SODA: Not Just a Drink!

From an Object-Centered to a Balanced Object-Process Model-Based Enterprise Systems Development

Dov Dori
Technion, Israel Institute of Technology
and
Massachusetts Institute of Technology
dori@mit.edu

ABSTRACT

Two software system lifecycle development paradigms have been competing on the minds and hearts of software developers and executives: The traditional Object-Oriented approach and the emerging Service-Oriented Architecture (SOA) or SO Development of Application (SODA). While OO puts objects and their encapsulated behavior at the center stage, emphasizing primarily rigid structure, SODA hails services as the prime players to cater primarily to behavior. We discuss the new SOA technologies from the extended enterprise and the service network all the way to the atomic service level and show that Object-Process Methodology (OPM), which strikes a unique balance between structure and behavior, is most suitable as the underlying SOA-based lifecycle engineering approach. Using OPCAT, the OPM-supporting systems modeling software environment, we construct the top level diagram of a model of SODA and simulate it using animation in order to show how OPM conveniently serves as an ideal overarching comprehensive methodology that encompasses the entire spectrum of service-oriented enterprise systems development.

1. Introduction: From Objects to Services

The Object-Oriented (OO) paradigm has dominated the software world since the early 1980's. It suppressed the previously accepted functional programming paradigm, which put functions at the central players in computer programs. The domination of the OO paradigm has promoted the proliferation of a plethora of object-oriented analysis and design methods. The amount of methods that were basically similar in principle but different in their types of diagrams and symbols, culminated in the proposal and eventual adoption by the Object Management Group (OMG) of the Unified Modeling Language (UML) as the lingua franca of OO-based software development.

Until recently, it looked like OO, along with its ensuing technologies, such as Design Patterns, Component-Based Development, and Aspect-Oriented Programming, is the ultimate solution to many problems that were at the basis of the "software crisis" of the late 1980's, and that consequently the OO domination is going to last for a very long time. However, the recent move of the IT world to cater to the business needs of enterprises of all sorts and magnitudes has created a rather surprising twist—a modern return to the good old functional approach, which puts back functions (now called services) rather than objects as the top-level entities in the architecture of complex, enterprise-wide systems.

It is thus becoming more and more evident that the most popular new three-letter acronym of the decade is going to be SOA—Service-Oriented Architecture [1], or its four-letter companion SODA—Service-Oriented Development of Applications [2]. Dressed with a new architectural garment and backed with state-of-the-art Internet-based cross-enterprise networking capabilities, this new trend is admittedly way more sophisticated and technologically advanced than its pre-OO old functional (also called structured) programming counterpart.

Software has slowly been undergoing a series of decoupling processes. Client/Server architecture has separated the database from the "fat client". IN the emerging "thin-client", the user interface was decoupled from the business logic. Now SOA is

decoupling the integration logic from the business logic. Newly proposed process-driven architectures decouple the process logic from the business logic. In a process driven architecture, the application is viewed from a process perspective [2]. A top-level process is broken down into level-1 subprocesses, called *activities* and each activity is further broken down into level-2 subprocesses, called tasks. The process information is then recorded as a "digital process" that can be executed within an "orchestration engine", also dubbed "process execution engine" or "process virtual machine", which we describe later.

2. Services as Software Processes

To better understand what Service-Oriented Architecture (SOA) really is, we need to first clarify the term service. A *service* is a unit of work done by a service provider to achieve desired end results for a service consumer. In other words, a service is a *process* which, by transforming objects, generates value for its consumer.

Examples of services in the real world have been abundant for thousands of years. Selling goods, hairdressing, mail delivering, food cooking and house cleaning are examples of services. In each such service, some object is transformed to bring value to its consumer. Selling goods changes the ownership of the goods from the seller to the buyer, who, in turn, uses them for her benefit. Hairdressing changes the look of the customer's hair from untidy to tidy. Mail delivering changes the location of each sent mail item from the sender to the recipient, usually benefiting both sides. Food cooking changes the raw food materials from inedible to edible, so the consumer can eat them and satisfy his hunger. House cleaning changes the house from dirty to clean, benefiting the tenants living in it.

In each service, as all these examples consistently show, there is at least one object that undergoes a state change, and it is this change exactly that creates value for the consumer or the customer.

This value motivates service providers to develop and offer services that consumers are willing to pay for. Provides develop specializations of their skills so they benefit from the economy of scale. The mail deliverer, for example, delivers hundreds of mail items to the same street in a fraction of the total time it would take each sender to personally deliver his mail piece to the recipient.

The pattern we have seen with services shows that invariantly, at least one object is transformed in order for the service to be of value. Hence we cannot talk about services in a vacuum—there must always be a transformee (a transformed object) which is involved. Moreover, we cannot talk about a service without reference to its provider and consumer, which are objects too.

3. Web Services

The situation in the software world is not principally different than that in the real world: services transform (informatical) objects by consuming or crating or affecting (changing the state of) these objects. And like in the real world, there are providers and consumers for each service. The service provider and the service consumer are "*software agents*," i.e., they are intelligent programs that act on behalf of their owners, who may be individual humans or actual organizations.

