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 ILogix 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, which capture the static and dynamic aspects of a system in a single view, enable the generation of potentially complete application logic rather than just skeleton code. We also explain the unique architecture and functionality of OPM-GCG—the generic code generator of OPCAT.

1 Introduction

Software engineering at large is ideally expected to cover all the phases of system development processes. The growth of formal visual design languages and methods has put the system analysis and design stages in the focus of leading software 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) [10], for example, supports object-oriented concepts, such as classes, methods, and inheritance relations. The implementation of these concepts in different programming languages might vary. Moreover, various parts of the same system can be translated into different target languages.

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 also 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 in these tools requires massive rewriting of their code generators.

In this paper, we compare the code generated from Rhapsody by ILogix [7], 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. 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.

The structure of the rest of the paper is as follows. Section 2 elaborates on existing code generators and their shortcomings. Section 3 briefly summarizes 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 (mostly of UML models) and output source code in target programming languages, have become major components of CASE tools. 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 [5] 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: first the static structure of the system is generated and, then, system behavior is added. 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. As discussed in [16], maintaining consistency among UML views is not a trivial task.

Based on the subject-oriented approach, Harrison et al. [6] proposed a method for generating high-level skeletal Java code from UML models. 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. Using Component Object Model (COM) [15], Rational Rose [13] 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 ILogix [7] generates partial system behavior. Dividing the UML views into constructive and non-constructive, Rhapsody defines translation rules only for the constructive UML views. 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) [9] has been adopted also as a universal language for communicating between methods and translating models across various programming languages. Park and Kim [11] 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. 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 [14].

Some of the mentioned shortcomings of code generators 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. 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. Analogically, *links* can be structural or procedural. *Structural links* express static relations between pairs of entities, while *procedural links* connect entities (objects, processes, and states) to describe the behavior of a system.

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. Designed also for machine interpretation, OPL has an XML-based notation that provides a solid basis for automatically generating the designed application. This dual representation of OPM also increases the processing capability of humans according to Mayer's cognitive theory [8].

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 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, **Product Handling**, is activated by the **User**. Upon activation, **Product Handling** affects **Product** and **Customer** 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 their vertical position order. 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** first changes **Order** status from **ordered** to **paid** creating the **Receipt** object (**Order Paying**), and then decrements **Product Quantity** by 1 and changes **Order** status from **paid** to its final state, **supplied** (**Order Supplying**).

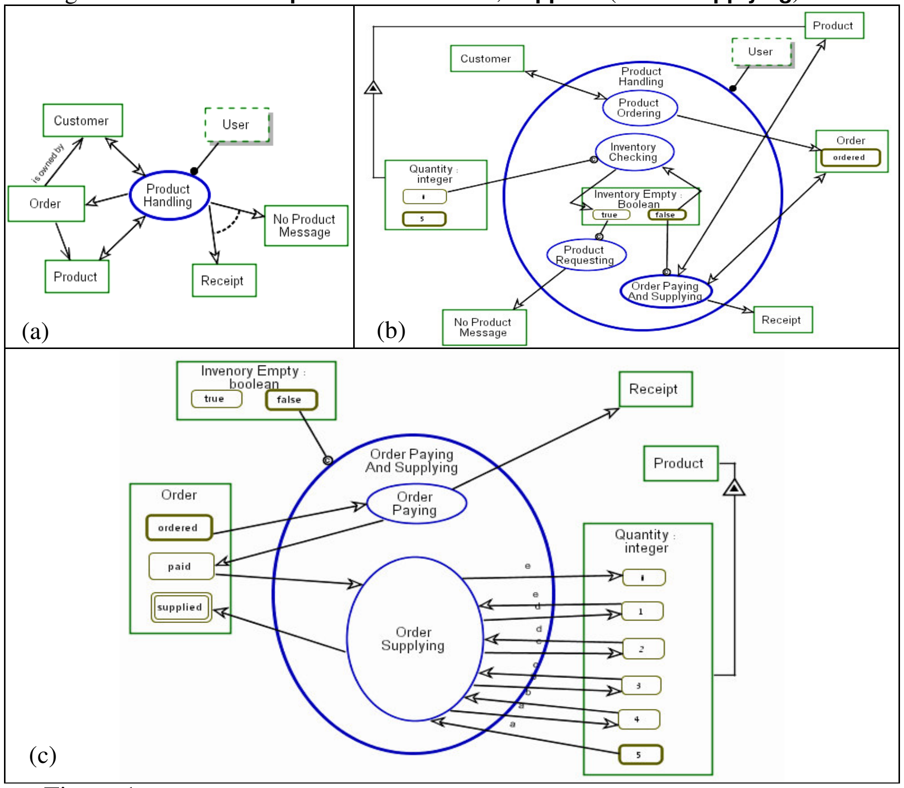

Figure 1. An OPM model of the inventory ordering system. (a) The top level System Diagram (SD). (b) SD1 in which Product Handling is in-zoomed. (c) SD1.1 in which Order Paying And Supplying is in-zoomed

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.

