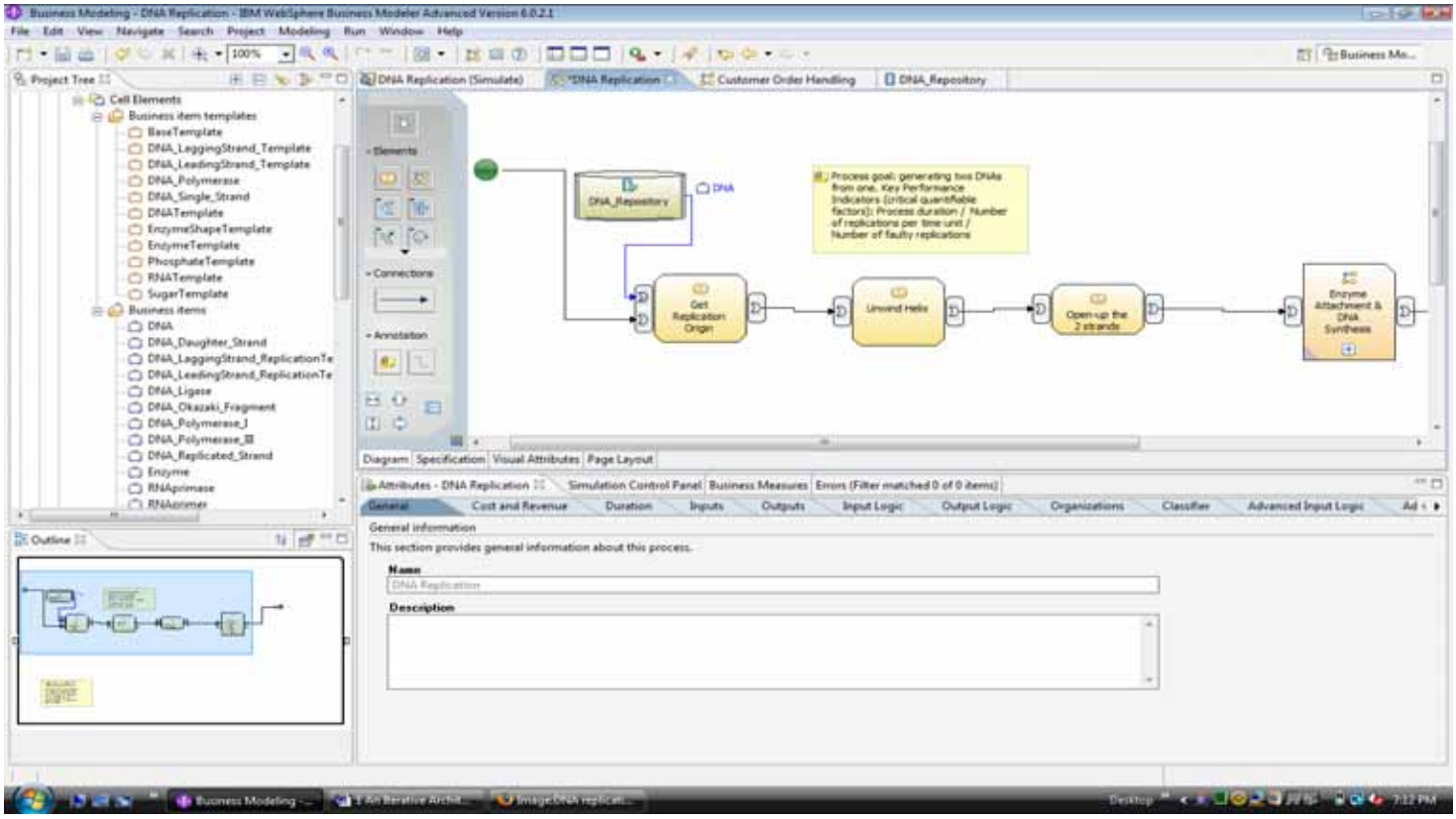

Figure 9: A partial list of Model Element Templates and Model Elements shown in the left column

