Object-Process Methodology (OPM) vs. UML: A Code Generation Perspective

Iris Reinhartz-Berger[1] and Dov Dori[2]

[1] University of Haifa
Carmel Mountain, Haifa 31905, Israel
iris@mis.hevra.haifa.ac.il
[2] Technion, Israel Institute of Technology
Technion City, Haifa 32000, Israel
dori@ie.technion.ac.il

Abstract. Modeling languages have been evolving at a high pace, encouraging the use of automatic code generators for transforming models to programs. Automatic code generators should enable mechanical and repetitive coding operations to be performed quickly, reliably and uniformly, yielding higher productivity and quality of the developed systems. One way to evaluate modeling languages is to examine their code generation capabilities. In this paper, we compare the code generated from Rhapsody by I-Logix, a leading UML CASE tool, to the code generated from OPCAT, an Object-Process Methodology (OPM) CASE tool. We found that UML consistency problems and its distributed representation of system behavior are reflected in the code, yielding partial code that is mostly structure-oriented. OPM models, on the other hand, capture the static and dynamic aspects of a system in a single coherent view, enabling the generation of potentially complete application logic rather than just skeleton code. We explain and demonstrate the unique architecture and functionality of OPM-GCG—the generic code generator of OPCAT, which can handle dynamic repositories of translation rules from Object-Process Language, a constrained subset of English, to various target programming languages.

1 Introduction

Software engineering at large is ideally expected to cover all the phases of system development processes, including requirement elicitation, analysis, design, implementation, testing, and maintenance, as well as backtracking to earlier phases. The growth of formal visual design languages and methods during the last two decades has put the system analysis and design stages in the focus of leading software systems development processes. Using these languages, developers communicate with each other on the basis of a common ontology rather than via a specific programming language or technology. The Unified Modeling Language (UML) [12], for example, supports object-oriented concepts, such as classes, methods, and inheritance relations. These models can be translated to various (object-oriented or other) programming languages. This way, UML enables developing systems following the object-oriented paradigm

and applying the resultant designs to different kinds of implementation frameworks. Moreover, various parts of the same system can be translated into different target languages. For example, objects of a web application can be implemented as HTML documents, while processes of the same application can be written in Java or Java Script.

For quite long, the IT community has been viewing the ability to translate design models to source code as one of software engineering's Holy Grails. Indeed, automatic code generators are valuable in maintaining consistency and eliminating the gap between design models and their implementations. Automatic code generation increases the productivity and quality of the developed systems, enables mechanical and repetitive operations to be done quickly, reliably and uniformly, relieves designers from mundane tasks so they can focus on essence, and enforces programmers to write structured, legible code. The move towards automating code generation is in line with the industrial experience that the most complex task in creating a new system is modeling it at the semantic level and not in writing its detailed code.

Developing code generators is not a trivial task. First, a good code generator should narrow the gap between the design models and their implementation. This gap exists due to differences in the abstraction level and in the perspectives adopted in the design and implementation stages [1]. Second, a code generator should be flexible and applicable to various programming languages. Most of the existing code generators define rules for translating visual constructs to corresponding code blocks in a specific programming language. These language-specific rules are usually strict and reflect the insight and the style of the code generator implementers. Changing the translation rules or the visual constructs in these tools requires massive rewriting of their code generators.

In this paper, we compare the code generated from Rhapsody by I-Logix [8], a leading UML CASE tool which generates UML static views, as well as some dynamic views, to the code generated by OPCAT [3] from Object-Process Methodology (OPM) [2] models. While UML supports multiple views of the same system, OPM is a holistic approach that enables modeling the system structural, behavioral, functional, and architectural aspects in a single coherent framework. This single view approach of OPM at least potentially increases the consistency and integrity of the code generated for OPM models, because these models are inherently coherent. Furthermore, since OPM enables balanced modeling of system structure and behavior, the implementations generated from OPM models are closer to complete applications than just a skeleton of the system structure (i.e., class declarations, inheritance, method declarations, etc.). In addition to comparing Rhapsody and OPCAT in terms of their automatically-generated code, we elaborate on the structure and functionality of OPM-GCG, the generic code generator of OPCAT, which can handle dynamic repositories of translation rules to various programming languages.

The structure of the rest of the paper is as follows. Section 2 elaborates on existing code generators, their architectures, and shortcomings. Section 3 briefly summarizes main features of OPM and compares it to UML using a simple inventory ordering system. Section 4 presents OPM-GCG architecture and demonstrates its behavior on the ordering system, while Section 5 compares the completeness and simplicity of the Java code generated for the ordering system from OPCAT to that generated from

Rhapsody. Finally, Section 6 summarizes the main benefits and limitations of OPM in comparison to UML from a code generation perspective.

2 Code Generators: Related Work

Code generators in the software engineering domain, which read repository data and output source code in target programming languages, have become major components of CASE tools. Since UML became a standard modeling language in 1997, most of the code generators translate UML models to different programming languages, especially object-oriented ones. Early UML code generators translated only the system structure, i.e., class diagrams, to object-oriented programming languages. These tools produced only limited skeleton code and ignored system behavior. Harel and Gery [6] utilized UML class and state diagrams to generate both system structure and behavior. In order to apply this approach, state machines need to be adopted as models of object behavior even if the objects perform simple, insignificant behaviors [4].

