IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4784

End User Monitoring: A Case for Linking Static and Dynamic Models

Subhasish Mazumdar
Computer Science Department
New Mexico Institute of Mining and Technology
Socorro, NM 87801 USA
mazumdar@nmt.edu
Voice: +1 505 835 5288
Fax: +1 505 835 5587

Wei Hu
Agilent Technologies, Inc.
Colorado Springs, Colorado, USA
wei_hu@agilent.com
Voice: +1 719 531 4531
Fax: +1 719 522 8044

ABSTRACT

End users are helpless with opaque software systems that are complex, error-prone, and equipped with incorrect and/or misleading documentation. Our approach is to allow end-users to monitor software systems; in order to tackle their formidable complexity, systems are built for on-line monitoring by basing them on an open conceptual model whose abstraction provides comprehension to the end-user. While prototyping this approach, we have found that it is not enough to have a static and dynamic conceptual model: they need to be linked. The realization of such a link can be very simple and yet very useful.

INTRODUCTION

End-users today are forced to trust software systems that they have very little control over, are hard to extend, work in unexpected ways, and do not conform to documented features while leaving important features undocumented [16]. As Dijkstra says, "... The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive worldwide distribution of bug-ridden software for which we should be deeply ashamed" [9]. Although software has recently become a consumer product world-wide, the customer's desire for quality has not translated into software reliability and integrity. Perhaps because of their awe of the product's complexity and opaqueness, consumers have accepted this state of affairs. Interestingly, these circumstances have co-existed with impressive developments in both formal methods [3, 12, 20] and practical aspects of software development (e.g., the Cleanroom method [17]). We have analyzed this situation and offered our approach [16]: build the software system so that it can be *monitored* by end-users, the motive being to empower the end-users with insight into the system.

Software execution can be monitored only if it is understood, and for us, understood by end-users. But how does one understand software? The *state* of a complex software system is typically defined as a snapshot of the values of the variables occurring in the code. But such a state is meaningless to the end-user. Furthermore, one needs a large sequence of such states to provide a record of an execution instance or a part thereof.

Conceptual modeling is fundamental to computing [5] because it lets us cope with the complexities of the real world that all software systems must interface with. Conceptual models have for long been used by database designers [1] who not only base their design on it (it is called it a *semantic model* and is typically an entity-relationship [6] schema) but also *retain it*: a database system keeps around a (derived) schema or meta-data to be queried at run-time just like ordinary data. Our approach takes a leaf out of their book by making the conceptual model of a software system an integral component of the final system. When viewed in the context of a conceptual model, we believe, the "incomprehensible" state becomes understandable to all. The conceptual model therefore, becomes the key to the end-user in understanding the software and tracking its execution.

To realize our approach, we require three changes. First, the software lifecycle has to be modified in order to incorporate a conceptual model at the requirement analysis, design, and implementation phases so as to finally allow the model to "live" as a part of the final product that it captures. If a method such as Cleanroom is followed, then checking the model against the code would be an added responsibility of the peer-review process. Second, the final system is to be augmented by an underlying database with query support. Third, during development, probes are to be inserted in the software so as to capture certain data during execution and send it to the underlying database. The end user can query both the conceptual model and the probed run-time data to obtain insight into the system and its actual behavior. Automatic enforcement of database integrity constraints can provide an alerting mechanism.

We looked for a conceptual modeling language that emphasized the static and dynamic aspects of the world and was suitable for software systems. We ruled out approaches such as SASD [8] in which entities of the real world are not clearly described. We chose UML (Unified Modeling Language) [11, 15, 18] for the following reasons. First, while chiefly used for modeling object-oriented systems, it has been claimed that UML can be used to specify, visualize, construct, and document *all* artifacts of software systems. Second, UML presents a fair amount of maturity since it represents a convergence of several analysis and design notations ([2, 19, 14]). Third, UML has proven successful in the modeling of large and complex systems and is becoming the lingua franca for software design and development; this reduces the learning overhead involved in following our approach. Fourth, it is independent of specific programming languages and development processes. Finally, the lack of formal semantics is not crucial in our case since we are using conceptual models at the highest levels of abstraction where we interface with the messy external world. Note that our choice was not influenced by questions regarding the mode of presentation of the conceptual model to the end user. We have prototyped our approach by reverse-engineering and rebuilding four software systems [13], the first two being application programs while the last two run in kernel mode under Linux.

In this paper, we report our observation about the adequacy of UML for our needs: while UML does provide the appropriate mechanisms and level of abstraction, there is a need for more than a static and a dynamic model. We suggest a data structure that can serve as a bridge: its simplicity belies its utility. The paper is structured as follows. The next two sections deal with our use of UML for static and dynamic models. Subsequently, we discuss what we found to be inadequate for our needs and how we bridged the gap. We then offer concluding remarks.