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 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 and consequently the system implementation (as we argue next).

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

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

generator. OPM-GCG consists of two parts: *OPCAT TIP (Template for Implementation Programming*) and *Implementation Generator*.

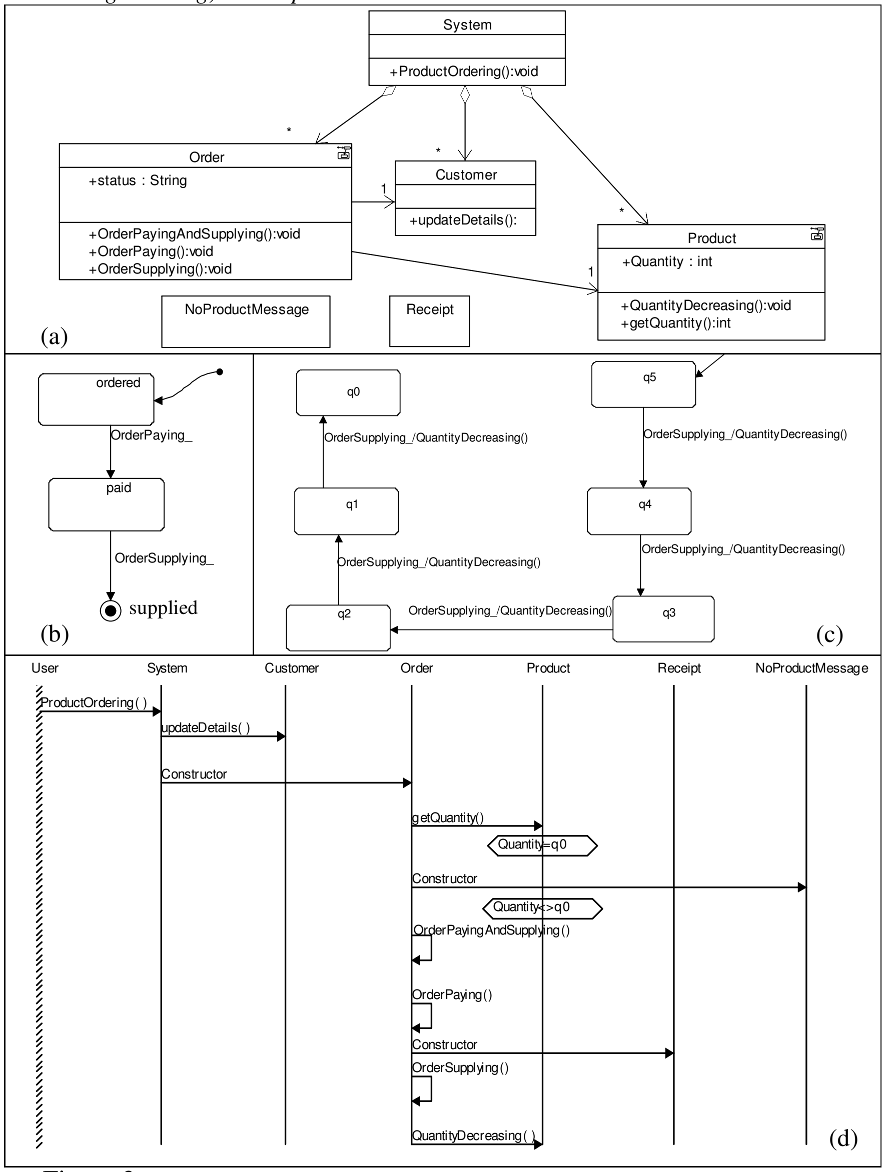

Figure 3. A UML model of the inventory ordering system. (a) The UML class diagram. (b) A Statechart diagram representing the behavior of an order. (c) A Statechart diagram representing the behavior of product quantity. (d) A Sequence diagram representing the behavior of ordering a product by a user

OPCAT TIP enables a user with special authorization to insert and update translation rules into a Templates & Translations DB. OPCAT TIP also automatically 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 Generator.