Chow et al. [1] proposed a two-step approach to generate Java code from UML class, component, Statechart, sequence, and activity diagrams: (1) generating the static structure of the system from the class and component diagrams, and (2) generating system behavior from the Statechart, sequence, and activity diagrams. Although this approach handles the important views of UML, it assumes consistency of the UML model input and is specific to code generation in Java (or other similar object-oriented programming languages). As discussed in [20], maintaining consistency among UML views is not a trivial task.

Based on the subject-oriented approach, Harrison et al. [7] proposed another method for generating Java code from UML models. This method yields high-level skeletal implementation that shields implementers from low-level representation choices and details. The generated code imposes constraints on the acceptable designs and represents a certain redundancy in the generated implementation due to its strong static type checking requirements.

Commercial CASE tools usually hide the logical translation rules from the users. Their code generators are often hard-coded and rarely reconfigurable. Using Component Object Model (COM) [18], Rational Rose [15] offers code generators as plug-ins, enabling new destination languages to be added dynamically. While these generators handle packages, classes, interfaces, imports, inheritance relations, fields, methods, and modifiers, they do not handle behaviors. Rhapsody by I-Logix [8] does generate partial system behavior. Dividing the UML views into constructive and non-constructive ones, Rhapsody defines translation rules only for the constructive UML views [9]. The class and Statechart diagrams are considered constructive, while interaction diagrams are only partially constructive, as Rhapsody uses them to define objects, operations, and messages. However, the bodies of the operations must be modeled in Statecharts or hand-coded in a browser. Use case and activity diagrams are non-constructive; they only help analyze and document the system.

The Extensible Markup Language (XML) [11], which has emerged as the Internet's *lingua franca*, has been adopted also as a universal language for communicating be-

tween methods and translating models across various programming languages. Park and Kim [13] propose an XML rule-based code generator for UML. They first define an API for extracting necessary design model data from various UML repository formats. Code generation is then applied on the extracted model data using a rule descriptor and a corresponding rule interpreter. The rule descriptor is an XML document that defines how to translate model data to source code. The rule interpreter reads a mapping rule description, constructs a code generator object, and triggers code generation. As noted by its authors, this work concerns only class diagrams and should be extended to the other UML views. In addition to the consistency problem of UML views that this tool should overcome, using XML as the input language may be abhorrent and non-friendly to some users [17].

Some of the shortcomings of the code generators reviewed in this section are inherited from using UML as their modeling language. These include the consistency problem, the incompetence of the generated behavioral code, and the requirement to support code generation from different views. Using OPM as the underlying methodology for the code generator enables a unified, balanced representation of the system structure and behavior in a single view. Moreover, the three built-in abstraction-refinement mechanisms of OPM, which are described in the next section, guarantee that all the parts of an OPM system model are always consistent. Hence, the code generated from OPM models is more likely to be closer to the code of a complete, consistent application.

3 Object-Process Methodology (OPM)

Object-Process Methodology (OPM) [2] is a holistic approach to the study and development of systems. It integrates the object-oriented and process-oriented paradigms into a single frame of reference. Structure and behavior, the two major aspects that each system exhibits, co-exist in the same OPM view.

The elements of the OPM ontology are entities (things and states) and links. A *thing* is a generalization of an *object* and a *process* – the two basic building blocks of any system expressed in OPM. *Objects* are (physical or informatical) things that exist, while *processes* are things that transform objects. At any specific point in time, an object can be exactly in one *state*, and object states are changed through occurrences of processes. *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, while general tagged structural links (similar to UML associations) enable expressing any type of persistent relation. *Procedural links* connect entities (objects, processes, and states) to describe the behavior of a system: (1) processes can transform (generate, consume, or change the state of) objects; (2) objects can enable processes without being transformed by them; and (3) objects can trigger events that invoke processes.

3.1 The Bi-Modal Representation of OPM

Two semantically equivalent modalities, 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, and the OPD syntax specifies correct and consistent ways by which entities can be linked. The Object-Process Language (OPL) is the textual counterpart modality of the graphical OPD-set. 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 serves domain experts and system architects engaged in analyzing and designing a system. Designed also for machine interpretation through a well-defined set of production rules, OPL has an XML-based notation that provides a solid basis for automatically generating the designed application. This dual representation of OPM increases the processing capability of humans according to Mayer's cognitive theory [10].

3.2 OPM Refinement and Abstraction Mechanisms

Complexity management aims at balancing the tradeoff between two conflicting requirements: completeness and clarity. Completeness requires that the system details be stipulated to the fullest extent possible, while the need for clarity imposes an upper limit on the level of complexity and does not allow for an OPD (and a corresponding OPL paragraph) that is too cluttered or overloaded with entities and links among them. OPM defines three refinement-abstraction mechanisms that enable presenting the system at various detail levels without losing the comprehension of the system as a whole. These three mechanisms are: (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 enclosing 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 mechanisms, the achieved OPM models are and remain consistent.

3.3 Modeling the Inventory Ordering System with OPM and UML

To demonstrate the differences between OPM and UML, we present an OPM model and a UML model of an elementary inventory ordering system (Figures 1 and 3, respectively). The system handles orders of a single product type. For simplicity, each order reserves one product and the initial quantity of the product is 5.

