A Simple Static Model for Understanding the Dynamic Behavior of Programs

Pierre Kelsen*
Faculty of Sciences, Technology and Communication
University of Luxembourg
L-1359 Kirchberg
pierre.kelsen@uni.lu

Abstract

To maintain software and to adapt it to changing requirements, one needs to have a solid understanding of both its structure and behavior. While there are a number of reverse engineering tools that aid in understanding the runtime behavior of programs, these are mostly based on variants of UML sequence diagrams or statechart diagrams. We propose a new model for understanding runtime behavior that presents several advantages over the more traditional models: it has a simple syntax (a very small subset of UML). Second one can tailor the same type of model to different abstraction levels while maintaining traceability. Third, they have a clearly defined semantics that makes them executable. Finally they capture both static and dynamic aspects of a system.

We present three scenarios where our model - named EOP-model- aids with program comprehension: (1) high-level debugging, with which one can observe the execution of the program at a higher level of abstraction; (2) high-level program slicing, which allows to identify the data items in our model that may influence an operation, regardless of the abstraction level, and (3) generating abstract views at varying levels of detail while maintaining traceability between model elements and the code.

1. Introduction

For the purpose of maintaining software and for adapting it to changing requirements it is essential to have a good understanding of it. The understanding must extend to both static (or structural) aspects and dynamic aspects. Often the software is poorly documented. In these cases one needs to apply reverse engineering techniques ([2]) to model the software.

* Work supported by the Luxembourg Ministry of Culture, Higher Education and Research under grant IST/02/03.

Traditionally reverse engineering tools fall in two categories: static reverse engineering tools target the structural aspects of the program while dynamic reverse engineering tools examine the runtime behavior of the program. In each case a suitable model has to be chosen for holding the extracted information. In the static case these could be class diagrams, deployment diagrams or call graphs. The Rigi tool ([10]) is an example of a reverse engineering tool that uses a visualization of call graphs for describing the structure of a program. Dynamic models are often variations of UML sequence diagrams or statechart diagrams ([11]). Examples of reverse engineering tools using such diagrams are [12, 5, 8, 9].

In this paper we describe a new model - the EOP-model- for representing programs. Syntactically an EOP-model uses a very small subset of UML: indeed it can be viewed as an object diagram corresponding to a fixed class diagram which represents the metamodel. Our model has several interesting features compared to classical models such as statecharts or sequence diagrams:

- the model can represent both static and dynamic aspects of a program, although this paper concentrates on the dynamic aspects;
- the syntax is very simple;
- it has a well-defined semantics; an important consequence is that the model is executable;
- it can be tailored to arbitrary levels of abstraction while maintaining traceability (that is, each high-level artifact relates to lower-level artifacts).

We describe three scenarios in which EOP-models can contribute to program comprehension:

- High-level Debugging: one can build a high-level model that is executable, that is, the execution of the program can be observed at a higher level of abstraction.
- High-level program slicing: program slicing (first introduced by Mark Weiser [16]) is a technique that

Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)
1092-8138/04 $ 20.00 © 2004 IEEE

eliminates the parts of the program that are not relevant to a particular computation. It gives an answer to the question "What parts of the program affect the value of a variable in a given statement?". The EOP-models allow us to scale the concept of program slicing to higher levels of abstraction: by representing a program at such an abstract level we can easily compute which actions ("operations") depend on which data items ("properties").

- Generating abstract views: to understand a program it is essential to be able to view it at different levels of details. Although this can be done using UML diagrams different diagrams need to be used at different abstraction levels. For instance use cases are used to represent the operations the system offers to the user while sequence diagrams represent the sequence of messages that realize a given operation. With the complex syntax of many of these diagrams this adds up to a fairly steep learning curve if one wants to make effective use of these diagrams. Another problem is traceability: with UML-diagrams establishing the mapping of the artifacts to the code is often difficult. EOP-models offer a solution to both of these problems: despite their simplicity multiple EOP-models allow us to easily view the system at different levels of abstraction, from the use case view all the way down to the view consisting of individual program instructions. Furthermore there are clear rules that map model elements to the code, thus guaranteeing traceability.

The structure of this paper is as follows: in the next section we start by defining the basic model, first at the class level (subsection 2.1), then at the object level (subsection 2.2). In subsection 2.3 we define the semantics of the EOP-models. In subsection 2.4 we describe the relationship between the model and a concrete program. We then explain how to use the model in the context of program comprehension (subsection 2.5). The final section presents some concluding remarks.

