Towards a Common Computational Synthesis Framework with Object-Process Methodology

Dov Dori[1,2] and Edward Crawley[2]

[1]Technion, Israel Institute of Technology, Haifa 32000, Israel

[2]Massachusetts Institute of Technology, Cambridge, MA, USA

dori@ie.technion.ac.il, crawley@mit.edu

Abstract

The field of computational synthesis is emerging is an interdisciplinary effort that encompasses applications in a variety of knowledge domains. Each domain has its own ontology, modeling methods, set of symbols, and syntax. The cross-cutting commonalities among the various implementation domains, which make computational synthesis a research domain in its own right, are articulated for the most part only in natural language. The computational synthesis community may greatly benefit from an agreement on a domain-independent paradigm and modeling methodology that are shared among the various fields of knowledge in which computational synthesis is or will be applied. This paper proposes Object-Process Methodology (OPM) as a unifying framework for modeling in a balanced and transparent way how the architecture, i.e., the combination of structure (objects) and behavior (processes) of the system being evolved fulfill the functional requirements (measured by fitness) for its survival in the environment. OPM specifies not only the evolvable life form as a final product, but also the universal computational synthesis metamodel – the model of the system that makes this evolution happen.

Introduction

As computational synthesis is emerging is an interdisciplinary effort that encompasses applications in a variety of knowledge domains, the need for a common underlying language is increasing. Each implementation domain that applies computational synthesis has its own ontology, modeling methods, set of symbols, and syntax. The cross-cutting commonalities among the various domains, which make computational synthesis a research domain in its own right, are articulated for the most part only in natural language. The computational synthesis community may greatly benefit from an agreement on a domain-independent paradigm and modeling methodology that are shared among the various fields of knowledge in which computational synthesis is or will be applied.

Acknowledging the potential benefits of a common methodology, this work examines the pros and cons of two prominent candidates for this task: Unified Modeling Language (UML) and Object-Process Methodology (OPM). OPM is arguably better fit for this purpose. Once a justified selection has been made, the work next focuses on top-down and bottom up modeling that would eventually lead to an agreed upon metamodel and analyze instances of evolving life forms in the various application domains.

Required Features of a Modeling Method

A universal methodology, which encompasses a unifying ontology and language for setting up a computational synthesis framework, can be accepted as a standard for computational synthesis applications. The methodology of choice, which can conveniently serve the various computational synthesis application domains, should feature a rather stringent set of requirements:

1. **Domain independence:** Creating a common meeting ground, the standard modeling framework to be adopted should enable domain-neutral specification of computational synthesis systems that researchers from disparate domains can relate to. Agreeing on such universal concepts, researchers and developers would be able to think in common generic terms rather than having to adapt their thinking to the set of concepts that are peculiar to the domain. Therefore, the approach should fit modeling systems in general, not just software systems. It should be capable of defining the elementary building blocks of the universe in a way that will be applicable to any domain conceivable for computational synthesis application.
2. **Modularity:** Since modularity is a key issue in computational synthesis (Lipson, Pollack, and Suh 2002), the methodology should also define how these building blocks link with each other to specify modules or functional units at varying scales, starting from the elements and going all the way to the entire life form, or even a society of such life forms, as a system.
3. **Structure-behavior balance:** The methodology should reflect in a balanced and transparent way how the architecture, i.e., the combination of structure (objects) and behavior (processes) of the system being evolved fulfils the functional requirements (measured by fitness) for its survival in the environment.
4. **Metamodeling capability:** The same methodology should be able to specify not only the evolvable life form

as a final product, but also the universal computational synthesis metamodel – the model of the system that makes this evolution happen.

Among other advantages, using a common set of terms and symbols across domains will provide a vehicle for direct comparisons of computational synthesis approaches and tools, thereby advancing the state-of-the-art in this field. In the next sections, we describe the two candidates, Unified Modeling Language and Object-Process Methodology and justify our selection.