As Figure 1(a) shows, the main process of the system, **Product Handling**, is activated by the **User**. Upon activation, **Product Handling** affects **Product** and **Customer** details and yields **Order** and either **Receipt** or **No Product Message**. Zooming into **Product Handling**, Figure 1(b) reveals its four sub-processes, which are executed

in an order determined by their vertical position. First, **Product Ordering** affects **Customer** and yields **Order** in its initial **ordered** state. Then, **Inventory Checking** checks if the **Product Quantity** is 0. If so (**Inventory Empty** is true), **Product Requesting** creates **No Product Message**. Otherwise, **Order Paying And Supplying** is activated. As shown in Figure 1(c), **Order Paying And Supplying** is done in two steps. First, **Order Paying** changes **Order** status from **ordered** to **paid**, creating the **Receipt** object. Then, **Order Supplying** decrements **Product Quantity** by 1 and changes **Order** status from **paid** to its final state, **supplied**.

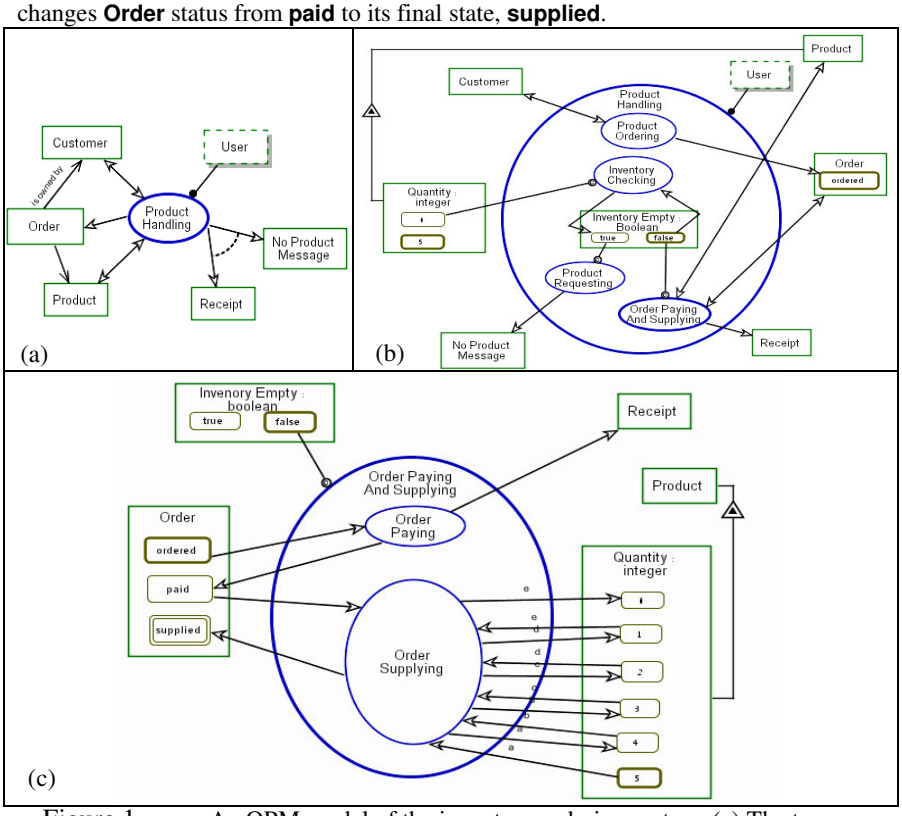

Figure 1. An OPM model of the inventory ordering system. (a) The top level System Diagram (SD). (b) SD1 in which Product Handling is in-zoomed. (c) SD1.1 in which Order Paying And Supplying is in-zoomed

Figure 2 is the semantically equivalent OPL script of the ordering system. This script is understandable to humans who are not familiar with visual design languages in general and OPM in particular. It also serves as documentation for the developed system and its XML representation is used as an input to OPM-GCG for code generation, as described in the sequel.

Order can be **ordered**, **paid**, or **supplied.**
Ordered is **initial.**
Supplied is **final.**
Order is owned by Customer.
Order relates to **Product.**
Product exhibits **Quantity.**
Quantity is of type integer.
User is environmental and physical.
User handles **Product Handling.**
Product Handling affects **Customer** and **Product.**
Product Handling yields **Order.**
Product Handling yields either **Receipt** or **No Product Message.**
Product Handling zooms into **Product Ordering**, **Inventory Checking**, **Product Requesting**, and **Order Paying And Supplying**, as well as **Inventory Empty.**
Inventory Empty is of type Boolean.
Inventory Empty is false by default.
Product Ordering affects **Customer.**
Product Ordering yields **ordered Order.**
Inventory Checking occurs if **Quantity** is **0.**
Inventory Checking changes **Inventory Empty** from false to true.

Product Requesting occurs if **Inventory Empty** is true.
Product Requesting yields **No Product Message.**
Order Paying And Supplying occurs if **Inventory Empty** is false.
Order Paying And Supplying affects **Product** and **Order.**
Order Paying And Supplying yields **Receipt.**
Order Paying And Supplying zooms into **Order Paying** and **Order Supplying.**
Order Paying changes **Order** from **ordered** to **paid.**
Order Paying yields **Receipt.**
Order Supplying changes **Order** from **paid** to **supplied.**
Following path **a**, **Order Supplying** changes **Quantity** from **5** to **4.**
Following path **b**, **Order Supplying** changes **Quantity** from **4** to **3.**
Following path **c**, **Order Supplying** changes **Quantity** from **3** to **2.**
Following path **d**, **Order Supplying** changes **Quantity** from **2** to **1.**
Following path **e**, **Order Supplying** changes **Quantity** from **1** to **0.**

Figure 2. The OPL script of the inventory ordering system

A UML model for the same system is presented in Figure 3. Figure 3(a) is a class diagram that represents the system structure. Figure 3(b) and Figure 3(c) are Statecharts representing the behavior of two object classes: order status and product quantity[1], respectively. Finally, Figure 3(d) is a sequence diagram that represents a typical scenario of product ordering by a user.