Web services are the atomic units of service which are available on the Internet. Web services are founded on three XML-based components: Web Services Description Language (WSDL), Simple Objects Access Protocol (SOAP), and Universal Description, Discovery, and Integration (UDDI). SOAP defines an XML messaging protocol for basic service interoperability. WSDL introduces a common grammar for describing services. UDDI provides the infrastructure required to publish and discover services in a systematic way. Together, these specifications allow applications to find each other and interact following a loosely coupled, platform-independent model.

4. SOA and SODA Defined and Analyzed

Following the definition in [1], Service-Oriented Architecture is a collection of distributed, self-contained Web services that communicate with each other independently of the context or state of other services. Put differently, SOA is an approach to architecting systems that is based on retrievable Web services which can be combined via loose coupling among interacting software agents.

While the OO encapsulation principle mandates that data and its processing be bound together, SOA constitutes a major departure from the OO approach, as it considers services as stand-alone entities, not encapsulated within a certain object. For example, if consumer electronic would be designed following OO philosophy, every CD would come bundled with its own player software. Instead, in the SA spirit, there is loose coupling between a CD and a CD player via a standard interface, so a CD can be played on any player. The loose coupling among interacting software agents is achieved by (1) few simple, universally available and ubiquitous interfaces with the participating software agents, and (2) schema-constrained extensible messages delivered through the

interfaces. The extensibility of the schema allows new versions of services to be introduced while keeping existing services.

Table 1. Scope and number of Web Services as of January 2006 provided by the SOA Portal at www.soahub.com/

Accounting (0)	Advertising (0)
Automotive (0)	Business Intelligence (1)
Business Process Management (2)	Communications Web Services (1)
Composite Applications (1)	Content Management (0)
ECommerce (0)	Enterprise Service Bus (ESB) (3)
Financial (0)	HealthCare / HIPAA (0)
Human Resources (HR) (0)	Insurance (0)
Legal (0)	Manufacturing (0)
Marketing (0)	Pharmaceuticals/Biotechnology (0)
Real Estate / Construction (0)	Sales / CRM (0)
SOA/Web Services Management (1)	SOA / Web Services Security (1)
Telecommunications (0)	Transportation / Logistics (0)

At least "on paper", SOA is becoming the solution of choice to contemporary IT world challenges. However, in practice, we are still at a very early stage in the evolution of SOA as a real encompassing solution. To realize how early, suffice it to look at the list in Table 1 of domains and the number of services in each one of them in the SOA portal defined there as "the most comprehensive directory of SOA solutions and solution providers." Only one domain (Enterprise Service Bus) has three services, one domain (Business Process Management) has two services, five have one service each, and the remaining 18 domains have no service whatsoever!

A very closely related acronym to SOA is SODA—Service-Oriented Development of Applications [2]. SODA applies the concepts of a service-oriented architecture to the design of a single application. SOA (or SODA) builds on concepts originated in object-oriented and component-based development and extends them with distributed computing and quality of service concepts. It handles asynchronous messaging, XML for identifying data types and use of protocols over the more traditional Application Program Interfaces (APIs).

Service oriented applications are composed of the run-time loose bundling of services, which are "orchestrated" to solve some problem. This loose coupling is the opposite of tighter compiling that has been used to bind modern object oriented systems. While each individual service is compiled, the orchestration between the services is intentionally kept out of the compiling, and the binding is rather done via compilation that is executed in run-time by an *orchestration engine*.

5. Orchestration Engine: The Powerhouse of the Extended Enterprise

An orchestration engine is a server that intelligently links services to each other [2]. Code segments called *orchestration scripts* define the interaction between services. These scripts are fed into the orchestration engine, where they are executed. Orchestration engines participate in the *service network*.

A service network is an application level network that leverages SOA. It is composed of a number of *service network participants* and typically contains many services.

The service network participants commonly include, in addition to orchestration engines, such technical components as service engines, SOAP engines, application servers, SOAP (both domain and content) routers, service directories, and Transaction Processing monitors for both short- and long-lived transactions. The latter are also known as *sagas*, which can last as long as days.

The orchestration scripts integrate internal applications and facilitate the *Extended Enterprise*—a concept that promotes the use of application-level communications between companies to enable near real-time integration. In an Extended Enterprise environment, applications of any participating company can be extensions to applications of peer partners. Some engines are designed to be specific to an orchestration language like BPEL4WS, discussed in the next section, while others will most likely support multiple orchestration languages that will develop in the future.

6. Business Process Execution Language

The resulting enterprise model serves as a basis for identification of services, which are mapped using *Business Process Execution Language* (BPEL) for Web services [3]. BPEL, also sometimes identified as WS-BPEL, BPELWS or BPEL4WS, is an orchestration language that defines a notation for specifying business process behavior based on Web Services. An XML-based language, BPEL is designed to enable task-sharing for a distributed- or a grid-computing environment, both within and across organizations, using a combination of Web services. Written by developers from such software houses as BEA Systems, IBM, and Microsoft, BPEL combines and replaces earlier versions like IBM's WebServices

Flow Language (WSFL) and Microsoft's XLANG specification.

By providing a language for the formal specification of business processes and business interaction protocols, BPEL4WS extends the Web Services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intra-corporate and the business-to-business spaces.

Using BPEL, one can formally describe a business process that will take place across the Web such that any cooperating entity can perform one or more steps in the process the same way. In a supply chain process, for example, a BPEL program might describe a business protocol that formalizes the information pieces that comprise a product order, and what exceptions may have to be handled. BPEL4WS allows companies to describe business processes that include multiple Web services and standardize message exchange internally and between partners. It is important to note that the BPEL specification does not specify how a given Web service should process a given order internally.