Unified Modeling Language

UML is the Object Management Group (OMG) standard (Object Management Group 2000) object-oriented graphical language for modeling software and information systems. Since UML has evolved bottom-up from OO programming concepts, it lacks a system-theoretical ontological foundation (Soffer et al. 2001) encompassing observations about common features characterizing systems regardless of their domain. It defines a model of the information system to be developed that can be specified and examined using nine different views. Each view is represented using a different graphical diagram type with its own set of notation and meaning. The nine views are (1) Use-case diagram; (2) Class diagram; (3) Object diagram; Behavior diagrams, which include (4) State transition (Statecharts) diagram and (5) Activity diagram; Interaction diagrams, which include (6) Sequence diagram and (7) Collaboration diagram, and Implementation diagrams, which include (8) Component diagram and (9) Deployment diagram.

While UML has undoubtedly contributed to streamlining software engineering practices since its commercial standardization in 1997, the way it was conceived and adopted by OMG put it on a difficult track for becoming universally usable. As noted, UML agglomerates nine diagram types, also called views, or models, declared to be the unified standard. But such a declaration cannot replace unification of the concepts and symbol sets associated with the models, along with removal of the many redundant entities and overlapping notions (Dori 2002a). A major problem with UML is the size of its alphabet of more than 150 symbols that are distributed among the nine diagram types. No less disturbing is the number of its diagram types, which makes each change in one diagram trigger numerous required consistency checks among the various diagram types. One author has described UML's mix of notations from different approaches as yielding "a confused picture, which mixes design forces and so is not really productive" (Simmons 2001). Siau and Cau (2001) have found that UML is up to 11 times more complex than other OO methods. The associated model multiplicity problem (Peleg and Dori 2000) concerns the fact that not even one of the nine UML models clearly shows a unified, integrated view of the two most prominent and useful system aspects: structure and behavior. The UML user is expected to decide which type of diagram to use when and how to create an integrated mental picture of the system. The UML model reader is likewise expected to mentally integrate different views of the software system into a coherent whole.

Object-Process Methodology

Object-Process Methodology, abbreviated OPM, (Dori 2002b) is a holistic approach to the study and development of systems, which integrates the structural and behavioral aspects that systems exhibit into a single frame of reference. Most interesting and challenging systems are those in which structure and behavior are highly intertwined and hard to separate. Structure and behavior, the two major aspects co-exist in the same OPM model without highlighting one at the expense of suppressing the other. Due to this structure-behavior integration, OPM provides a generic, domain independent conceptual infrastructure for modeling complex systems, expressed in a bimodal way, graphically and textually, as explained below.

The *elements* of the OPM ontology are *entities* and *links*. Entities are of three types: *objects*, *process*es, and *states*. These are the basic building blocks of any system expressed in OPM. *Objects* are (physical or informatical) things that exist, while *processes* are things that transform objects. *Links* can be structural or procedural. *Structural links* express static relations between pairs of entities. Aggregation, generalization, characterization, and instantiation are the four fundamental structural relations. *Procedural links* connect entities (objects, processes, and states) to describe the behavior of a system.

The OPM model manifests the system's behavior in three major ways: (1) Transforming: processes can transform (generate, consume, or change the state of) objects; (2) Enabling: objects can enable processes without them being transformed; and (3) Triggering: objects can trigger events that (at least potentially, if some conditions are met) invoke processes. Accordingly, a procedural link can be a transformation link, an enabling link, or an event link.

A *transformation link* expresses object transformation, i.e., object consumption, generation, or state change. An *enabling link* expresses the need for a (possibly state-specified) object to be present, in order for the enabled process to oc-

cur. The enabled process does not transform the enabling object. An *event link* connects a triggering entity (object, process, or state) with a process that it invokes. The event types that OPM supports include state entrance, state change, state timeout, process termination, process timeout, reaction timeout, and a host of external events, which include clock events and triggering by an environmental entity, such as a user or an external device.

The Bimodal Graphic-Text Representation of OPM

Two semantically equivalent modalities, or representation modes, one graphic and the other textual, jointly express the same OPM model. A set of inter-related Object-Process Diagrams (OPDs) constitute the graphical, visual OPM formalism. Each OPM element is denoted in an OPD by a symbol. The OPD graphical syntax specifies correct and consistent ways by which entities can be connected via structural and procedural links, each having its specific, unambiguous semantics.