As these figures demonstrate, the object-oriented nature of UML requires breaking even this relatively small part of the system behavior into several pieces (methods) and then further decomposing them to sequences or Statecharts. In OPM, these behavior patterns are represented by stand-alone processes, which greatly simplify the system model.

[1] Product quantity states are labeled q0 through q5 to enable Rhapsody to generate legal Java variable names for them.

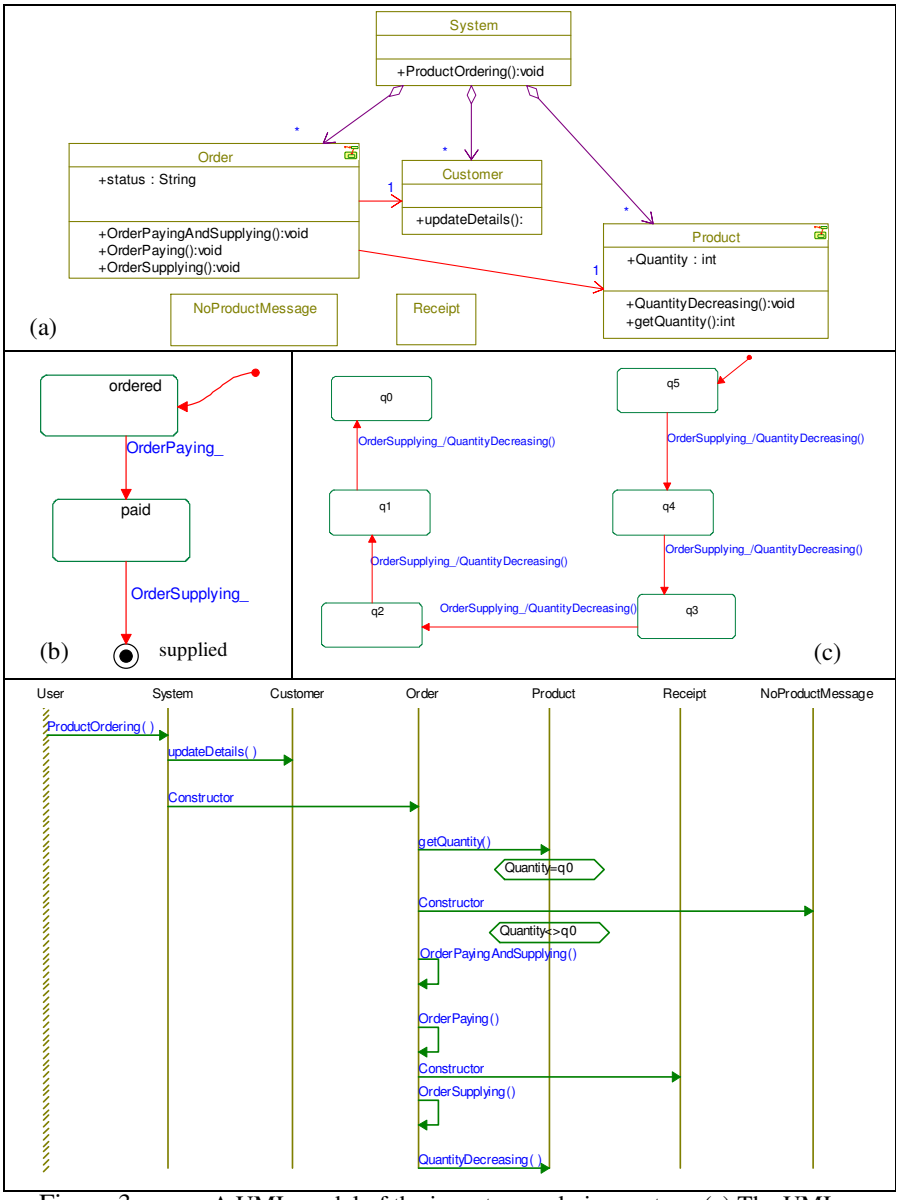

Figure 3. A UML model of the inventory ordering system. (a) The UML class diagram. (b) A Statechart diagram representing the behavior of an order. (c) A Statechart diagram representing the behavior of product quantity. (d) A Sequence diagram representing the behavior of ordering a product by a user

4 OPM-GCG Architecture and Functionality

As noted, OPM is supported by a CASE tool called Object-Process CASE Tool (OPCAT) [3]. An important component of OPCAT is OPM-GCG, the generic code generator. Figure 4 is the top level System Diagram (SD) of an OPM model of OPM-GCG, which describes its architecture and functionality in OPM. **OPM-GCG** consists of two parts: **OPCAT TIP** (Template for Implementation Programming) and **Implementation Generator**. **OPCAT TIP** exhibits the **Templates & Translations DB** and the process **OPCAT TIP Handling**. A **Super User**, i.e., a user with special authorization, inserts and updates translation rules into the **Templates & Translations DB** through **OPCAT TIP Handling**. This process also generates **Templates & Translations XML Files**, each of which contains XML-formatted OPL templates and their translations to a specific target programming language. These files serve as input for the **Implementation Generating** process of the **Implementation Generator**.