Standard Web Service protocols are not sufficient for true systems integration. The full potential of Web Services as an integration platform can be achieved when applications and business processes are able to integrate their complex interactions by using a standard process integration model. Models for business interactions typically assume sequences of peer-to-peer message exchanges, both synchronous and asynchronous, within stateful, long-running interactions involving two or more parties. However, the interaction model supported by WSDL is a stateless model of synchronous or uncorrelated asynchronous interactions. BPEL4WS formally defines and describes business interactions and the message exchange protocols used by business processes in their interactions. The definition of such business protocols involves precisely specifying the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal implementation.

The separation of the public aspects of business process behavior from internal or private aspects is due to (1) reluctance of businesses to reveal their internal decision making and data management to their business partners, and separating public from private process, so private aspects of the process implementation can be changed without affecting the public business protocol. These protocols are clearly described in a platform-independent manner and capture all behavioral aspects that have cross-enterprise business significance. This enables each participant to understand and plan for conformance to the business protocol without the need for additional time consuming human agreements, which still hinder a more rapid progress in business-to-business automation.

Human user interactions are currently not covered by WS-BPEL, which is primarily designed to support automated business processes based on Web services. In practice, however, many business process scenarios require user interaction. A work under way by IBM and SAP, dubbed BPEL4People [4], describes scenarios where users are involved in business processes, and defines appropriate extensions to WS-BPEL to address these.

7. SOA-based Lifecycle Development

Schematically the lifecycle of an enterprise venturing SOA comprises three main processes: Conceptual Modeling, Enterprise Service Modeling, and Web Services Embedding. The lifecycle of SOA-based development ideally starts with a comprehensive analysis which results in a conceptual model of the enterprise, in which services are to be embedded and operational. Hence, successful SOA implementation requires careful planning founded on a detailed yet holistic view of the contemplated architecture. *Business Process Modeling, Analysis, and Management* (BPM) is the orchestration of various business systems into identifiable and controllable system-of-systems that can potentially consolidate a myriad of related enterprise processes. There is a plethora of BPM tools that can be used at this stage.

Conceptual Modeling yields the Enterprise Conceptual Model. This model constitutes the basis for the skeletal Enterprise BPEL4WS Model. Using the Web Services Repository, the skeletal Enterprise BPEL4WS Model is populated with concrete Web Services via the Web Services Embedding process, changing its state from "skeletal" to "web-services embedded".

The Enterprise itself—the focus of all these activities, is the object undergoing the lifecycle. The initial state of the Enterprise is "pre-modeled". Other states are "conceptually modeled", "BPEL4WS-modeled", and "web-serviced", the latter being the final state. As the Enterprise undergoes its SOA lifecycle, it changes its states. Each of the three major processes also affects the Enterprise by changing its state. Conceptual Modeling changes the Enterprise from "pre-modeled" to "conceptually modeled", Enterprise Service Modeling changes it from "conceptually modeled" to "BPEL4WS-modeled", and Web Services Embedding—from "BPEL4WS-modeled" to "web-serviced".

As each one of the three major processes—Conceptual Modeling, Enterprise Service Modeling, and Web Services Embedding—is executed, one of the three parts comprising the Enterprise Model Set— the Enterprise Conceptual Model, the Enterprise BPEL4WS Model, and the Web Services Repository—are respectively created.

Since the Enterprise keeps evolving, a Conceptual Model Enhancing process usually takes place continuously, using the current web-services embedded Enterprise BPEL4WS Model, and it keeps updating the Enterprise Conceptual model.

While this SOA lifecycle model makes a lot of sense, in spite of the significant efforts by the leading software giants worldwide, current tools cover only segmented parts of the entire lifecycle of SOA-based development. For example, Business Process Analysis tools model organizational processes but do not directly carry on the resulting conceptual model to tackle the technological part, where BPEL is utilized. BPEL, in turn, is not seamlessly integrated into the three XML-based components of Web services— Web Services Description Language (WSDL), Simple Objects Access Protocol (SOAP), and Universal Description, Discovery, and Integration (UDDI).

8. SOA vs. OO: The Giants Battle

In most complex systems in general, and enterprises in particular, structure and behavior are highly intertwined and hard to separate. For example, in a manufacturing system, the manufacturing process cannot be contemplated in isolation from its inputs—raw materials, model, machines, and operators, and its output—the resulting product. The inputs and the output are objects, some of which are transformed by the manufacturing process, while others just enable it. In spite of this simple, straightforward observation, what we are witnessing is a conflict between two competing paradigms: OO and SOA.

While OO emphasize structure using objects as the only top players in the modeling game, SOA conversely focuses on the system's behavior via modeling processes (services) while suppressing the role of objects as prime building blocks of the enterprise conceptual model. The current state of affairs is that each of these two competing approaches highlights one of the two major system aspects—structure and behavior—at the expense of suppressing the other. The bottom line is that neither OO nor SOA alone can possibly serve as a solid foundation for a comprehensive conceptual modeling foundation of any interesting complex system under study, as each suffers from lack of appropriate representation of either structure or behavior. All this points to one clear conclusion: There must be a methodology that formally and intuitively unifies, merges, and reconciles the Object- and Service-Oriented approaches. Luckily, such paradigm is already inexistence for over a decade. It is called Object-Process Methodology (OPM).