Object-Process Language (OPL) is the textual counterpart modality of the graphical OPD set. Defined by a context-free grammar, OPL is a dual-purpose language, oriented towards humans as well as machines. Catering to human needs, OPL is designed as a constrained subset of English, which enables domain experts and system architects to collaborate in analyzing and designing a system. Every OPD construct is expressed by a semantically equivalent OPL sentence or phrase. Designed also for machine interpretation through a well-defined set of production rules, OPL provides a solid basis for automating the generation of the designed application. This dual representation of OPM increases the processing capability of humans according to the cognitive theory (Mayer 2001).

Catering to the modality principle of cognitive theory (Mayer 2001), OPM enables modeling systems both graphically, via an OPD set, and textually, using OPL, a subset of English. OPCAT[1] (Dori, Reinhartz-Berger and Sturm 2002), a Java-based Object-Process CASE Tool, automatically translates each OPD into its equivalent OPL paragraph (collection of OPL sentences) and vice versa.[2]

Figure 1 is a snapshot of OPCAT 2 graphic user interface.

Using OPCAT, users can model complex systems, express and query knowledge. The example in

Figure 1 deals with medicine and the particular OPD zooms into the diagnosing process. The GUI of OPCAT 2 in

Figure 1 shows the hierarchy of diagrams and things (i.e., objects and processes) on the left, the graphic (OPD) window at the top right, the text (OPL) window at the bottom right, and the palette with the various OPM entities and links at the bottom. Some of the OPL sentences, all of which were generated automatically by OPCAT, are shown in the OPL window. To get the feeling for their closeness to natural English, some of these sentences are listed below. Next to each OPL sentence is its type.

Patient can be **healthy** or **sick.** *(State enumeration sentence)*
Doctor handles **Test Prescribing.** *(Agent sentence)*
Test Prescribing yields **Test Prescription.** *(Result sentence)*
Doctor handles **Physical Examination** Process. *(Agent sentence)*
Physical Examination Process requires **Patient.** *(Instrument sentence)*
Patient and **Doctor** handle **Interviewing.** *(Agent sentence)*
Testing requires **Patient, Laboratory,** and **Test Prescription.** *(Instrument sentence)*
Testing yields **Test Results.** *(Result sentence)*
Doctor handles **Result Analyzing.** *(Agent sentence)*

These example sentences show how knowledge that combines structure and behavior can be represented in both intuitive (yet formal) graphics and naturally intelligible text. Users who are not familiar with the graphic notation of OPM can verify their specifications by inspecting the OPL sentences, which are automatically generated on the fly in response to the user's graphic input. For example, comparing the OPL sentence "**Doctor** handles **Test Prescribing.**" with the relevant part of the OPD shows that the link which gives rise to the reserved word "handles" is the agent link, which originates from the agent (**Doctor**) and points to the destination process (**Test Prescription**). As the example shows, the knowledge that OPM can represent is not restricted to just structural. It can also be procedural, showing temporal order and enabling cause and effect analysis.

The OPM Text-Graphic Equivalence Principle

A basic OPM principle, demonstrated in
Figure 1, is the graphic-text equivalence:

> *Anything that is expressed graphically by an OPD is also expressed textually in the corresponding OPL paragraph, and vice versa.*

The option of choosing between natural language text and graphics to specify some modeling artifacts and alternating between the two at the user's discretion is a unique feature of OPM and OPCAT. The relatively small set of OPD symbols and corresponding OPL sentence types increases OPM's accessibility and ease of learning and using of to

[1] OPCAT 2 can be downloaded free from http://iew3.technion.ac.il/~dori/opcat/index-continue.html

[2] As of writing this paper, the text-to-graphics direction is only partially operational.

both system architects and domain experts. System architects, who are familiar with the OPD graphic syntax, can use the visual OPD symbols, while domain experts can read the English-like OPL script to understand the specification and verify that the system is designed to meet their requirements. The automatic translation into an OPL script also improves the documentation quality of the developed system. The automatic implementation (code and database schema) generation, currently under development, will ensure that the specification designed by the system architects and endorsed by the domain experts is indeed reflected without any translational gap in the actual system.