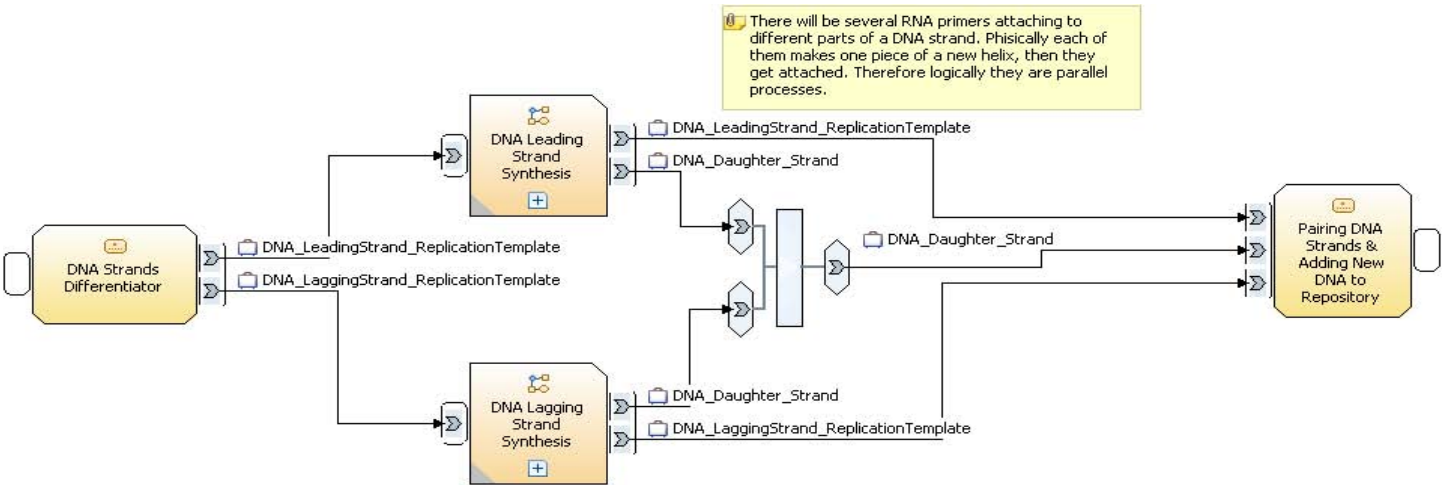

Figure 10: “DNA Leading Strand Synthesis” and “DNA Lagging Strand Synthesis” parallel subprocesses

create a small piece of RNA called an RNA primer, which is instrumental in DNA replication. In the future, for more complex processes, we might extend the *resource* concept by defining roles. Roles represent portable characteristics for resources and can be easily attached or detached with regards to dynamic resource operations.

The “Enzyme Attachment & DNA Synthesis” subprocess (Figure 6) consists of two other subprocesses, “DNA Leading Strand Synthesis” and “DNA Lagging Strand Synthesis”, operating as two parallel processes at each DNA fork formed on the DNA strand (Figure 10). In this paper we focus only on one single DNA replication fork.

Each of the above subprocesses operates on one strand of a single DNA and they move physically at opposite directions (Figure 11).

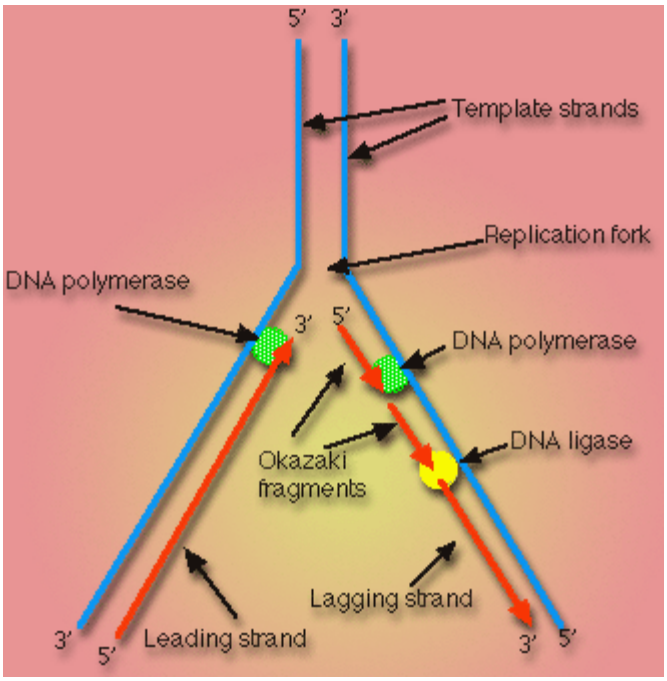

Figure 11: Anti-parallel movement of replication process

The initiation of these processes needs an introductory process that we have called "DNA Strands Differentiator." Through this subprocess, each DNA strand is logically marked as either a leading template or a lagging template and sent to their corresponding subprocesses. In the overall operation, the template strands for both the DNA leading and lagging strands are in fact global parameters. Many operations are performed with reference to either of those two template strands at different times, exactly the way conventional global parameters are treated. Furthermore, numerous operations are performed on either of those two strands at the same time in a parallel or semi-parallel manner. By semi-parallel we mean processes that start with just a small time-lag and operate practically in parallel on the same strand. In such a situation the global parameter is not treated in the conventional manner. To keep track of the operations as they propagate in real-time, what we really are interested in are dynamic pointers to certain areas of leading and lagging template strands. So if we pass the leading or lagging template strand to a process or task, we are in fact passing a specific pointer on the strand so that the intended operations can resume with reference to the specific coding of that point. This would create a very complex situation because we have one single global parameter that is subject to many parallel and serial operations that are in one way or the other related to each other but in terms of modeling they are not connected tasks or processes. In other words, we cannot pass the global parameter from one task to the other as conventionally done in such situations. To remedy this situation we create repositories. A repository is a storage area for one single type – either simple or complex type – of information created in a process. We normally name a repository by the element contained in it. The DNA repository would be a global repository to keep it available for all different tasks and subprocesses involved. Inside its structure lies the two lagging and leading strands each having their structure while associated to each other. In the replication process, each strand will be subject to different process-handlings, yet there will be some kind of synchronicity between the two sets of processes running on each strand. In our model the DNA_Repository is under the Repositories Catalog which is a sub-section of the "Process Catalogs".