9. Object-Process Methodology

Modeling of complex systems should conveniently combine structure and behavior in a single model. Motivated by this observation, OPM [5, 6] is a comprehensive, holistic approach to modeling, study, development, engineering, evolution, and lifecycle support of systems.

DMReview [1] defines:

> **Object-Process Methodology (OPM):**
> A conceptual modeling approach for complex systems that integrates in a single view the functional, structural and procedural aspects of the modeled system using formal yet intuitive graphics that is translated on the fly to a subset of natural language.

Employing a combination of graphics and a subset of English, the OPM paradigm integrates the object-oriented, process-oriented, and state transition approaches into a single frame of reference. Structure and behavior coexist in the same OPM model without highlighting one at the expense of suppressing the other to enhance the comprehension of the system as a whole.

Rather than requiring that the modeler views each of the system's aspects in isolation and struggle to mentally integrate the various views, OPM offers an approach that is orthogonal to customary practices.

According to this approach, various system aspects can be inspected in tandem for better comprehension. Complexity is managed via the ability to create and navigate via possibly multiple detail levels, which are generated and traversed through by several abstraction/refinement mechanisms.

Due to its structure-behavior integration, OPM provides a solid basis for representing and managing knowledge about complex systems, regardless of their domain. This section provides an overview of OPM, its ontology, semantics, and symbols.

10. The OPM Ontology

The elements of the OPM ontology, shown in Figure 1, are divided into three groups: entities, structural relations, and procedural links.

10.1. Entities

Entities, the basic building blocks of any system modeled in OPM, are of three types: stateful objects, namely *objects* with *states*, and *processes*. As defined below, processes transform objects by (1) creating them, (2) destroying them, or (3) changing their state. The symbols for these three entities are respectively shown as the first group of symbols at the left hand side of Error! Reference source not found., which is the symbols in the toolset available as part of the GUI of OPCAT [7].

Figure 1. The three groups of OPM symbols in the toolset of OPCAT

10.2. OPM Things: Objects and Processes

Objects are (physical or informatical) things that exist, while *processes* are things that transform (create, destroy, or change the state of) objects. Following is a set of basic definitions that build on top of each other.

*An **object** is a thing that exists.*

Objects are the things that are being transformed in the system. Processes are the things that transform objects in the system.

*A **process** is a thing that represents a pattern of object transformation.*

In OPL, **bold Arial font** denotes non-reserved phrases, while non-bold Arial font denotes reserved phrases. In OPCAT, various OPM elements are colored with the same color as their graphic counterparts (by default, objects are green, processes are blue, and states are brown).

Objects and processes are collectively called *things*. The first two rows of Table 2 show the symbol and a description of the two types of OPM things. The next two rows show two basic attributes that things can have: essence and affiliation.

***Essence** is an attribute that determines whether the thing is physical or informational.*

The default essence is informatical. A thing whose essence is physical is symbolized by a shaded shape.

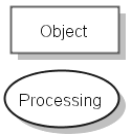
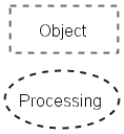
Thing / Attribute	Symbol	Description / OPL sentence
Object		A thing (entity) that has the potential of stable, unconditional physical or mental existence. **Object Name** is an object**.**
Process		A thing representing a pattern of transformation that objects undergo. **Processing** is a process**.**
Essence		An attribute that determines whether the thing (object or process) is physical (shaded) or informational. **Processing** is physical**.**
Affiliation		An attribute that determines whether the thing is environmental (external to the system, dashed contour) or systemic. **Processing** is environmental**.**

Table 2. Things of the OPM ontology and their basic attributes

***Affiliation** is an attribute that determines whether the thing is environmental (external to the system) or systemic.*

The default affiliation is systemic. A thing whose affiliation is environmental is symbolized by a dashed contour.

10.3. OPM States

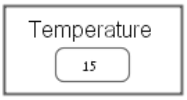
Objects can be stateful, i.e., they may have one or more states.

*A **state** is a situation at which an object can exist at certain points during its lifetime or a value it can assume.*

Stateful objects can be affected, i.e., their states can change.

***Effect** is a change in the state of an object.*

Table 3. States and values

	Symbol	Description / OPL sentence
Stateful object with two states		A situation at which an object can exist. **Website** can be **reachable** or **unreachable.**
Value		A value that an object can assume. **Temperature** is **15.**
Stateful object with three states: initial, default, and final		A state can be initial, default, or final**.** **Car** can be **new,** which is **initial, used,** which is **default,** or **junk,** which is **final.**

10.4. OPM Structure Modeling

Structural relations express static, time-independent relations between pairs of entities, most often between two objects.

Structural relations, shown as the middle group of six symbols in Figure 1, are of two types: fundamental and tagged.

10.4.1. The four fundamental structural relations

Fundamental structural relations are a set of four structural relations that are used frequently to denote relations between things in the system. Due to their prevalence and usefulness, and in order to prevent too much text from cluttering the diagram, these relations are designated by the four distinct triangular symbols shown in Figure 1.