2. A Static Model

We start by describing a metamodel in form of a class diagram. This metamodel defines the space of allowable models: each EOP-model will correspond to an object diagram for this class diagram. We term our model static because class diagrams and object diagrams are traditionally used to represent structural or static aspects of the system under consideration.

2.1. The Metamodel

The model comprises three types of entities: events, operations and properties.

- **Operations:** they represent actions taken by the system that may modify the state of the system and may also return a result. In real programs operations will be represented by methods/functions/procedures or by pieces of code.
- **Events**: these are occurrences that may enable ("trigger") operations.
- **Properties**: they represent the data items manipulated by operations. An operation may *require* a property as input or it may *affect* a property by setting its value.

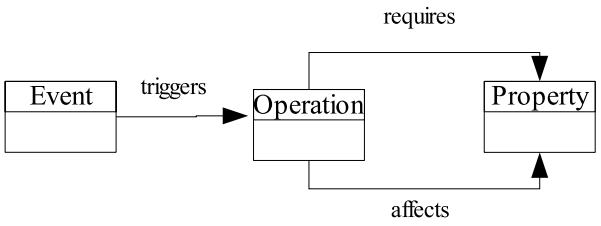

Figure 1. A class diagram for the basic model

We can represent the metamodel described so far by a class diagram (see figure 1): there are three classes, representing events, operations and properties, and three relations *triggers*, *requires* and *affects*.

2.2. The EOP-Model

To model the behavior of a particular system, we need to instantiate the metamodel, that is, we must produce an object diagram that conforms to the class diagram. In figure 2 we present an example of such a model.

We call such an object diagram an **EOP-model** (named after **e**vents, **o**perations, **p**roperties). Thus, each EOP-model is an instantiation of the metamodel and it describes events, properties and operations for a particular system.

2.3. Semantics of EOP-Models

So far our discussion has centered on the structural aspects of the model. If we want to apply these models to real systems, we need to define the semantics of the model precisely.

We first give an informal description of the semantics. We may view a model M as a machine that reacts when one of its events is fired. In that case each operation triggered

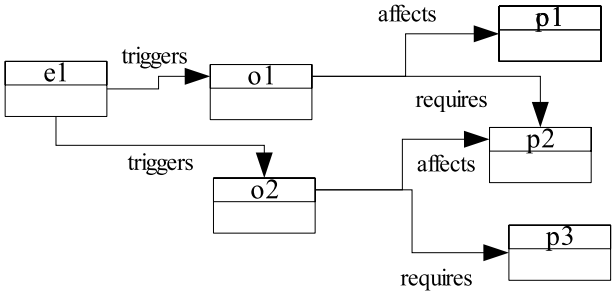

Figure 2. An object diagram for a concrete model.

by this event monitors its required properties. The value of a property is either defined or undefined (no value) when an event is fired. Once the values of all its required properties are defined, the operation can be executed, meaning that, based on the values for the required properties, it computes new values for the affected properties and sets these properties to the new values. After this has been done the operation becomes inactive. The machine "stops" when no more operations can be executed. The whole process starts all over again when another event is fired.

Now we take a more formal approach to defining the behavior of an EOP-model M. Let $E(M)$, $O(M)$ and $P(M)$ denote the set of events, operations and properties of M. In the above example $E(M) = \{e1\}$, $O(M) = \{o1, o2\}$and $P(M) = \{p1, p2, p3\}$. As we said earlier we assume that each property can be either undefined or have a defined value. To simplify the discussion, let $P \cup \{\perp\}$be a common domain for all property values. The value $\perp$(not in P) denotes the undefined value of a property. We may view an operation as a mapping that maps defined values for the required properties to new values for the affected properties. Let o be an operation that requires q properties and that affects r properties. Then o can be interpreted as a function from P^q to P^r. Note that we assume that a property value, once defined, cannot become undefined because of an operation. We call this mapping the *local transformation function for operation o* and denote it by f_o. Note that the model does not specify what the transformation function for each operation is.