The outcome of the "DNA Leading Strand Synthesis" is the original DNA leading strand template (parental strand now separated from its original bonds) plus a new DNA replicated strand (daughter strand) complementary to it. Likewise, the outcome of the "DNA Lagging Strand Synthesis" is the original DNA lagging strand template plus a new DNA replicated strand complementary to it. (Figures 10 and 12)

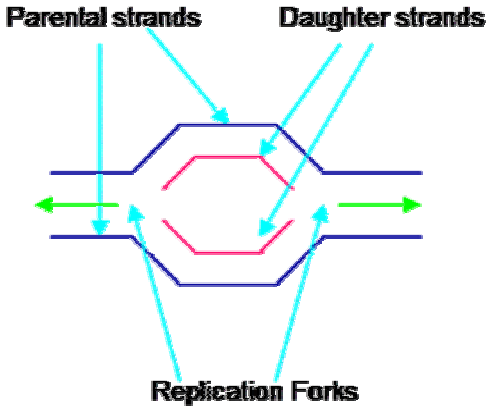

Figure 12: Simple representation of
Parental Strands and Daughter Strands

We direct the newly manufactured strands into a logical *join* (Figure 10). This software logical *join* is a pre-consolidation mechanism that operates on synchronizing the inputs by waiting to receive the corresponding input on all its incoming branches then sends all the inputs to the next task, "Pairing DNA Strands & Adding New DNA to Repository" (Figures 10 and 13).

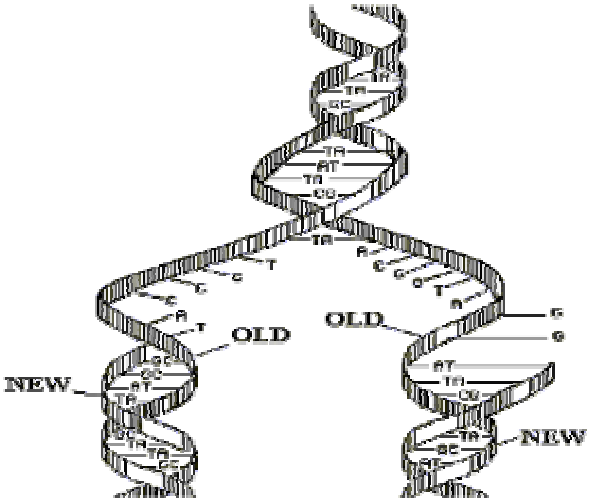

Figure 13: Pairing DNA Strands

So far we have only explained the top-level process as depicted in Figure 10. Now let's have a look inside the "DNA Leading Strand Synthesis" process. From a biological point of view, at each DNA fork, an enzyme attaches to the DNA leading strand template at the start of the fork and creates a small piece of RNA called "RNA Primer." This is represented by the "RNA Primer Synthesis" *task* in our model (Figure 14). Then, starting from the endpoint of the RNA primer, a new DNA strand (daughter strand) is synthesized complementary to the leading strand template, until it reaches the next RNA primer that has been synthesized at another replication origin. In this way the output of the "RNA Primer Synthesis" task is a physical piece of RNA plus a pointer to the starting point of the replication fork on the DNA leading strand template. These two are passed to the "DNA Strand Synthesis" task (Figure 14) as input where the DNA manufactured strand or daughter strand is actually coded and generated.

The "DNA Lagging Strand Synthesis" subprocess (Figure 10) follows a completely different logic (Figure 15). A DNA polymerase I molecule which is shown as a green circle on the right hand strand in Figure 15, attaches to the strand which would be the lagging strand template, and generates replicated pieces that are called "Okazaki fragments." It should be noted, although the red arrows in the figure point continuously downward, the first fragment generated in the scope of this figure is, contrary to the visual impression, the lowest piece in the figure, then the one above it and finally the top fragment arrow. The direction of the arrows only indicates the direction of elongation. The reason for this pattern is that as the fork opens up, the lagging strand that moves opposite to the direction of the opening would only get a chance to create one small piece as big as the opened up physical space allows. This is while the leading strand, that elongates in the same direction that the fork moves, can continuously grow as the fork moves along and so is demonstrated as a continuous red arrow on the left side of Figure 15. After the Okazaki fragments are generated, the enzyme DNA Ligase (shown in purple) stitches them together making a continuous replicated strand of DNA. To model this logic, we created a While Loop that continues to generate Okazaki fragments until the DNA lagging strand is complete.