The four fundamental structural relations are:

(1) aggregation-participation, a solid triangle, ▲, which denotes the relation between a whole thing and its parts,

(2) generalization-specialization, a blank triangle, △, which denotes the relation between a general thing and its specializations, giving rise to inheritance,

(3) exhibition-characterization, a solid inside blank triangle, ◬, which denotes the relation between an exhibitor – a thing exhibiting a one or more features (attributes and/or operations) – and the things that characterize the exhibitor, and

(4) classification-instantiation, a solid circle inside a blank triangle, ◬, which denotes the relation between a class of things and an instance of that class.

Table 4. The fundamental structural relation names, OPD symbols, and OPL sentences

Structural Relation Name		Root & Refineables
Forward	Backward	
Aggregation	Participation	Whole Parts
Exhibition	Characterization	Exhibitor Features
Generalization	Specialization	General Specializations
Classification	Instantiation	Class Instances

Table 4 lists the four fundamental structural relations. The name of each such relation consists of a pair of dash-separated words. The first word is the forward relation name, i.e., the name of the relation as seen from the viewpoint of the thing up in the hierarchy. The second word is the backward (or reverse) relation name, i.e., the name of the relation as seen from the viewpoint of the thing down in the hierarchy of that relation.

Each fundamental structural relation has a default, preferred direction, which was determined by how natural the sentence sounds. In **Table 4**, the preferred shorthand name for each relation is underlined. Each one of the four fundamental structural relations is characterized by the hierarchy it induces between the root—the thing attached to the tip of the triangle and the leaves—the thing(s) attached to the base of the triangle, as follows.

(1) In aggregation-participation, the tip of the solid triangle, ▲, is attached to the whole thing, while the base—to the parts.

(2) In generalization-specialization, the tip of the blank triangle, △, is attached to the general thing, while the base—to the specializations.

(3) In exhibition-characterization, the tip of the solid inside blank triangle, ◬, is attached to the

exhibitor (the thing which exhibits the features), while the base is attached to the features (attributes and operations).

(4) In classification-instantiation, the tip of the solid circle inside a blank triangle, ◬, is attached to the thing class, while the base—to the thing instances.

The things which are the leaves of the hierarchy three, namely the parts, features, specializations, and instances, are collectively referred to as *refineables*, since they refine the ancestor, the root of the tree.

> ***Refineable*** *is a generalization of part, feature, specialization, and instance.*

Figures 2 through 5 present examples for each one of the four fundamental structural links, showing the root and the refineables for each link, and the corresponding OPL sentence in the figure legend.

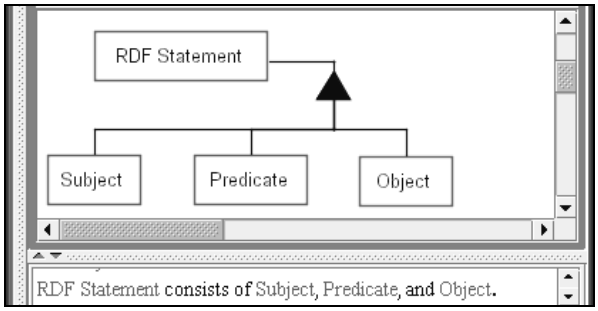


Figure 2. OPD of the sentence "**RDF Statement** consists of **Subject, Predicate,** and **Object.**"

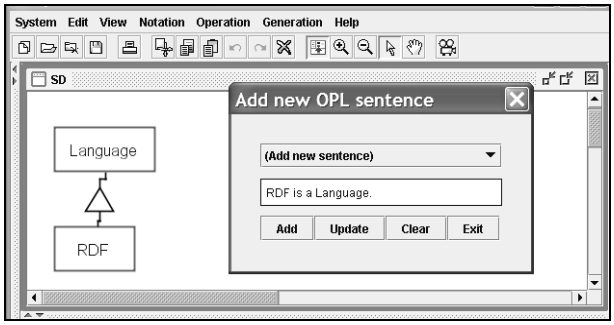


Figure 3. The OPD obtained by inputting into OPCAT the OPL sentence "**RDF** is a **Language.**"

10.5. OPM Behavior Modeling

Procedural links connect entities (objects, processes, and states) to express dynamic, time-dependent behavior of the system. Behavior, the dynamic aspect of a system, can be manifested in OPM in three ways:

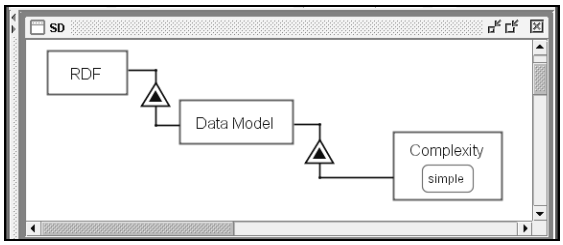


Figure 4. The OPD representing the sentence "RDF has a simple data model."

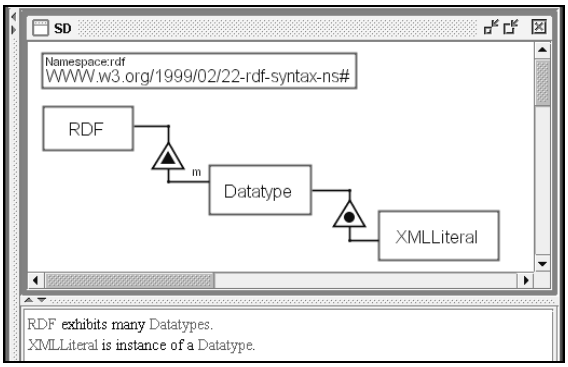


Figure 5 The OPM model of "**XMLLiteral** is instance of **Datatype** of **RDF**."