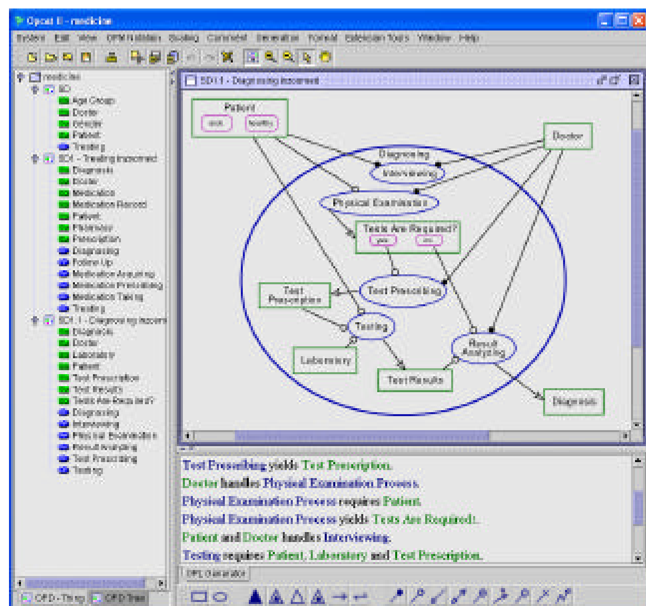

Figure 1. A snapshot of OPCAT 2 GUI, showing the OPD window (top), part of the corresponding OPL window (bottom), and the OPD tree (left).

A major advantage of OPM, in addition to being a universal system modeling framework that maintains the required balance between the system's structure and behavior, is its ability to model the system at varying levels of detail. Easy traversal among varying levels of detail. This ability is made possible through three refinement/abstraction mechanisms: (1) *unfolding/folding*, which is used for refining/abstracting the structural hierarchy of a thing and is applied by default to objects; (2) *in-zooming/out-zooming*, which exposes/hides the inner details of a thing within its frame and is applied primarily to processes; and (3) *state expressing / suppressing*, which exposes/hides the states of an object. Using flexible combinations of these three abstraction/refinement mechanisms, OPM enables specifying a system to any desired level of detail without losing legibility and comprehension of the resulting specification. These mechanisms enable complexity management by providing for the creation of a set of interconnected OPDs (along with their corresponding OPL paragraphs). Each OPD is not excessively complex, but together they can specify very complex systems. Modules consisting of both objects and processes can be generated by utilizing these OPM's built-in refinement/abstraction mechanisms.

The hierarchical specification prevents information overload and enables comfortable human processing. The complete OPM system specification is expressed graphically by the resulting set of consistent, inter-related OPDs, and textually by the corresponding OPL script, which is the union of the information expressed in the OPL paragraphs.

The ability to freely traverse up and down the abstraction-refinement hierarchy provides for natural modeling of modules, a critical issue in computational synthesis. This ability caters to another cognitive principle – the limited channel capacity (Mayer 2001), which is addressed in OPM and implemented in OPCAT with the three abstraction/refinement mechanisms.

Comparing and Selecting a Computational Synthesis Modeling Platform

Following the introduction of UML and OPM, we return to the list of criteria of features for a computational synthesis modeling platform and examine the two candidate approaches with respect to each criterion in order to select the one with the most desirable features.

1. Domain independence: UML is a language for specifying software systems, while OPM is a complete methodology, designed for developing systems in general. Since UML is geared to software and uses software jargon, it only models informatical objects. To cover a broader spectrum, OPM assigns to each thing (object or process) an *essence* attribute, which enables the distinction between *physical* things and *informatical* things. Another important thing attribute is *affiliation*, which can be *systemic* (denoting that the thing belongs to the system) or *environmental* (denoting that the thing is part of the environment). Physical things are denoted graphically as having depth (they are shaded) while environmental things are marked by a dashed contour. Both physical and informatical things, as well as systemic and environmental things, can appear in the same diagram. This allows for modeling of embedded systems that combine hardware and software, and biological systems where the physical and control aspects can be distinguished. Figure 2 shows a simple OPD and its corresponding OPL paragraph that contain combinations of physical and environmental things. Note that the default essence and

affiliation values are informatical and systemic, respectively, so sentences are generated only for the non-default essence and affiliation values, namely physical and environmental. Another attribute of thing, which is of particular interest to the computational synthesis domain, is *source*, with values *natural* and *artificial*. This attribute enables specifying biological systems and systems that combine natural and man-made artifacts.