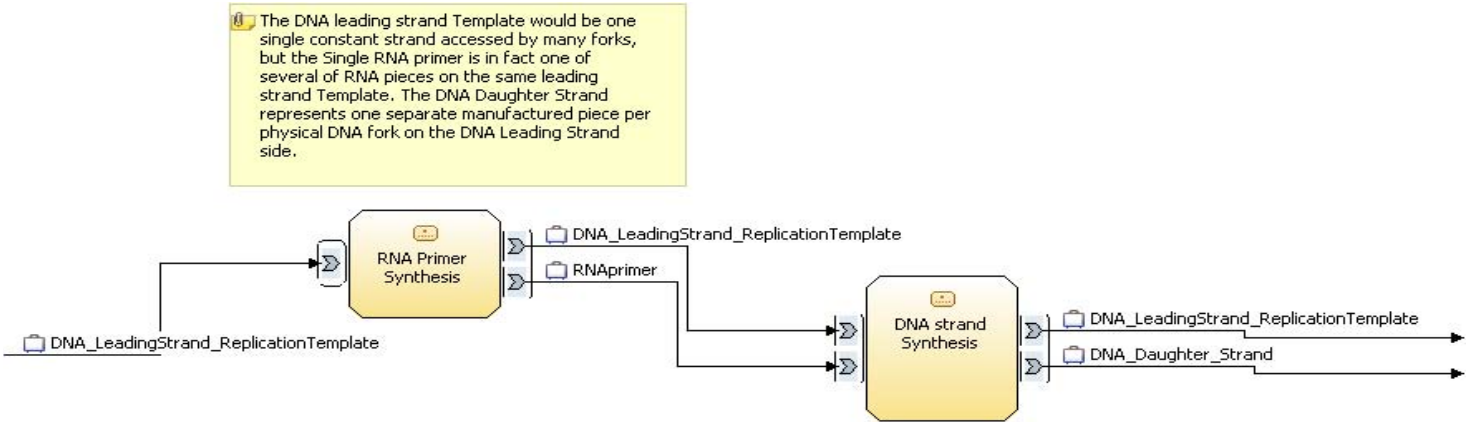

Figure 14: Inside DNA Leading Strand Synthesis sub-process

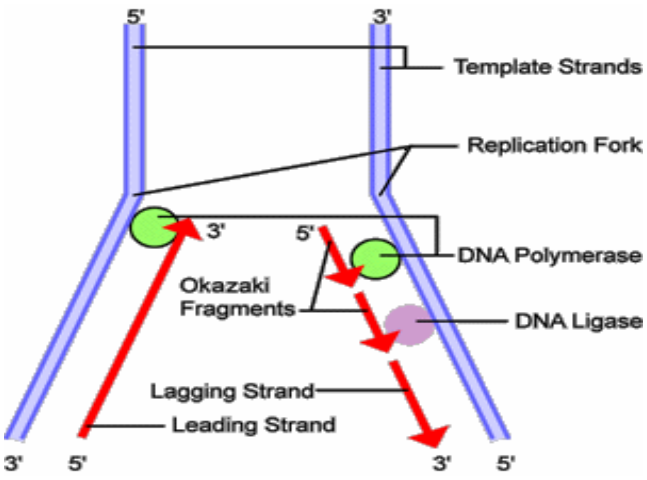

Figure 15: Different replication patterns of leading strand and lagging strand

However, in the IBM Business Modeler that we are using, the Expression Builder that sets up a loop condition can only access a local repository. Therefore it is necessary that we create a local repository only for the DNA fork segment that the local loop operates on (Figure 16).

This repository holds a specific fork segment and operates as the condition setter for the While Loop we created. This fork segment is a copy of a specific fork contained in the global DNA_Repository and is used as a logical operator for the process. There is a second loop in this process which is responsible for stitching the Okazaki fragments as long as they exist. After both loop operations are completed, the content of the repository is in fact a DNA segment which will replace a part of the lagging strand template on the global "DNA_Repository". Hence this value should get uploaded to the global "DNA_Repository" to replace one specific piece of the parental strand. However, this uploading

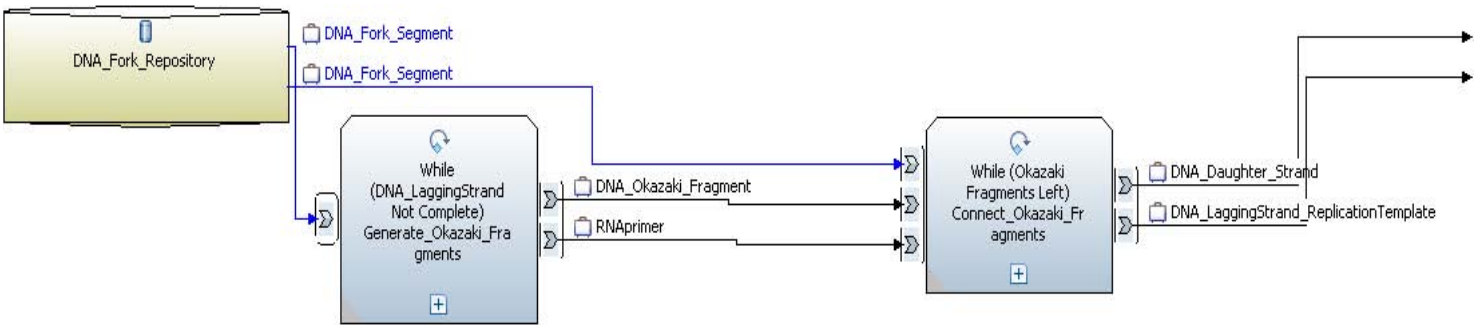

Figure 16: Inside DNA Leading Strand Synthesis sub-process

needs to be handled in a synchronized manner through the logical *join* we discussed in the parent process, which gets passed to the "Pairing DNA Strands & Adding New DNA to Repository" task (Figure 10).

4.1. QUANTIFIED CONTROLS AND SIMULATIONS:

One of the by-products of designing Evolvable Systems by means of Process-oriented Modeling is the capability of advance implementation and testing of the model through defining and assigning quantification measures and running them in the form of simulations even before the system is fully coded. This gives the designer(s) a chance to modify the system before full implementation given the goals and requirements defined, and even fine-tune the ingredients of the system to as detailed specifications as they wish. This subsection should be considered as an added benefit, and not the sole conclusion of our Process-oriented Modeling method. The thrust of this paper is presenting a method in response to the specific problem of the design of Evolvable Systems as defined.