(1) A process can *transform* (generate, consume, or change the state of) objects,

(2) An object can *enable* a process without being transformed by it, and

(3) An object or a process can *trigger* an event that might, in turn, invoke a process if some conditions are met.

Accordingly, a procedural link can be a transformation link, an enabling link, or an event link.

10.5.1. Transformation links

A *transformation link* expresses how a process transforms one or more objects. The transformation of an object can be its consumption, generation, or state change. The transforming process is the *transformer*, while object that is being transformed is called *transformee*.

10.5.2. Input and output links

In Figure 6, **Cashing** is linked to the two states of **Check**: An input link leads from the initial **uncashed** state to **Cashing**, while an output link leads from **Cashing** to the final state **cashed**.

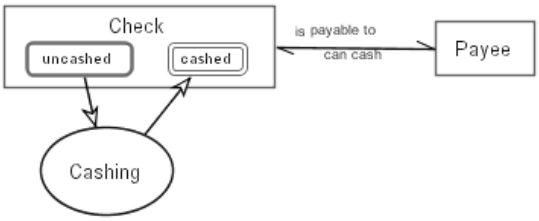


Figure 6. The **Cashing** process changes the state of **Check** from the **uncashed** to the **cashed** state.

The OPL sentence generated automatically by OPCAT as a result of adding these input and output links is:

Cashing changes **Check** from **uncased** to **cashed.**

10.5.3. Effect link

Sometimes we may not be interested in specifying the states of an object but still show that a process does affect an object by changing its state from some unspecified input state to another unspecified output state. To express this, we suppress (hide) the input and output states of the object, so the edges of the input and output links "migrate" to the contour of the object and coincide, yielding the effect link shown in Figure 7.

The OPL sentence that represents this graphic construct is:

Cashing affects **Check.**

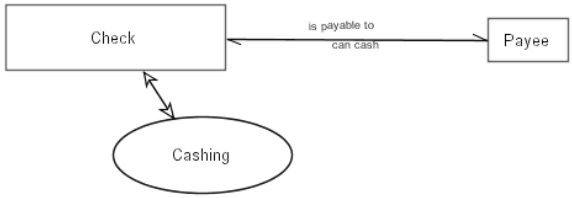


Figure 7. Suppressing the input and output states of **Check** cause the two link edges to migrate to the contour of **Check** and coincide, yielding the single bidirectional effect link between **Check** and **Cashing**.

We have seen that one type of object transformation is effect, in which a process changes the state of an object from some input state to another output state. When these two states are expressed (i.e., explicitly shown), then we can use the pair of input and output links to specify the source and destination states of the transformation. When the states are suppressed, we express the state change by the effect link, a more general and less informative transformation link.

10.5.4. Result and consumption links

State change is the least drastic transformation that an object can undergo. Two more extreme transformations are generation and consumption, denoted respectively by the result and consumption links. Generation is a transformation which causes an object, which had not existed prior to the process execution, to become existent. For example, as Figure 8 shows, **Check** is born as a result of a **Check Making** process.

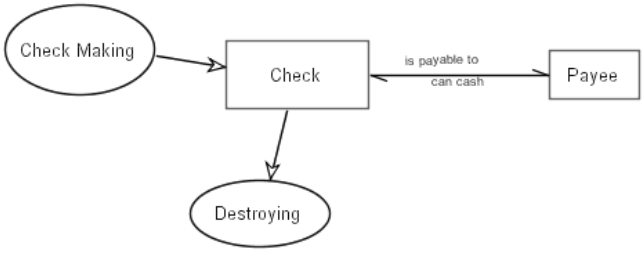


Figure 8. The object **Check** is consumed as a result of executing the **Destroying** process.

The result link is the arrow originating from the generating process and leading to the generated object. The OPL sentence that represents this graphic construct (shown also in Figure 8) is:

Check Making yields **Check.**

In contrast to generation, consumption is a transformation which causes an object, which had existed prior to the process execution, to become non-existent. For example, **Check** is consumed as a result of a **Destroying** process.

10.5.5. State-specified result and consumption links

We sometimes wish to be specific and state not only that an object is generated by a process, but also at what state that object is generated. Some other times, we might wish to be able to state not only that an object is consumed by a process, but also at what state that object has to be in order for it to be consumed by the process. As Figure 9 shows, the object **Check** is generated in its **unendorsed** state as a result of executing the process **Check Making**.

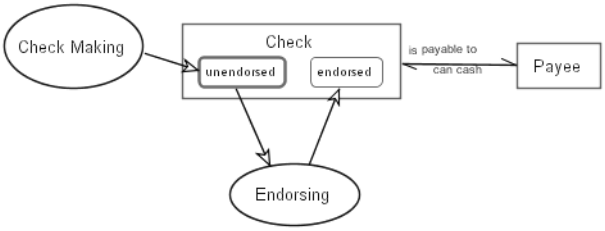


Figure 9. The object **Check** is generated in its **unendorsed** state as a result of executing the **Check Making** process.

The OPL sentence that represents this state-specified result link graphic construct (shown also in Figure 9) is:

Check Making yields **unendorsed Check.**

In comparison, the "regular," non-state-specified result link is the same, except that the (initial) state is not specified:

Check Making yields **Check.**

The difference is the addition of the state name (**unendorsed** in our case) before the name of the object (**Check**) that owns that state.