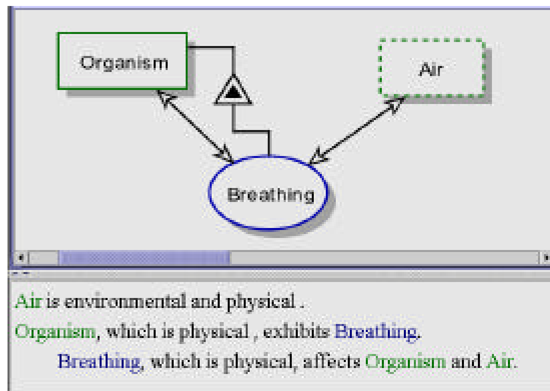

Figure 2. Examples of physical (shaded) and environmental (dashed) things and the corresponding OPL sentences

The OPM ontology is generic at the most basic level: The elementary building blocks of the universe are stateful objects and processes. These elementary building blocks are applicable to any domain of discourse in general and to any conceivable computational synthesis application in particular. This generic ontology enables a domain-neutral specification of computational synthesis systems that researchers from disparate domains can relate to rather than having to adapt their points of reference to the set of concepts that are peculiar to the domain under consideration.

2. Modularity: As noted, modularity is a key issue in computational synthesis (Lipson, Pollack, and Suh 2002). Apart from package, which is a collection of packages and classes, UML does not have adequate complexity management tools. The use of multiple models exacerbates the definition of modules, since each module needs to be specified using several diagram type to account for its various aspects. With OPM, modules can be defined in a straightforward manner using the in-zooming and unfolding abstraction refinement mechanisms. Modules link with each other to specify functional units at varying scales, starting from the elements and going all the way to a system that is a society of life forms. OPM modules can be designed to replace each other. Replacability of two modules entails that the interface of the replacing module be congruent with that of the replaced module. This means that all the links in the two exchangeable modules be of the same number and type.

3. Structure-behavior balance: The object-oriented (OO) paradigm, which UML models, favors objects over processes. Objects are the supreme entities, and processes can only be modeled as "methods" or "services" of objects. This is the implication of the encapsulation principle, which requires that processes always be embedded in objects and forbids the existence of stand-alone processes. While this may be a good programming practice, in reality, many processes cannot be correctly modeled as being "owned" by one particular class of objects. Rather, their occurrence mandates the presence of several objects of different classes that are equally "responsible" for the process occurrence. Thus, in spite of the fact that UML requires up to nine different diagram types to model a single system, none of these diagrams allows the representation of a process as an independent entity. This is a very severe limitation of the OO concept and consequently of UML: Reality must often be coerced to reflect the fact that processes cannot be modeled as entities in their own right.

In OPM there is no such limitation. As we have seen, the single OPM model allows objects and processes to co-exist and interact on equal footing – there is no "discrimination" against processes and no object supremacy. This way, structure and behavior are modeled in a fair and balanced way without one suppressing the other (Dori, 1995). OPM reflects in a balanced and transparent way how the architecture, i.e., the combination of structure (objects, or form) and behavior (processes, or function) of the system, which the computational synthesis system tries to evolve, fulfils the functional requirements (as measured by the level of fitness) for its survival in the environment.

4. Metamodeling capability: Due to the generic nature of OPM, it is able to specify, in addition to the evolvable life form as a final product, also the universal computational synthesis metamodel, i.e., the model of the system that makes this evolution happen. The UML universal computational synthesis metamodel is extremely difficult, if not impossible, to obtain. The next section demonstrates how such an OPM metamodel would look like.