Fix a local transformation function for each operation of M. At the outset all operations in the model are *disabled*. For something to happen an event needs to be fired. Once an event is fired, all operations that are triggered by this event are *enabled*. An enabled operation enters the *executable* state once all of its required properties have been set to a defined value. An executable operation can be executed. The effect of executing an operation is to set each affected property to the value computed by the local transformation function, taking as arguments the defined value of each required property (at the moment the operation starts executing). After the operation terminates it enters the disabled state. The state transitions for an operation are illustrated in figure 3.

When an event is fired a sequence of operations is executed. We call such an execution sequence *valid* if

1. every operation that is executed is in the executable state,
2. at the end of the sequence there are no more executable operations

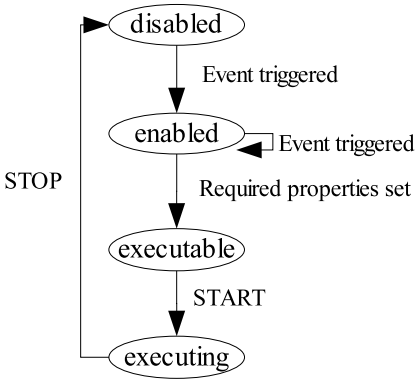

Figure 3. The state diagram for an operation

Note that new events may be generated when a property is set to a new value. Thus termination of the process is not guaranteed. If it terminates, however, the final values of the properties depend on the event which is fed into the model, on the initial property values, on the local transformation functions, and on the order in which the operations in the valid execution sequence are executed.

2.4. From Programs to EOP-Models

We have defined the semantics of the model independently of a particular system. Our goal is of course to use our EOP-model to represent the behavior of a program. In order to do this we need to elaborate on the relationship between an EOP-model and a particular program.

The idea is to define sets of events, operations and properties as well as the relations *triggers*, *requires* and *affects* in the context of a program. In other words one constructs an EOP-model in the context of a concrete program. We call this a *concrete EOP-model.* The following conditions should be satisfied by any concrete EOP-model:

- When an event is fired, the resulting sequence of operations executed by the program is a valid execution sequence for the EOP-model.
- When an operation is executed by the program, it is in an executable state.
- When an operation is executed, the program sets the affected properties to values determined by the values of its required properties.

These three conditions must hold over all possible runs of the program. The intention is that this ensures that the model faithfully represents the program. As an example consider a Java program. Define

- an event to be the execution of a particular statement;
- an operation to be a sequence of Java statements;
- a property to be a local variable or a field of the class containing the method.

This indicates one class of concrete EOP-models. Different concrete models within this class (using coarse-grained operations versus using fine-grained operations for instance) allow one to view the program at different levels of abstraction. In fact one notable strength of our approach is that our model can be tailored to any level of abstraction. This will become clearer in the following section.

2.5. Using EOP-Models

In this section we describe several ways of using the models to enhance program comprehension. All our examples will refer to a piece of example code given in figure 4. This code is a small excerpt of event handling code in a software written by a second-year undergraduate student at our university.

2.5.1. High-level debugging Classical debugging operates at the level of lines of code. Even though it is a very useful tool in correcting bugs, it is only of limited use if one tries to get a high-level view of what the program does.This is where our model can be helpful: one can choose a higher level of abstraction and still allow a dynamic simulation of the program's behavior at that level. We illustrate this with the sample code of figure 4.

First we need to define an EOP-model. We define a single event *loadEvent* that is fired when the first line of this code segment is executed.

Define the following operations:

```
1:if(e.getActionCommand().equals("Load")) {
2:  if (Model.getModel().isModified()) {
3:    if (mp == null) {
4:          int v = JOptionPane.showConfirmDialog(null,
5:             "There are unsaved modifications.\nLose them?",
6:             "Really Load?", JOptionPane.YES_NO_OPTION);
7:          if (v == JOptionPane.NO_OPTION)
8:            return;
9:     }
10:    else {
11:      mp.setModel(null);
12:      mp = null;
13:    }
14:  }
15:
16:  JFileChooser jfc = new JFileChooser();
17:  if (docFile == null) {
18:    if (lastPath != null)
19:      jfc.setCurrentDirectory(new File(lastPath));
20:  }
21:
22:  if (jfc.showOpenDialog(this)==
23:                JFileChooser.APPROVE_OPTION) {
24:    docFile = jfc.getSelectedFile();
25:    XmlModelPersistor xmls = new XmlModelPersistor();
26:    xmls.setFile(docFile);
27:    lastPath = docFile.getPath();
28:    Model.getModel().dataLoad(xmls);
29:    reloadTypeFilterComboBox();
30:    Model.getModel().clearModified();
31:    Application.setTitleFilename(docFile.getAbsolutePath());
32:  }
33:}
```

Figure 4. Sample code

- the if-statement on lines 2-14 constitutes operation *treatModifiedModel*;
- the statement *jfc.showOpenDialog(this)* on line 22 makes up operation *askFile*;
- the body of the if-statement on lines 22-23, that is, lines 24-31, constitutes operation *loadModel*.