Analogously, a state-specified consumption link leads from a (final) state to the consuming process. For example, assuming a check can only be destroyed if it is cashed, Figure 10 shows the state-specified consumption link leading from the final state cashed of **Check** to the consuming process **Destroying**.

The OPL sentence that represents this state-specified consumption link graphic construct (shown also in Figure 10) is:

Destroying consumes **cashed Check.**

10.5.6. Enablers and enabling links

An *enabler* is an object that is required for a process to happen, but is not transformed by the process. An *enabling link* expresses the need for a (possibly state-specified) object to be present in order for the enabled process to occur. The enabled process does not transform the enabling object. An *agent* is a human enabler, while an *instrument* is a non-human one. The agent and instrument links are the first two in the procedural links group in Figure 1.

11. OPM Model of SOA-based Lifecycle

Having presented the main concepts of OPM, their graphic symbols and OPL syntax, we now turn back to the SOA lifecycle discussed in Section 6.

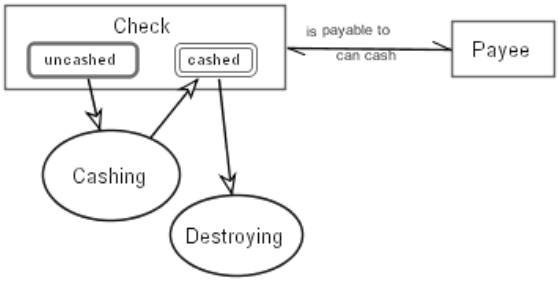


Figure 10. The object **Check** is consumed in its **cashed** state as a result of executing the **Destroying** process.

Figure 11 is an Object-Process Diagram (OPD) of a generic OPM model of SOA-based Lifecycle Development. Figure 12 is the Object-Process Language (OPL) paragraph of the SOA-based Lifecycle Development that was generated automatically by OPCAT. Note that both the OPD and the OPL faithfully and intuitively describe the exact model discussed in Section 7 – SOA-based Lifecycle Development.

In order to get a more vivid picture of our modeled system in action and capture design errors as soon as they are created, we have the option in OPCAT to run an animated simulation.

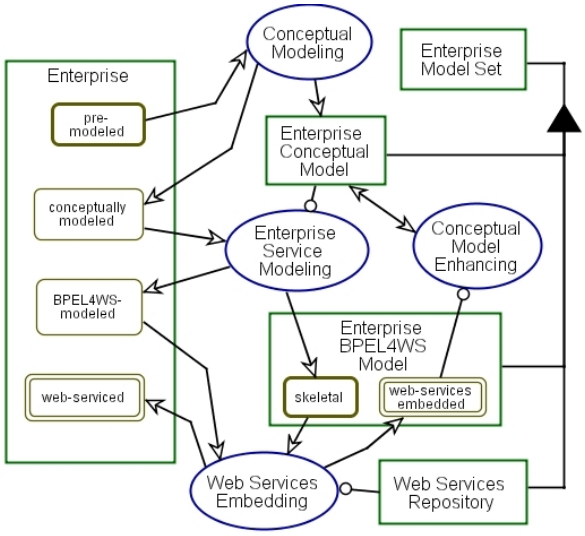


Figure 11. OPD of SOA-based Lifecycle Development.

Enterprise can be **pre- modeled**, **conceptually modeled**, **web-serviced**, **or** **BPEL4WS- modeled**.
pre- modeled is initial.
web-serviced is final.
Enterprise Model Set consists of **Enterprise Conceptual Model**, **Enterprise BPEL4WS Model**, and **Web Services Repository**.
Enterprise BPEL4WS Model can be **skeletal** or **web-services embedded**.
skeletal is initial.
web-services embedded is final.
Conceptual Modeling changes **Enterprise** from **pre-modeled** to **conceptually modeled**.
Conceptual Modeling yields **Enterprise Conceptual Model**.
Enterprise Service Modeling requires **Enterprise Conceptual Model**.
Enterprise Service Modeling changes **Enterprise** from **conceptually modeled** to **BPEL4WS- modeled**.
Enterprise Service Modeling yields **skeletal Enterprise BPEL4WS Model**.
Web Services Embedding requires **Web Services Repository**.
Web Services Embedding changes **Enterprise BPEL4WS Model** from **skeletal** to **web-services embedded** and **Enterprise** from **BPEL4WS- modeled** to **web-serviced**.
Conceptual Model Enhancing requires **web-services embedded Enterprise BPEL4WS Model**.
Conceptual Model Enhancing affects **Enterprise Conceptual Model**.

Figure 12. OPL of SOA-based Lifecycle Development.

Figure 13 shows the initial stage, where no process has yet been activated and the only existing objects are the **Enterprise** in its pre-modeled state and the **Web Services Repository**.

Figure 14 shows another snapshot of the animation, where **Enterprise Service Modeling** has already executed and generated the **skeletal Enterprise BPEL4WS Model**, while **Web Services Embedding**, now in action, is concurrently changing the **Enterprise** from **BPEL4WS-modeled** to **web-serviced** and **Enterprise BPEL4WS Model** from **skeletal** to **web-services embedded**.