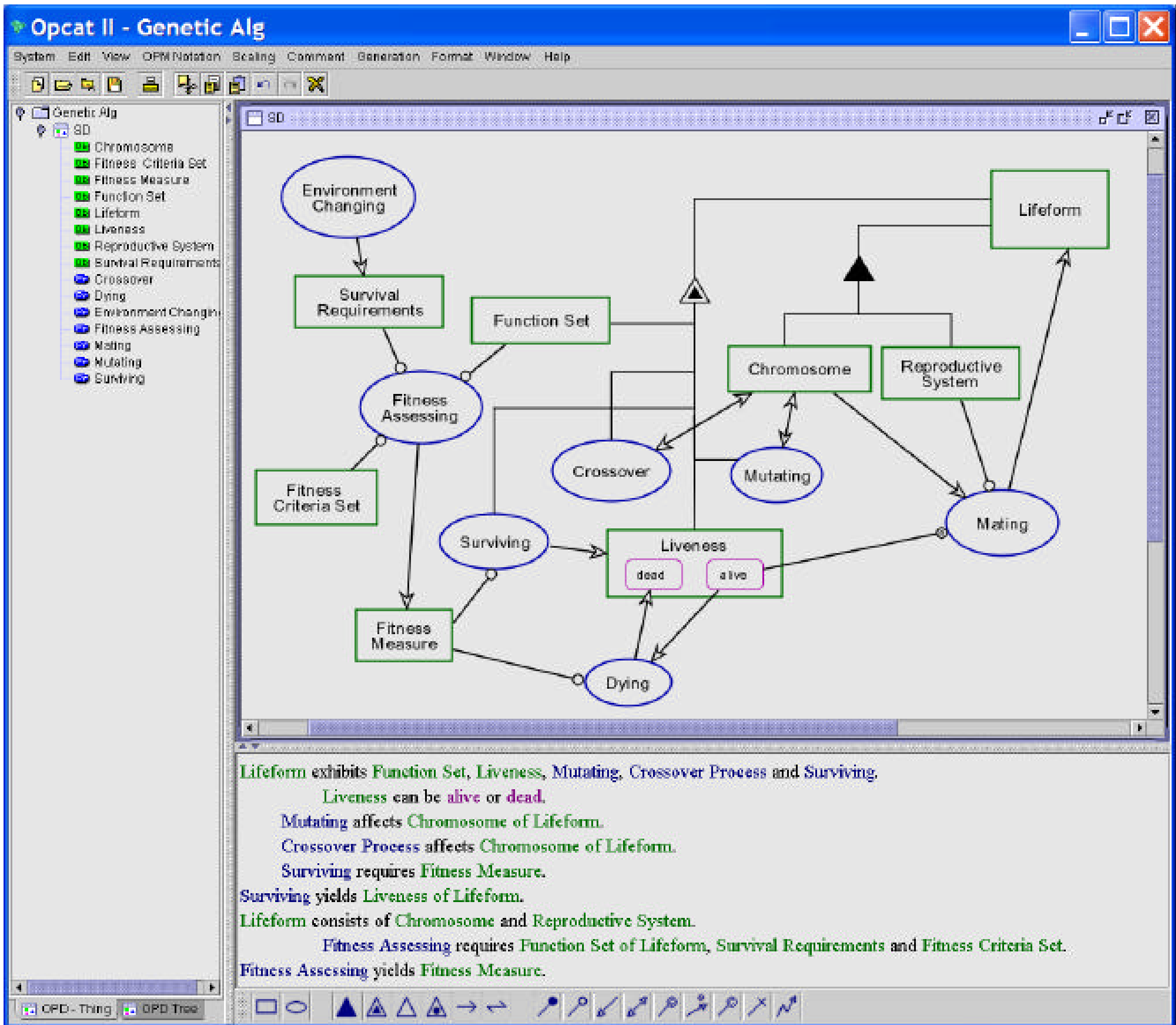

Figure 3. A domain-neutral OPM metamodel of a generic evolutionary or computational synthesis system

The Three Computational Synthesis Meta-modeling Levels

OPM can be applied to model systems of computational synthesis at three related levels along the abstraction-concreteness continuum. The highest, most abstract level model is the domain-neutral OPM model of a generic evolutionary or computational synthesis system that can be used as a template for the various domains. At a lower, intermediate abstraction level is the domain-specific OPM model of the particular computational synthesis application, such as computational synthesis of analog electronic circuits, robots, or chess players. Finally, at the most concrete level is the model of the computational synthesis system in terms of the pertinent domain. Thus, in the domain of analog electronic circuits, the computational synthesis system is specified in terms of the resistors, capacitors, inductors, and transistors as the building blocks and the set of instructions such as "x-move-to-new" and "x-cast-to-input" as the permissible atomic operations in the evolution process (Lohn, et al., 2000). The end result of the evolved system—a successful circuit—is depicted as is customary in the domain, using the capacitor, inductor, and transistor symbols. In the domain of robots, the computationally synthesized system is specified in terms of the elements constructing

the truss, their joints, and the neural network that controls them. And in the domain of chess players, each player may be modeled as a collection of strategies for board situations expressed in chess notation. Hence, while researchers do have means to express their resulting evolved life forms in their domain-specific language, they usually do not have a formal means to describe the system that generated these results.