After the **System Designer** chooses a target programming language, **Implementation Generating** uses the corresponding **Templates & Translations XML File** along with the **System OPL-XML Script** in order to create the system **Implementation** (**User Interface**, **Code**, and **DB Schema**). The **System OPL-XML Script**, which is external to the OPM-GCG, stores the OPL script of a specific system, representing its OPM model in an XML format. This script is automatically generated in OPCAT, while the designer creates and improves an OPM model.

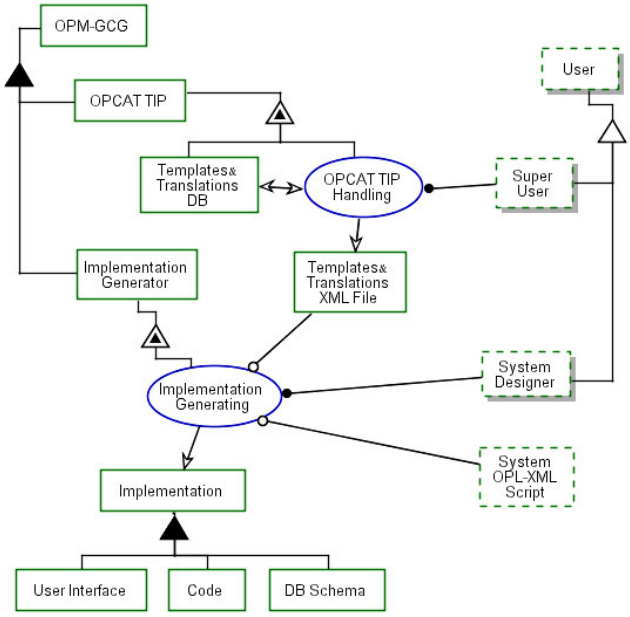

Figure 4. The top level SD of OPM-GCG

4.2 OPCAT TIP

The main screen of OPCAT TIP includes three tabs: OPL, XML, and Translations. In the OPL tab, shown in Figure 5, the OPL templates are described as being composed of sub-elements and characterized by attributes. For example, an object exhibition sentence, which describes the features (attributes and operations) of an object, has three constituents: *ObjectName*, which holds the sentence subject (object) name and is mandatory, *ExhibitedObject*, which is a template that appears once for each attribute of the object[2], and *Operation*, which is a string that appears once for each method of the object. The XML schema, which is automatically generated by OPCAT TIP for this template and appears in the XML tab, is:

```
<xs:element name="ObjectExhibitionSentence">
  <xs:complexType>
   <xs:sequence>
       <xs:element name = "ObjectName" type="xs:string"/>
       <xs:element ref="ExhibitedObject" minOccurs="0"
                   maxOccurs="unbounded"/>
       <xs:element name = "Operation" type="xs:string"
                   minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
</xs:element>
```

An example of an object exhibition sentence in the inventory ordering system is "**Product** exhibits **Quantity.**", i.e., a **Product** has an attribute named **Quantity**. The XML presentation of this particular sentence is:

```
<ObjectExhibitionSentence>
     <ObjectName>Product</ObjectName>
     <ExhibitedObject>
            <ObjectName>Quantity</ObjectName>
     </ExhibitedObject>
</ObjectExhibitionSentence>
```

In the Translations tab of OPCAT TIP, the selected OPL template is translated into programming or markup languages, such as Java or HTML. The translation is expressed by an ordered set of operations. Each operation may have a (possibly complex) condition and an action. Following the standard Event-Condition-Action (ECA) paradigm, when an instance of the OPL template is found in the **System OPL-XML Script** and the condition is satisfied, the action is executed. An example of such an operation can be carrying out the action *insertAtLocation(,,BEFORE_ENDING_TAG,MethodSection,,)* for each *Operation* of the *Object Exhibition Sentence,* if the object has methods, i.e., if `templateContains(Operation)` returns true. The supported functions in OPM-GCG are explained in the next section.

[2] The constituents of *ExhibitedObject* are: *Minimal Cardinality*, *Maximal Cardinality*, and *Object Name*.