We note that operation *treatModifiedModel* may affect the variable *mp* if *Model.getModel().isModified()* returns true. Therefore we define two properties *mp* and *modelModified* that interact with operation *treatModifiedModel*: property *mp* corresponds to the variable of the same name, and property *modelModified* represents *Model.getModel().isModified()*.

Similarly we define properties *selectedFile* (which is initially undefined) , *docFile* and *lastPath* that interact with operations *askFile* and *loadModel*, respectively. The resulting model, based on inspection of the sample code, is given in figure 5.

A tool for debugging the model could work as follows. Run the program. When the loadEvent is triggered (i.e., the corresponding line of code is executed) the program starts to interact with the model. The first operation executed in the

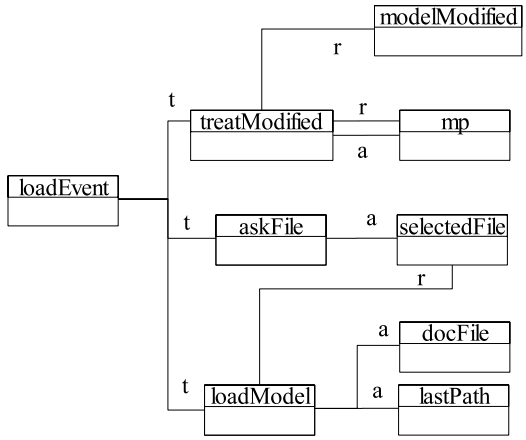

Figure 5. Model for the sample code; "t", "r" and "a" represent *triggers, requires* and *affects* links. Links are directed from left to right.

program will be highlighted in the model. Similarly for the second and third operation. At each step the user can stop the simulation and inspect the values of the properties. This way he can visually observe the effect of the various operations on the properties.

We close by commenting on related work. The idea of making models executable is not new. We mention two specification languages that support executable models. SDL([4]) is a graphical language that is formal and object-oriented. Because of its precise semantics SDL models can be simulated or used as a basis for code generation using appropriate tools. The UML, on the other hand, lacks precise and formal foundations for several constructs that hinders the development of executable UML models. Recently there has been some effort to transform UML into an executable language [13] by instrumenting it with a precise Action Semantics [1]. Our approach is quite different: rather than attempting to make our model more precise by opting for a richer language (such as UML or SDL) we choose to keep the graphical language very simple. The model only describes certain aspects of the system while the rest of the behavior is defined at the code level.

2.5.2. High-Level Program Slicing The program slicing technique ([16]) is a program analysis technique that tries to discover the parts of the program that may influence a value computed at some point of interest. It is typically used if an erroneous value is computed at some point. In this case it can help in detecting the "culprit" statement that is responsible for producing the wrong value.

The original slicing technique introduced by Weiser is also called *static slicing* because it computes the program slice irrespectively of actual input values. If we take these input values into account smaller program slices can in general be computed. This technique of *dynamic slicing* was first introduced by Korel and Laski ([7]). Hybrid approaches relying on a combination of static and dynamic slicing are presented in [3, 6, 15]. See [14] for an overview of program slicing techniques.

Here we carry over the idea of static program slicing to EOP-models. Indeed it is natural to ask the question what operations can influence a required property when it is used by a given operation. In order to compute this we need to assume two things: first, if an operation does not affect a property then it does indeed not modify it. Second, we are given an execution sequence for the operations. For the model in figure 5, for instance, we may assume that operations treatModified, askFile and loadModel are executed in that order (see sample code).

Under these two assumptions there is a simple algorithm that computes all the operations that may influence a given property used by an operation o: initialize an empty set of operations. Start by adding all operations preceding o that affect this property. For any operation in the set, add all operations that precede o and that affect a required property of this operation. Repeat this procedure until no new operations are found. This algorithm can be implemented in linear time (linear in the size of the model) using standard techniques.