Each Quantified Control could be viewed as an individual measurement dealing with either one entity or a collective pattern of measurements covering several entities. Furthermore, each of these measurements can be associated with either one run-through of a process or associated with several run-time instances of processes. In the latter, the data collected for the measurements would cover a set of run-time instances. The mechanisms of the *Control Measurements* are built into the system at the process-level design and can utilize factual or hypothetical data for testing consistency of the results or verification of the running system utility and goals even before the actual coding of the system starts. In our case we chose two Quantified Controls for our DNA replication process, *energy consumption* and *duration*. The quantification results can be achieved in the WebSphere Business Modeler environment through running simulations. However, for the best results and for more complex handling, it is possible to combine the services of the WebSphere Business Modeler and the WebSphere Business Monitor, although we do not present such combination in this paper. As explained before, in an environment of almost never-ending disclosure of micro-level information about Evolvable Systems, it is critical to have system-level mechanisms that accommodate continuous defining, specifying and analyzing quantifiable aspects of the processes in a growing but seamlessly integrated software system. Our "Quantified Controls and Simulations" is an attempt in that direction. Conceptually, we need to start with identifying observation points as the entities or points that encompass critical information of the system. For each observation point which will be in fact monitored in real-time, we specify the information of interest that we want monitored by utilizing an association of the *business measures* with the concerned *entity*. The *business measures* can be associated with the whole process or with the constituent entities of the process. Business measures include *Key Performance Indicators (KPI)* and *Metrics*. A *KPI* is a measurement of concern with reference to the run-time state(s) of a process or some overall system goals. Therefore, in addition to the optimal target point for each criterion, we can also designate allowable or critical ranges so that the Quantified Controls would show the exact standing of the system performance in that area. The calculation of the *KPI* relies on the definition of some *metrics* that are used either individually or as a set. Obviously, a metric is a data container consisting of some measurement criterion of the system operations. If the metric application returns the result of a single run of a process, it would be categorized as an *instance metric*, while if the metric application returns the result of multiple runs of a process in some kind of a mathematical formulation, it would be an *aggregate metric*. For our purposes we define two *KPIs* and one *Instance Metric*. The numbers we use in these simulations are absolutely arbitrary and are used just to show how the mechanisms of Quantified Controls and Simulations can be designed, integrated into the system and utilized.

The first *Key Performance Indicators (KPI)* is "DNA Replication per 24 Hour". The description of a KPI should include exact definitions and requirements for implementation as well as instructions on how to complete and calculate the monitored measure in the WebSphere Business Monitor Development Toolkit. *DNA Replication per 24 Hour* is the total number of DNA replications per cell, in each 24 hours. This Number should be taken from the following several sources at different stages of advancement:

1. Initially, it will be a general fixed number that is indicated by the science for normal human being.

2. It will be a number based on better information sets. This should be taken from specific information based on the categories such as age, sex, and other health or ethnic information about a specific person.

3. The number would be obtained based on accurate testing or

sensory devices attached to the specific patient.

For now we assume one replication is done per 3 hours in a cell, so we put it as the base of our target value and as a normal goal. The target value then will be eight (8) with an indicated type of *Number*. Since we indicated a target or optimal value, we also need to indicate a range value which could be defined as a percentage of the target value or as an actual range of numbers. In this case we defined a range of four (4) to ten (10). A range can be defined as a *safe range* or a *warning range*, and consequently require special attention or action. The type of the *alert* or *action* can be specifically defined and integrated. We can also specify a time period for monitoring depending on circumstances. In addition, it is also possible to select *dimension(s)* across which to calculate a KPI. Our choice of dimension here is "DNA Replication Speed".

The second KPI we defined is "DNA Replication Duration". This is the time duration for one replication process to run through real-time. One way of calculating it would be by subtracting the time stamp of first input reaching the first task from the time stamp of the last output leaving the last task. Initially we assume setting the target value based on optimal numbers indicated by science for a healthy person which we assumed at three hours, with a specified range of two to four hours. The dimension across which to calculate this KPI would be *DNA Replication Duration*.

The one metric we defined refers to a task duration measurement and is called "Unwind Helix Duration". In this case we defined it as an *instance metric*. The unwind helix duration will be calculated by subtracting the input time-stamp from the output time-stamp. The default value given is three minutes, although we could define it in any other time units consistent with other duration measurements. We can get the accuracy of milliseconds on all time measurements.

The process simulation provides an opportunity to observe and calculate all aspects of a process at the finest granularity which is the task level if required. In this process, Key Performance Indicators can be set along with observation points to analyze and design the internals of the dynamic architecture of the system. Simulating a designed system and checking it against a set of important criteria before coding the system is not only highly desirable but critical for complex evolvable systems. In fact, since an evolvable system is always subject to changes at the architectural level, it is a necessity to test the system at the same level for consistency purposes as well as ascertaining that the system-level goals are properly met. The Process-oriented modeling provides the required substrate for this type of architectural and design level testing and simulation, as the conceptual and software units of interaction are processes and their interrelations, as opposed to detail objects and classes. Fortunately, WebSphere Business Modeler presents the capability of running simulations. Unfortunately for our purposes, the environment is written for Business purposes as it is named such, not for our specialized design. Therefore we have to extend the platform to utilize it for our research intentions. We hope one of the side-benefits of this research would come to be encouraging the likes of IBM by giving them clear substance and direction in writing off-the-shelf software that could be used directly for such