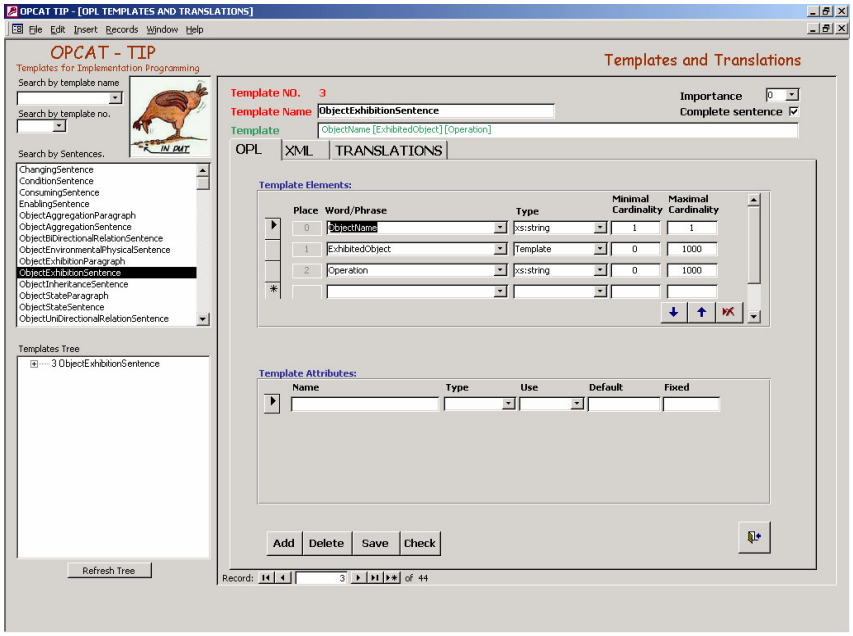

Figure 5. OPCAT TIP screen snapshot – the OPL tab

OPCAT TIP enables inserting new OPL templates and translations, updating existing OPL templates and translations, copying translations, copying references of translations, and checking the completeness of the translation rules. These capabilities enables dynamically defining OPL templates and translation rules without requiring deep knowledge of XML, as OPCAT TIP automatically generates XML files of the OPL templates and translations database.

4.2 Implementation Generator

The inputs for the implementation generator are (1) the target programming language (as specified by the designer), (2) the XML file of the OPL templates and translations for a specific language, and (3) the **System OPL-XML Script** (an XML file containing the OPL script of a system). When the implementation generator finds a match between an OPL sentence in the **System OPL-XML Script** and an OPL template in the **Templates & Translations XML File**, it executes the required operations in their specified order transforming the **System OPL-XML Script** into several XML files, each containing an XML format of a programming language file. Hence, conditions for executing OPCAT TIP operations are usually Boolean functions which check the existence of an element or an attribute in the input or output XML files. The operation actions insert, update, or remove elements or attributes from the output XML files. This way, the **System OPL-XML Script** needs to be read only once by the implementation generator, enabling several updates of different OPL sentences to the same code

block. Table 1 summarizes the conditions and actions currently supported by the implementation generator of OPM-GCG.

Table 1. Supported functions of OPM-GCG

Type	Function Signature	Function Description
Condition Functions	`none()`	Always true
	`templateContains (template t)`	The current OPL template contains t as its sub-element
	`translationContains (path p, file f, tagName tn, attributeName an, attributeValue av)`	The translation file f at path p contains <tn an="av"…>
	`equals (element e, value v)`	The value of the sub-element or sub-attribute e of the current OPL template equals v
	`Complex condition`	A combination of several atomic conditions using and, or, and not connectors
Action Functions	`createDirectory (path p)`	Creates a directory in path p
	`createFile (path p, file f, translation t)`	Creates a file named f at path p with the initial content of t
	`translate4all (path p, file f, template t)`	Translates all the sub-elements of type t of the current OPL template into file f at path p
	`replaceContent (path p, file f, tagName tn, attributeName an, attributeValue av, translation t)`	Replaces the content of <tn an="av"…> in file f at path p with the content of t
	`insertAtLocation (path p, file f, location l, tagName tn, attributeName an, attributeValue av, translation t)`	Inserts the content of t at the location l in respect to <tn an="av"…> in file f at path p. l can be one of BEFORE_STARTING_TAG, BEFORE_ENDING_TAG, AFTER_STARTING_TAG, AFTER_ENDING_TAG.

After creating the output code files in XML format, the implementation generator enables three possible options for handling the XML tags in the output files: (1) The XML tags can be left in order to support markup languages, such as HTML or WSDL (Web Service Description Language) [19], (2) The XML tags can be omitted to support regular programming languages (such as Java), and (3) The XML tags can become comments in the target programming language for debugging purposes. The implementation generator gets the **Super User** preference for handling tags in the output XML files from OPCAT TIP through the corresponding **Templates & Translations XML File**. If the **Super User** chooses to change the tags into comments, he or she is asked to supply the symbol of a single line comment and the starting and ending symbols of a multiple line comment in the target programming language.

4.3 Generating Java code for the Inventory Ordering System with OPM-GCG

Due to space limitation, Figure 6 and Figure 7 were chosen to demonstrate the Java code generated by OPM-GCG for a representative object (**Order**) and process (**Product Handling**), respectively.

```
// File Order.java
package OrderSystem;
import opmTypes.*;

public class Order extends opmObject {
   opmStatus theStatus;
   Customer theisownedbyCustomer;
   Product therelatestoProduct;

   public Order () {
        theStatus = new opmStatus();
               initializeStatus();
    }

    public void initializeStatus () {
        theStatus.addState (
        new opmState("ordered", true, false, true));
         theStatus.addState(
        new opmState("paid", true, false, false));
         theStatus.addState(
        new opmState("supplied", false, true, false));
         theStatus.addState(
        new opmState("cancelled", false, true, false));
     }

     public Customer gettheisownedbyCustomer() {
                return theisownedbyCustomer;
     }

     public void settheisownedbyCustomer (
        Customer newisownedbyCustomer) {
         theisownedbyCustomer= newisownedbyCustomer;
    }

    public Product gettherelatestoProduct() {
        return therelatestoProduct;
    }

    public void settherelatestoProduct (Product
        newrelatestoProduct) {
        therelatestoProduct= newrelatestoProduct;
    }
}
```

Figure 6. The OPM-GCG-generated code for **Order**

As these figures show, the system implementer can supply in addition to the translation rules a library of files for the target programming language. These general, system-independent files are influenced by the OPM approach. In Java, for example, there are six library classes, *opmObject*, *opmProcess*, *opmState*, *opmStatus*, *opmEvent*, and *opmEventQueue* which represent the core OPM elements. *OpmStatus*, for example, is a class that handles the different possible states of an object. It allows for

an object to be in exactly one state at any moment and controls the state entrance, state exit, and state change events of the object.