Future Research

How do we go about developing such common computational synthesis formalism? There are two complementary approaches for achieving this result. One is top-down and the other is bottom-up. The top-down direction entails modeling the generic evolutionary system based on computational synthesis and biologically induced theories, while the bottom-up direction concerns modeling examples and case studies of research projects from various domains and trying to see how they fit in the generic framework. For best results, both directions should be pursued concurrently. Figure 3 shows a first attempt at the top-down approach of modeling a generic computational synthesis system. The OPD shows the main objects and processes as they may be commonly used. These might be translated into the various application domains. For the bottom-up direction we will explore a number of domains, which may include analog electronic circuits (Koza et al., 1997; Lohn, et al., 2000), robotics (Sims, 1994; Lipson & Pollack, 2000), computer programs (Koza, 1989) and game theoretic issues (Axelrod, 1987).

References

Axelrod, R. (1987). The evolution of strategies in the iterated prisoner's dilemma. In *Genetics Algorithms in Simulated Annealing*. Davis, L. (Ed.), Morgan-Kaufman, 32-41.

Dori, D. (1995). Object-Process Analysis: Maintaining the Balance between System Structure and Behavior. *Journal of Logic and Computation*, 5, 227-249.

Dori, D. (2002a). Why Significant Change in UML is Unlikely. *Communications of the ACM*, pp. 82-85, Nov. 2002.

Dori, D. (2002b). *Object-Process Methodology - A Holistic Systems Paradigm*, Heidelberg: Springer Verlag.

Dori, D., Reinhartz-Berger, I., and Sturm, A. OPCAT – A Bimodal CASE Tool for Object-Process Based System Development. 5th International Conference on Enterprise Information Systems (ICEIS'2003), Angers, France, 23-26 April, 2003 (to appear).

Koza J.R., (1989) Hierarchical genetic algorithms operating on populations of computer programs, *Proc. 11th Int. Joint Conference on Genetic Algorithms*, 768-774.

Koza, J.R., Bennett, F.H. Andre, D., M. Keane, A and Dunlap, F. (1997). Automated Synthesis of Analog Electrical Circuits by Means of Genetic Programming, *IEEE Trans. on Evolutionary Computation*, 1:109-128.

Lipson, H. & Pollack J. B., (2000). Automatic Design and Manufacture of Artificial Lifeforms, *Nature* 406:974-978.

Lohn, J.D., Haith, G.L., Colombano, S.P. & Stassinopoulos, D. (2000) Towards Evolving Electronic Circuits for Autonomous Space Applications. *Proc. IEEE Aerospace Conference*, Big Sky, MT.

Lipson, H. Pollack, J.B. & Suh, N.P. (2002) On the Origin of Modular Variation.
http://www.mae.cornell.edu/lipson/CS750/readings.htm

Mayer, R.E. (2001). *Multimedia Learning*. New York, NY: Cambridge University Press.

Object Management Group (2000). Unified Modeling Language (UML) 1.3 Documentation.

Peleg, M. and Dori, D. (2000). The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods. *IEEE Transaction on Software Engineering*, 26 (8):742-759.
http://www.omg.org/cgi-bin/doc?formal/2000-03-01

Siau, K. and Cau, C. (2001). Unified Modeling Language: A complexity analysis. *Journal of Database Management* 12(1):26-34.

Simons, T. (2001). Dependencies and associations. Precise UML Group Email Forum (June 21); see www.cs.york.ac.uk/puml/puml-list-archive/0310.html.

Sims, K. (1994). Evolving Virtual Creatures. Computer Graphics Annual Conference Series (SIGGRAPH'94 Proceedings) pp. 15-22.

Soffer, P., Golany, B., Dori, D., and Wand, Y. (2001). Modeling off-the-shelf information systems requirements: An ontological approach. Requirements Engineering 6, 3,183–199.