After the system designer chooses a target programming language, the Implementation Generator uses the corresponding Templates & Translations XML File along with the System OPL-XML Script in order to create the system Implementation (User Interface, Code, DB Schema, etc.). The System OPL-XML Script 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.

4.1 OPCAT TIP

The main screen of OPCAT TIP includes three tabs: OPL, XML, and Translations. In the OPL tab, 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>
```

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.

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

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. For example, *templateContains (template t)* checks if the current OPL template contains t as its sub-element, while *translationContains (path p, file f, tagName tn, attributeName an, attributeValue av)* checks if the translation file f at path p contains <tn an=”av”…>. The operation actions insert, update, or remove elements or attributes from the output XML files. For example, *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[3]. The supported functions enable several updates of different OPL sentences to the same code block, while reading the System OPL-XML Script only once by the implementation generator.

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, (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 user preference for handling tags in the output XML files from OPCAT TIP through the corresponding Templates & Translations XML File. If the 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.

[3] Location can be one of BEFORE_STARTING_TAG, BEFORE_ENDING_TAG, AFTER_STARTING_TAG, AFTER_ENDING_TAG.

4.3 An Example for Generating Java Code with OPM-GCG

Due to space limitation, Figure 4 was chosen to demonstrate the Java code generated by OPM-GCG for a representative process, **Product Handling**. As this figure shows, 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 ontology. In Java, for example, this library consists of *opmObject*, *opmProcess*, *opmState*, *opmStatus*, *opmEvent*, and *opmEventQueue* which represent the core OPM elements.

```
// 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 4. The OPM-GCG-generated code for Product Handling

As can be seen, 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 view. These relations include process inputs, outputs, enablers, triggers, etc.

5 Comparison of OPCAT- and Rhapsody-Generated Codes

To evaluate the relative value of OPM-GCG-generated code, we compare the code generated from OPM-GCG to the code generated from Rhapsody by I-Logix for the same inventory ordering system[4]. 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 and RiJOXF, are non-intuitive and seem to be part of the particular Rhapsody implementation of the Java code generator. 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, 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. 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 ILogix for an analogous UML model of the same system, the OPM-

[4] 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.

GCG code appears to be simpler, more intuitive, easier to maintain and update, and more complete.

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 [12]. In 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. We also plan to empirically evaluate the quality and complexity pf the code produced by OPM-GCG and leading UML code generators to the code which is written manually by programmers to the same systems.

References

1 Chow, K.O., Jia, W., Chan, V., Cao, J.: Model-Based Generation of Java Code. Int. Conf. on Parallel and Distributed Processing Techniques and Applications, 2000.

2 Dori, D.: Object-Process Methodology - A Holistic Systems Paradigm. Springer Verlag, New York, 2002.

3 Dori, D., Reinhartz-Berger, I., Sturm, A.: OPCAT - A Bimodal CASE Tool for Object-Process Based System Development. Int. Conf, on Enterprise Information Systems, 2003, pp. 286-291. Software download site: http://www.objectprocess.org/

4 Doudlass, B.P.: Real-Time UML. Addison-Wesley, 1998.

5 Harel, D., Gery, E.: Executable Object modeling with Statecharts. IEEE Computer, 30 (7), 1997, pp. 31-42.

6 Harrison, W., Barton, C., Raghavachari, M.: Mapping UML Designs to Java. Conf. on Object-oriented programming, systems, languages, and applications, 2000, pp. 178-187.

7 ILogix: Rhapsody in C – Code Generation Guide, http://safariexamples.informit.com/0201699567/Rhapsody/Doc/Books/codegenc.pdf

8 Mayer, R.E.: Multimedia Learning. Cambridge University Press, 2001.

9 Object Management Group (OMG): Extensible Markup Language (XML), http://www.w3.org/XML/, 2002.

10 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

11 Park, D.H., Kim, S. D.: XML Rule Based Source Code Generator for UML CASE Tool. 8th Asia Pacific Software Engineering Conference, 2001, pp. 53-60.

12 Peleg, M., Dori, D.: The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods. IEEE Tran. on Software Engineering, 26 (8), 2000, pp. 742-759.

13 Rational Cooperation: Rational Rose, http://www.rational.com/products/rose/index.jsp

14 Sarkar, S., Cleaveland, C.: Code Generation Using XML Based Document Transformation. Published on The Server Side – Your J2EE Community.

15 Stearns, D.: The Basics of Programming Model Design. Microsoft Coorperation, 1998.

16 Workshop on Consistency Problems in UML-based Software Development, 5th Int. Conf. on the Unified Modeling Language - the Language and its applications, 2002.