As shown in the **Product Handling** code (Figure 7), OPM-GCG generates not only the system structure, but also its behavior, including the body of the system processes (through the function *run*) and their preconditions (through the function *preConditionHolds*). This ability of OPM-GCG is inherited from OPM, which enables explicitly specifying the relations between system structure and behavior in the same model. These relations include process inputs, outputs, enablers, triggers, etc.

```
// File ProductHandling.java
package OrderSystem;
import opmTypes.*;

public class ProductHandling extends opmProcess {
  Boolean theInventoryEmpty;

  public ProductHandling () {
       theInventoryEmpty=new Boolean(false);
  }
  public boolean preConditionHolds () {
       boolean check = true;
       if (check) {
              }
              return check;
  }
  public void run (Customer theCustomer, Product
       theProduct, Order theOrder, Receipt theReceipt) {
       if (preConditionHolds ()) {
              // Effect theCustomer
              // Effect theProduct
              theOrder = new Order();
              theReceipt = new Receipt();
              ProductOrdering  theProductOrdering =
                     new ProductOrdering();
              theProductOrdering.run(theCustomer,
                     theOrder);
              InventoryChecking  theInventoryChecking =
                     new InventoryChecking();
              theInventoryChecking.run(
                     new Integer(
                   theProduct.gettheProductQuantity()),
                          theInventoryEmpty);
              OrderPayingAndSupplying
                     theOrderPayingAndSupplying =
                     new OrderPayingAndSupplying();
              theOrderPayingAndSupplying.run(
                          theInventoryEmpty, theOrder,
                     theProduct, theReceipt);
       }
  }
  public boolean gettheInventoryEmpty() {
       return theInventoryEmpty.booleanValue();
  }
  public void settheInventoryEmpty(boolean
       newInventoryEmpty) {
       theInventoryEmpty= new Boolean
       (newInventoryEmpty);
  }
}
```

Figure 7. The OPM-GCG-generated code for **Product Handling**

5 Comparison of OPCAT- and Rhapsody-Generated Codes for the Inventory Ordering System

To evaluate the relative value of OPM-GCG-generated code, we compare the code generated from OPM-GCG for the inventory ordering system (using the OPM model in Figure 2) to the code generated from Rhapsody by I-Logix using the UML model in Figure 3[3]. We have selected Rhapsody as the UML code generator for our comparison since to the best of our knowledge it is the only commercial tool that generates at least partial code for the system behavior.

Rhapsody uses a library of over 30 Java classes, only some of which can be mapped to UML concepts (e.g., `RiJState` and `RiJEvent`). Others, such as `RiJInformer`, `RiJOXF`, and `RiJThread`, are non-intuitive and seem to be part of the particular Rhapsody implementation of the Java code generator. For the inventory ordering system, Rhapsody generated nine classes: System, Customer, Order, Product, Receipt, User, NoProductMessage, OrderPaying_, and OrderSupplying_. Evidently, understanding the generated code requires deep understanding not only of the various UML views, but also of Rhapsody and its internal libraries. A system implementer who wishes to update this code, as is usually required in the development process of a system, needs to engage in studying the specific Rhapsody environment before managing to execute simple code update operations. These operations are required, for example, for generating the system behavior as expressed in interaction diagrams (e.g., in Figure 3(d)), which, in Rhapsody's terms, are only partially constructive. Under these conditions, implementers understandably quite often prefer to write code manually from scratch rather than putting an automatic code generator to work and editing the code it generated.

Size-wise, the OPM-CGC code generated for the ordering system contains 265 lines, compared with 739 lines for the corresponding Rhapsody code. In other words, the size of the Rhapsody-generated Java code was almost 3 times larger than the corresponding OPM-GCG-generated code for the same system. Having 3 times less code to study, inspect, and modify is certainly an advantage even before looking into other aspects such as clarity, generality, completeness, and environment-independence.

6 Summary and Future Work

The differences between OPM and UML are highly perceivable during the analysis and design stages. While UML is a multiple-view, object-oriented modeling language, OPM supports a single unifying structure-behavior view. These differences percolate also to the implementation stage through the different perspectives of the supporting code generators. In this paper, we presented OPM-GCG, an OPM-supporting generic code generator that translates OPM design models to various target programming

[3] The complete codes generated from Rhapsody and OPM-GCG for the inventory ordering system, along with the UML and OPM models, can be found at http://mis.hevra.haifa.ac.il/~iris/research/CodeGenerationData.zip.

languages. The translation rules of OPM-GCG are defined offline through a user-friendly tool and used by the implementation generator to create code files from the system OPL-XML file. This file, which is automatically created by OPCAT, contains the textual representation of the single, bimodal and multi-resolution OPM model in an XML format.