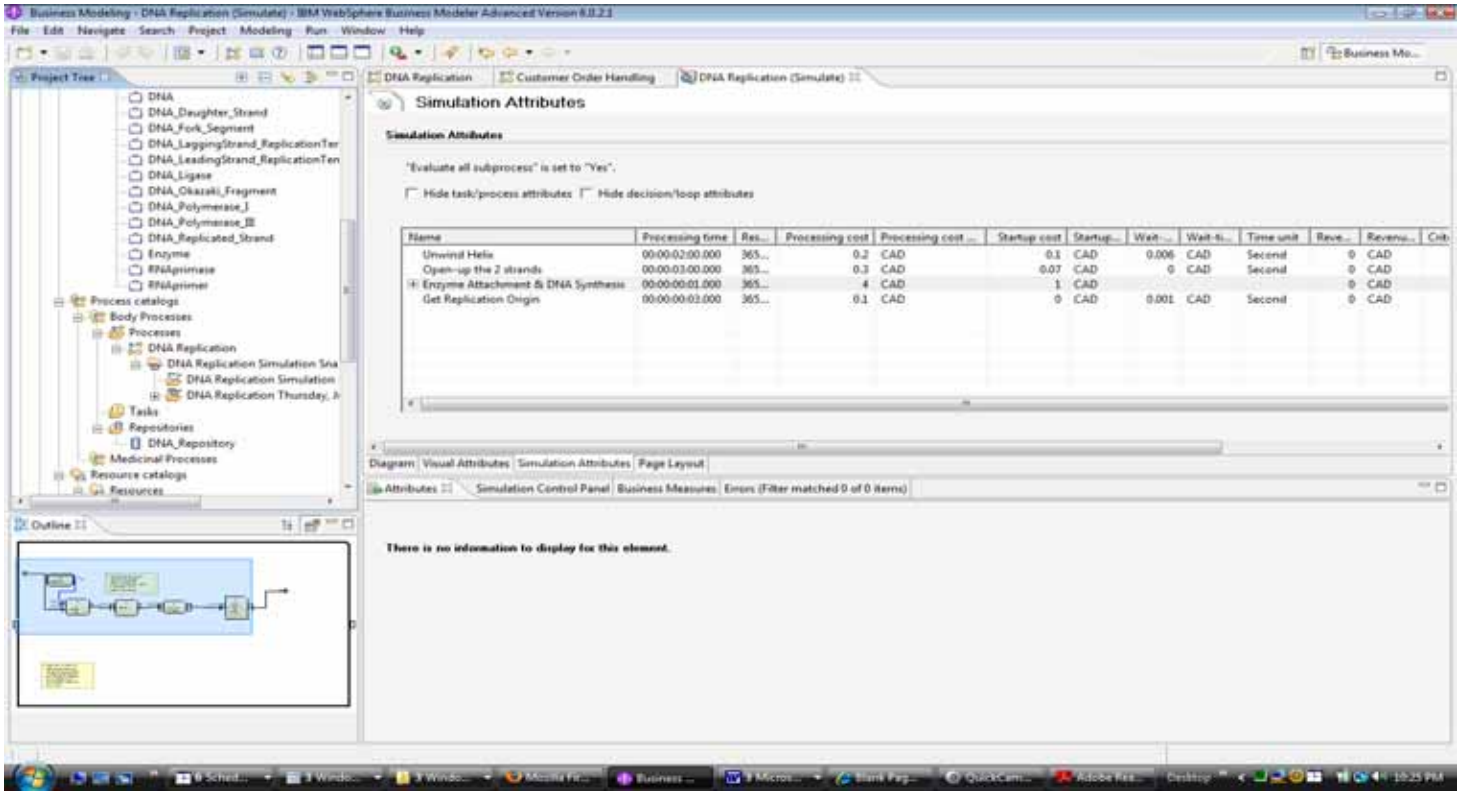

Figure 17: The result of a simulation based on two Quantified Controls

Bio-systems. One of the dimensions of our process observation was *energy used* at the micro-levels of operations in the DNA replication model. Although we could always define and implement structures for our observation criteria, we found it more appealing to use the ready features of the platform. Therefore in an analogy between the energy spent for performing tasks and processes on one hand, and the cost incurred for business operations on the other, we chose to utilize the cost/currency capability of the platform for calculating the *energy used* concept. Although the energy unit at the cellular level is considered in ATP, for simplicity we assumed it in the well known calorie format and chose the symbol CAD (for our definition indicating Calories Disposed) in our simulation. In this way we assigned to each task or process some arbitrary units of calories consumed, as well as a time duration for each, and then ran the simulation. The brief results of the simulation are shown in the Figure 17 which shows the processing durations and energy consumed for each task, process and in aggregate. The simulation variables and values can be fine-tuned at the global, local or instance levels. Simulations provide great opportunity for fine-tuning the architectural-level and design-level concerns and modifications. Our Bio-system simulations deserve an independent paper and we consider it as a future work.

The specified Quantified Controls can be exported as *monitor models* to WebSphere Business Monitor Development Kit which is part of WebSphere Integration Developer. It is also necessary to specify the event collection mechanisms for control patterns of the monitor model for use in WebSphere Business Monitor.

5. CONCLUSION

In this paper we presented a methodological framework for the design of open evolvable systems. We placed our focus on two interrelated research questions:

1. What is an efficient methodology for designing a system where we do not know its boundaries?

2. How could we manage architecturally *consistent* design and development of an open-ended system that is continuously changing?

We laid out our perspective for the architectural scope of the evolvable systems to be *generic* systems in contrast to the mostly domain-oriented conventional approaches, and in this path we defined system requirements to clarify and emphasize the two concepts of *evolvability* and *open-ended design* capability.