More refined views on each one of these top-level processes, such as **Web Services Embedding**, can be specified within in-zoomed OPDs that OPCAT enables to create with a click of a button, maintaining the consistency across all the OPDs in the system's OPD set.

12. Summary and Future Work

It seems like the jury is already out on the outcomes of the fierce battle between OO and SOA, with SOA exiting with the upper hand. However, SOA needs a solid, well balanced methodological foundation in order to provide it with a holistic framework. Object-Process Methodology, which balances structure and behavior in a single bi-modal model is the ideal paradigm for this purpose. We have constructed the top-level diagram of an OPM SOA-based lifecycle engineering model. The combination of intuitive and formal graphics with the subset of English, generated on the fly, enables all the stakeholders involved to be on the same page with respect to the system's details at all levels.

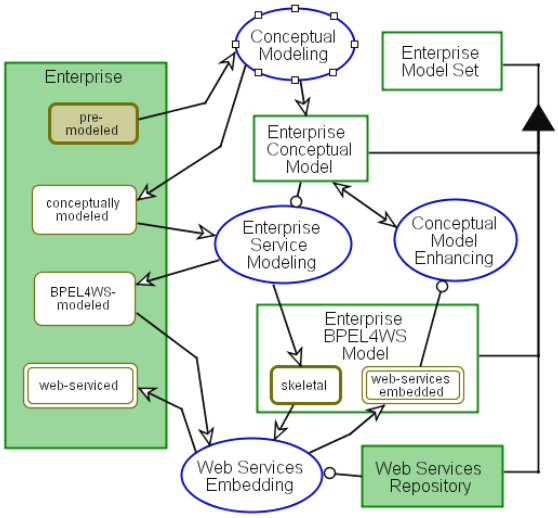


Figure 13. Animated simulation of SOA-based Lifecycle Development: The initial system state

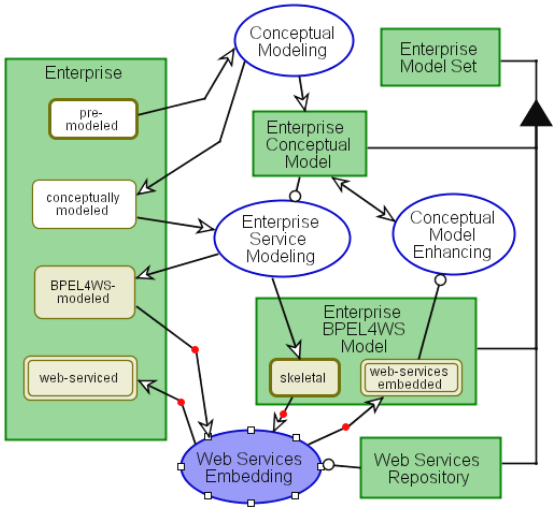


Figure 14. Animated simulation of SOA-based Lifecycle Development: Web Services Embedding in action

The main benefits of OPCAT's holistic modeling approach include improved communication between all involved stakeholders via OPCAT's intuitive yet comprehensive common graphic and textual language;

leveraging past investments while mitigating the risk of abrupt transformation to new architectures via step-by step implementation; avoiding duplications and enhancing reuse of the organization's assets; improving the utilization of legacy systems; and animated simulation of the entire system at the design level.

Following near future R&D work, OPCAT SOA Modeling Solution™ will enable the organization to clearly and concisely model its entire end-to-end service-based architecture based on Object-Process Methodology. The resulting platform-independent holistic model of the organization's Service Oriented Architecture will seamlessly integrate business processes, legacy systems, the corporate data warehouse, internal and external databases and services, and corporate policies, such as security, privacy, and access control, incorporating and hooking to the plethora of state-of-the-art SOA-related technologies surveyed in this paper.

The comprehensive system-of-systems model will serve as the underlying infrastructural blueprint for the entire enterprise and enable overseeing its operations, management, and control. It can be simulated and verified conceptually, exported and implemented by tools employing such standards as BPEL, J2EE, and .NET, while being maintained current and accurate via two-way communication with these deployment tools.

Acknowledgements

This work was partially supported by the Gordon Research Fund for Systems Engineering, Technion and by the European Networks of Excellence COCOON and TERREGOV.

References

[1] DMReview. Available http://www.dmreview.com/resources/glossary.cfm?keywordId=S, 2006.

[2] Service Oriented Enterprise. Available http://www.serviceoriented.org/service_oriented_enterprise.html, 2006.

[3] Business Process Execution Language for Web Services version 1.1 IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. Available http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, 2006.

[4] Kloppmann, M., Koenig D., Leymann, F. Pfau, G., Rickayzen, A., von Riegen, C., Schmidt, P., and Trickovic, I. WS-BPEL Extension for People – BPEL4People A Joint White Paper by IBM and SAP, July 2005. Available: http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/

[5] Dori, D. Object-Process Analysis: Maintaining the Balance between System Structure and Behavior. *Journal of Logic and Computation*, 5, 2, 1995, pp. 227-249.

[6] Dori, D. *Object-Process Methodology - A Holistic Systems Paradigm*, Springer Verlag, Berlin, Heidelberg, New York, 2002.

[7] Dori, D., Reinhartz-Berger, I., and Sturm, A. Developing Complex Systems with Object-Process Methodology using OPCAT. *Conceptual Modeling – ER 2003. Lecture Notes in Computer Science (2813)*, 2003, pp. 570-572.