As demonstrated in this paper for a simple inventory ordering system, the Java code generated by OPM-GCG includes system behavior (processes, control flows, event triggers, etc.). Comparing this code to the code generated from Rhapsody by I-Logix for the analogous UML model of the same system, the OPM-GCG code appears to be simpler, more intuitive, easier to maintain and update, and more complete, while being almost three times as short as the code generated by Rhapsody.

While some of these differences are due to the specific code generation implementation of I-Logix, the crucial differences stem from the structure-oriented approach of UML, in which behavior is spread over six diagram types, a fact that inevitably invokes the model multiplicity problem [14]. OPM, on the other hand, enforces the development of consistent models through its three built-in refinement/abstraction mechanisms. These mechanisms define sets of rules that are checked whenever the abstraction level of an OPM model changes. The consistency of OPM models is completely and faithfully reflected in the resulting code, eliminating the infamous analysis-design-code gap that is responsible for many software project failures.

The main shortcoming of OPM-GCG is that it uses a modeling language (methodology) that is not (yet) standard. Although three independent empirical experiments that compared OPM to other languages (OMT [14], UML [16], and Arena [5]) have shown that OPM is better in modeling the dynamic and functional aspects of systems, sticking to standards is important and desirable in software engineering as in any other domain. To be compatible with the UML standard, OPCAT generates UML models from OPM models. We plan to also be able to convert a UML multi-view model to an OPM model. These capabilities enable the system designer to stay current with the standard, while using the most suitable approach for the system development task at hand.

We have just finished an empirical evaluation that examined the quality and completeness of the code produced by (1) OPM-GCG (2) Rhapsody and (3) extreme programming, and we are processing the results. In this experiment, the subjects were required to develop an automatic test checking system. In other future research on OPM-GCG power and scalability, we plan to generate large code segments of applications that combine structure and behavior in complex, intertwined ways from their OPM models in order to check the comprehensive and completeness of OPM-GCG-generated code.

References

1. K.O. Chow, W. Jia, V. Chan, and J. Cao. “Model-Based Generation of Java Code”, *International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA'2000)*, 2000. http://www.dvo.ru/bbc/pdpta/vol5/p522.pdf
2. D. Dori. *Object-Process Methodology - A Holistic Systems Paradigm*. Springer Verlag, New York, 2002.
3. D. Dori, I. Reinhartz-Berger, and A. Sturm. “OPCAT - A Bimodal CASE Tool for Object-Process Based System Development”, *Proceedings IEEE/ACM 5th International Conference on Enterprise Information Systems (ICEIS 2003)*, 2003, pp. 286-291. Software download site: http://www.objectprocess.org/
4. B.P. Doudlass. *Real-Time UML*. Addison-Wesley, 1998.
5. D. Gilat. "*A Framework for Simulation of Discrete Events Systems based on the Object-Process Methodology*". PhD Thesis, Technion – Israel Institute of Technology, 2003.
6. D. Harel and E. Gery. “Executable Object modeling with Statecharts”, *IEEE Computer*, 30 (7), 1997, pp. 31-42.
7. W. Harrison, C. Barton, and M. Raghavachari. “Mapping UML Designs to Java”, *Proceedings of the conference on Object-oriented programming, systems, languages, and applications(OOPSAL’00)*, 2000, pp. 178-187.
8. ILogix, Rhapsody overview, http://www.ilogix.com/products/rhapsody/index.cfm
9. ILogix, Rhapsody in C – Code Generation Guide, http://safariexamples.informit.com/0201699567/Rhapsody/Doc/Books/codegenc.pdf
10. R.E. Mayer. *Multimedia Learning*. Cambridge University Press, 2001.
11. Object Management Group (OMG). *Extensible Markup Language (XML)*, http://www.w3.org/XML/, 2002.
12. Object Management Group (OMG). *UML 1.4 - UML Semantics*. OMG document formal/01-09-73, 2001, http://cgi.omg.org/docs/formal/01-09-73.pdf
13. D.H. Park and S. D. Kim. "XML Rule Based Source Code Generator for UML CASE Tool". *8th Asia Pacific Software Engineering Conference (APSEC'01)*, 2001, pp. 53-60.
14. M. Peleg and D. Dori. “The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods”, *IEEE Transaction on Software Engineering*, 26 (8), 2000, pp. 742-759.
15. Rational Cooperation, Rational Rose, http://www.rational.com/products/rose/index.jsp
16. Reinhartz-Berger and D. Dori. "OPM vs. UML – Experimenting Comprehension and Construction of Web Application Models". Accepted to *Emprical Software Engineering journal*, 2004.
17. S. Sarkar and C. Cleaveland. “Code Generation Using XML Based Document Transformation”. Published on The Server Side – Your J2EE Community.
18. D. Stearns. “The Basics of Programming Model Design”, *Microsoft Coorperation*, 1998, http://msdn.microsoft.com/library/default.asp?url=/ library/en-us/dncomg/html/ msdn_basicpmd.asp
19. W3C Consortium. *Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl
20. Workshop on Consistency Problems in UML-based Software Development, 5th International Conference on the Unified Modeling Language - the Language and its applications (UML’2002), 2002. http://www.ipd.bth.se/uml2002/RR-2002-06.pdf