On the premises of the complexities of *evolvability*, we presented our version of "Process-oriented Modeling" as a fundamental approach to provide for consistent *open-ended design* capability. We pondered on the choice of the substrate for carrying out our theoretical views and we came to the conclusion that biological systems are probably the most complex evolvable systems we can consider and we provided the reasoning behind it. We applied our Process-oriented Modeling approach to a simplified version of the DNA replication process as a proof of concept and elaboration of the method in practice. The purpose of this application is not to show evolution of the DNA process itself. The purpose is to demonstrate the inner working and flexibility of the Process-oriented approach in the design of Open Evolvable Systems. This application elaborates how to approach the issue of design and development in an extremely complex environment by conceptualizing and modularizing the system as a collection of dynamic processes, as opposed to a collection of hierarchical objects and classes. By the application of Process-oriented modeling to DNA replication mechanism, our purpose was to demonstrate how a process could be picked randomly out of hundreds or thousands of processes in an open complex system, then flexibly add newly discovered subprocesses or parent processes and yet preserve the integrity of the system. We also indicated that the use case modeling as a powerful method in the business world might not have any application to the biological systems' modeling as an instance of the Open Evolvable systems' modeling. Finally, we briefly discussed the dynamic aspects of the design process management by crystallization of the goals of the system through the idea of "Quantified Controls" and verifications of the purposes in the framework of Simulations even before coding the system.

Furthermore, we hope this methodology will prove to be of special value for scientific applications that are by default much more complex than business applications. Our choice of bio-systems for applying our method proved to be a valuable experience for us as it turned our attention to many intricacies of the stringent requirements of this domain. We hope future extensions of this research, which can include modeling malfunctioning physiological processes and development of Requirements Models for prevention or treatment through the use of Process-oriented Modeling, will open up grounds for a new line of medical applications and devices.

6. FUTURE WORK

The future work will focus on two parts. The first is extending and fine tuning the Process-oriented Modeling to go beyond the Problem Model phase and cover the rest of the design lifecycle as contended in the Integrated Triple Sequence Model [1]. Along this line the current proof of concept application model presented in this paper will inevitably be extended to new phases of Requirements modeling and specifications modeling which is the ultimate goal of the whole enterprise.

The second part is transforming the current architectural model into a code-implemented system [42] by devising a mostly automated roadmap. One of the conceptual requirements for the Open Evolvable Systems we laid down in this paper was the capability of applying high-level architectural choices without a need for conventional coding. The fact is no software package is written to serve as a comprehensive platform for our end-to-end purposes and we do not intend to code such a platform from scratch. In fact that would be a pre-coded solution that will undermine our evolvability assumptions. Therefore, our only choice is to carefully study the capabilities of the off-the-shelf packages as we have done so far, carry out the initiatives to extend them for our purposes, and stitch them together to serve our requirements and specialized design.

REFERENCES

[1] Bastani, B. (March 2007): A Requirements Analysis Framework for Open Systems Requirements Engineering. ACM SIGSOFT Software Engineering Notes, Volume 32, Issue 2.

[2] Center for Lifelong Learning & Design, University of Colorado, Boulder (November 2001): The Software Technology of the 21st Century. *ISFST2001 — International Symposium on Future Software Technology, ZhengZhou, China.*

[3] Budgen, D. (2003): Software Design. Pearson – Addison Wesley. ISBN: 0201722194.

[4] Ghezzi, C., et al. (2003): Fundamentals of Software Engineering. Prentice Hall. ISBN: 0133056996.

[5] National Institute of Standards and technology: http://www.nist.gov/dads/HTML/finiteStateMachine.html

[6] The 6th International Conference on Evolvable Systems, ICES 2005: http://147.83.49.249:8090/

[7] Arlow, J., et al. (2005): UML 2 and the Unified Process. Addison Wesley. ISBN: 0321321278.

[8] Fischer, G., E. Scharff (2000): Meta-Design: Design for Designers. *Proceedings of the conference on Designing interactive systems: processes, practices, methods, and techniques* pp. 396-405.

[9] Reilly, D., et al. (2002): An instrumentation and control-based approach for distributed application management and adaptation. *Proceedings of the first workshop on Self-healing systems, Charleston, South Carolina,* pp. 61 – 66.

[10] Fischer, G. (1998): Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments. Automated Software Engineering. 5(4): pp. 447-464.

[11] Fischer, G. (1998): Beyond 'Couch Potatoes': From Consumers to Designers. *In 1998 Asia-Pacific Computer and Human Interaction, APCHI'98, IEEE (Ed.).* IEEE Computer Society, pp. 2-9.

[12] Kramer, J., J. Magee (Nov. 1990): The Evolving Philosophers Problem: Dynamic Change Management. IEEE Trans. on Software Eng., 16 (1 l), pp. 1293-1306.

[13] Cheng, S., et al., (2002): Using Architectural Style as a basis for System Self-Repair. *The Working IEEE/IFIP Conference on Software Architecture, Montreal.*

[14] Dellarocas, C., et al. (1998): An architecture for constructing self-evolving software systems. *Proceedings of the third international workshop on Software architecture, Orlando, Florida, USA.*

[15] Clarke, L.A., L. J. Osterweil (2000): Continuous Self-Evaluation for the Self-Improvement of Software. *Proceedings of the First International Workshop on Self-Adaptive Software, (IWSAS2000), Oxford, UK.*

[16] Oreizy, P., et al. (1998): Architecture-Based Runtime Software Evolution. *Proceedings of the 20th Int'l Conference on Soft. Eng. (ICSE'98), Kyoto, Japan,* pp. 177-186.

[17] Subramanian, N., L. Chung (2002): Tool support for engineering adaptability into software architecture. *Proceedings of the international workshop on Principles of software evolution (IWPSE '02): Evolution patterns and models.*

[18] Nardi, B.A. (1993): A Small Matter of Programming. The MIT Press, Cambridge, MA.

[19] Hulse, C., et al. (1999): Reducing maintenance costs through the application of modern software architecture principles. *Proceedings of the 1999 annual ACM SIGAda international conference on Ada.* ACM SIGAda Ada Letters, Volume XIX, Issue 3.