Since the model may be situated at a high level of abstraction we may think of this analysis procedure as high-level program slicing.

2.5.3. Generating Abstract Views It is essential for program comprehension to be able to view the software at different levels of abstraction. This has been achieved partially using standard UML models like use case diagrams, class diagrams and sequence diagrams. The traditional approach has several problems however:

- often the relationship between high-level artifacts and low-level constructs (e.g., lines of code) cannot be traced;
- the semantic of the underlying models is not always clearly defined; an immediate consequence of this is that these models are not executable;
- different types of diagrams are used at different levels of abstraction, implying a steep learning curve.

EOP-models have a very simple syntax and clearly defined semantics. Furthermore they can describe the systems at different levels of abstraction. We illustrate this by showing how one can describe a system at a use case level using an EOP-model. This model is to be contrasted with that discussed for the sample code which is more low-level.

We can view the system as a black box that offers a set of operations that are triggered by events. As an example consider an online book shop. It may offer operations for registering as a new user, for logging in, for placing an order, for canceling an order and so on. The corresponding EOP-model would simply take up the system operations as operations of the model. It would have one event per system operation. E.g., the event loginEvent would trigger the login operation. This operation may require properties *username* and *password* and affect the property *loggedUsers*. Similarly the event *submitEvent* may trigger the *submitOrder* operation that requires the *orderedItems* property (essentially a list of books) and that affects the *ordersPlaced* property.

Generally speaking our model facilitates abstract representations since it removes certain details, namely the event firing and capture method: in real programs this may take quite a bit of machinery (registering listeners, removing listeners, iterating over list of listeners) to implement, thus obscuring the more important part. It also hides the internal details of the operation: operations are only known by what properties they require and what properties they affect; the details of their implementation are not part of the model.

3. Conclusion

In this paper we have presented EOP-models as a new approach to understanding the dynamic behavior of a program. EOP-model are characterized by a simple syntax and a well-defined semantics. Different EOP-models can be used to view a program at different levels of abstraction. Thanks to the precise semantics high-level models are executable, meaning that one can observe the execution of the program at the level of abstraction provided by the model.

Much work remains to be done. In the context of program comprehension the development of suitable tools needs to be taken up: indeed only by using tools on programs of realistic complexity can the usefulness of our model be conclusively validated. Another interesting avenue of research is use of EOP-models in the analysis and design process. Because of their simple syntax and precise semantics they should be helpful in refining a program from the specification level down to the implementation level. These questions are the subject of ongoing research.

Acknowledgment. Many thanks to Christian Glodt and Emil Weydert for fruitful discussions on this topic.

References

[1] Alcatel, I-Logix, Kennedy-Carter, Inc. Kabira Technologies, Inc. Project Technology, Rational Software Corporation, and Telelogic AB. Action semnatics for the uml. In *Document ad/2001-03-01. OMG*, 2000.

[2] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.

[3] Jong-Deok Choi, Barton P. Miller, and Robert H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[4] ITU-T. *Specification and Description Language(SDL). Recommendation Z.100.*, 11/99.

[5] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*. IEEE Computer Society Press Los Alamitos California, 1997.

[6] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the Conference on Software Maintenance*, pages 386–395. IEEE Computer Society Press, 1993.

[7] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[8] Kai Koskimies and Hanspeter Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proceedings of the 18th International Conference on Software Engineering*, pages 366–375. IEEE Computer Society Press / ACM Press, 1996.

[9] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visulization. *IEEE Computer*, 30(5):63–70, May 1997.

[10] Hausi A. Müller, Kenny Wong, and Scott R. Tilley. Understanding software systems using reverse engineering technology. In V. S. Alagar and R. Missaoui, editors, *Object-Oriented Technology for Database and Software Systems*, pages 240–252. World Scientific, 1995.

[11] Object Management Group. *Unified Modeling Language*, March 2003. statut : Version 1.5 , http://www.omg.org/docs/formal/03-03-01.pdf.

[12] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of Java programs. *Lecture Notes in Computer Science*, 2269:151–162, 2002.

[13] C. Raistrick, I. Wilkie, and C. Carter. Executable uml (xuml). In *Proceedings 3rd International Conference on the Unified Modeling Language UML*, 2000.

[14] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. Surveys the state-of-the-art in program slicing and gives many references to the literature.

[15] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 107–119, 1991.

[16] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, March 1981.