[20] Laddaga, R. (2000): Active Software. *First International Workshop on Self-Adaptive Software (IWSAS2000), Oxford, UK,* Springer-Verlag.

[21] Dabrowski, C., K. Mills (2001): Analyzing Properties and Behavior of Service Discovery Protocols Using an Architecture-Based Approach. *Proceedings of Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia.*

[22] Sun Microsystems Inc. (2000): Jini Architecture Specification - v1.1, http://www.sun.com/jini/specs.

[23] Bauer, M., D. Dengler (1999): TrIAs: Trainable Information Assistants for Cooperative Problem Solving. *Proceedings of the Third International Conference on Autonomous Agents* (Agents'99).

[24] Shaw, M. (2001): The Coming-of-age of Software Architecture Research. *Proceedings of the 23rd international conference on Software engineering, Toronto, Canada*.

[25] Medvidovic, N., et al. (2002): Modeling Software Architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11, Issue 1.

[26] Robertson, P., et al. (2000): Self Adaptive Software. *First International Workshop on Self-Adapative Software, Oxford, UK, April 17-19, 2000,* Springer, New York, 2001.

[27] Garlan, D. (2000): Software architecture: a roadmap. *Proceedings of the conference on the future of Software engineering*. ACM Press, 2000.

[28] Ambriola, V., A. Kmiecik (2002): Software architectures:

Architectural transformation. *Proceedings of the 14th international conference on Software engineering and knowledge engineering*.

[29] Darimont, R., et al. (1997): GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. *Proceedings of the 19th International Conference on Software Engineering, Boston, MA.*, pp. 612 – 613. ISBN:0-89791-914-9.

[30] Maccari, A. (2002): Experiences in assessing product family software architecture for evolution. *Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida*, pp. 585 – 592. ISBN:1-58113-472-X.

[31] Sha, L., et al. (Feb. 1996): Evolving Dependable Real-time Systems. *Proceeding IEEE Aerospace Applications Conference. Aspen, CO.*, vol. 1, pp 335-346. ISBN: 0-7803-3196-6.

[32] Kramer, J., J. Magee (Nov. 1990): The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, vol. 16. no. 11.

[33] Peterson, J., et al. (July 1997): Principled Dynamic Code Improvement. Yale University Research Report YALEU/DCS/RR-1135. Department of Computer Science, Yale University.

[34] Lung, C., et al. (1997): An Approach to Software Architecture Analysis for Evolution and Reusability. *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research. IBM Centre for Advanced Studies Conference. Toronto, ON*.

[35] Popova, V., A. Sharpanskykh (2007): Process-Oriented Organization Modeling and Analysis. *Proceedings of 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, joint with ICEIS'07*, published by INSTICC Press.

[36] Chung, L., B. Nixon (1995): Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach. *International Conference on Software Engineering 1995, Seattle, Washington.*

[37] Ottjes, J., H. Veeke (June 2002): Prototyping in Process Oriented Modeling and Simulation. *Proceedings of the 16th European Simulation Multiconference (ESM 2002).* ISBN 90-77039-07-4.

[38] Belhajjame, K., et al. (2001): A flexible Workflow Model for Process-oriented Applications. *Proceedings of the Second International Conference on Web Information Systems Engineering, WISE'2001.*, Dec. 3-6, 2001. Volume 1, pp. 72 – 80.

[39] Jackson, M. (1995): Software Requirements & Specifications. New York, NY: The ACM Press.

[40] IBM WebSphere Business Modeler Library: http://www-306.ibm.com/software/integration/wbimodeler/library/

[41] Balzer, R. (Oct. 1996): Enforcing architectural constraints. *Second International Software Architecture Workshop (ISA W-2), San Francisco, CA.*

[42] Aldrich, J., et al. (May 2002): ArchJava: connecting software architecture to implementation. *Proceedings of the 24th international conference on Software engineering*.

[43] Ostwald, J., et al. (July 2003): Organic Perspectives of Knowledge Management. *I-KNOW'03 Workshop on (Virtual) Communities of Practice within Modern Organizations, Graz, Austria.*

[44] Edelson, E. (1999): Gregor Mendel and the Roots of Genetics. Oxford University Press. ISBN: 0195122267.

[45] Avery, O., et al. (Feb. 1944): Studies on the Chemical Nature of the Substance Inducing Transformation of Pneumococcal Types: Induction of Transformation by a Desoxyribonucleic Acid Fraction Isolated From Pneumococcus Type III. Journal of Experimental Medicine. Volume 79(2): pp. 137-158.

[46] Crick, F. (Aug. 1970): Central Dogma of Molecular Biology. Nature. Volume 227: pp. 561-563.

[47] Cairns, J., C. Davern (1967): The Mechanics of DNA Replication in Bacteria. Journal of Cellular Physiology. Volume 70(S1): pp. 65-76.

[48] Stenesh, J. (1998): Biochemistry. Springer. ISBN: 0306457334.

[49] Meng, A. (2000): On Evaluating Self-Adaptive Software. *Proceedings of the First International Workshop on Self-Adaptive Software, (IWSAS2000).*

[50] Badr, N., et al. (May 2002): A Conflict Resolution Control Architecture for Self-Adaptive Software. *Architecting Dependable Systems, WADS 2002 (ICSE 2002), Orlando, Florida.*

[51] Garlan, D., B. Schmerl (2002): Model-based Adaptation for Self-Healing Systems. *Proceedings of the first workshop on Self-healing systems, Charleston, South Carolina.* ISBN:1-58113-609-9.