STATIC MODEL

We want a static model to focus on certain abstract or concrete entities and their relationships; these entities are to capture the vital aspects of the problem domain and highest-level constructs of the software system. The static model in UML is called the class view and its diagrammatic form is called the class diagram. The UML static model is

very similar to the Entity Relationship (ER) model [6] when extended to include inheritance and generalization / specialization [10]. The entities are "classes" and binary relationships are "associations". However, UML can express a variety of special relationships such as aggregation, composition, generalization. The classes reflect concepts that are meaningful in an application, including real-world concepts and abstract ones. For example, a ticketing system for an airline company has concepts such as tickets, reservations, system interface, and data storage. Tickets and reservation are real-world concepts easily understood by end-users and should be included in our static model. Similarly, in a bank system, a customer object is associated with an account object. Such associations arise in the real world and grasping them presents no difficulty to the end-user.

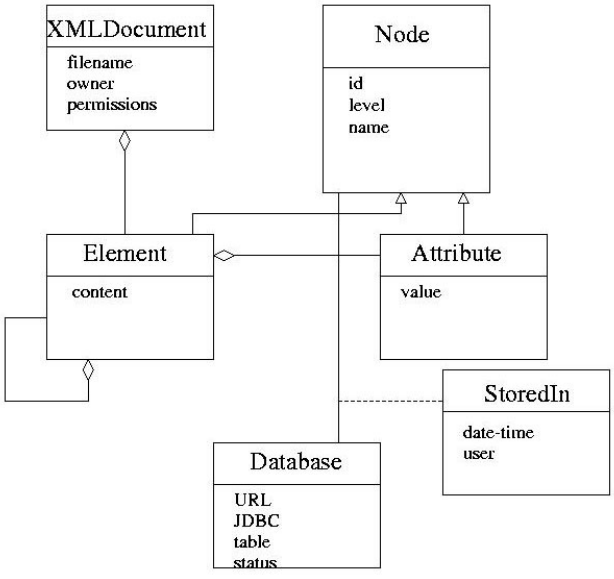
Figure 1 shows the static model of an application called XMLStore that converts XML data in a file into a relational database table. There are five classes. *XMLDocument* represents an XML file object with attributes *filename*, *owner* and *permissions*. Such an object consists of *Element* objects; the line joining the two ends in an open diamond indicating aggregation, i.e., a part-whole relationship. A similar aggregation between *Element* and itself signifies the possibility of arbitrary nesting of elements. Each element itself may contain zero or more objects called *Attribute*. A *Node* object represents the commonality between an element and an attribute (a triangle signifies a superclass), an example of generalization. Its attributes are *id* (its identifier), *level* (of the node in the XML tree), and *name* (of the element or attribute forming the node). These attributes are inherited by both subclasses. A *Database* object is specified by attributes *URL* (address of the database on a network), a *JDBC* (JDBC driver name specifying the kind of target DBMS), *table* (table name), and a *status* of that database. A sixth class *StoredIn* is special: it is an *association class* denoting the association between *Node* and *Database* reflecting the storage of nodes, i.e., attributes and elements in a database. The association class allows the association to have attributes; in this case, the date/time of storage and the user responsible.

Perspective

It has been suggested that UML class diagrams be associated with one of three perspectives [11]. In the *conceptual* perspective (*essential* perspective in [7]), classes capture concepts in the problem domain and associations represent relationships between such concepts. The *specification* perspective deals with interfaces of the software; classes are at the interface or type level and associations represent responsibilities. The *implementation* perspective is the one exhibited in most common UML applications: it deals with classes as they exist in an object oriented programming language and associations are pointers. Clearly, our use of the class diagram can be described as the first: the conceptual perspective.

Since UML is often used in program development, and many of the conceptual classes can also be implementation classes, how can one be sure that we have the right perspective? First, the classes of design concepts, such as classes dealing with the Graphical User Interface (GUI), should be absent. Second, UML classes also include operations on objects of the class, but most of these operations are decided in the design phase and are design related. For example, though *ticket* may be a valid class in the implementation phase with a method called *print*, this method may not reflect any facility available to the end user. Thus we prefer to eliminate operations from the class. Third, some attributes added to the class during the design phase should not be available to the end user. For example, a persistent object identifier (oid) attribute designed to identify an object in the class may be used only in internal operations hidden from the end-user.

Figure 1: *Static Model for XMLStore*



DYNAMIC MODEL

A dynamic model captures the dynamic behavior of the system. We need a dynamic model that focuses on the overall behavior of a system at a level appropriate for end-users. Our requirements are:

Readability: the level of abstraction should be appropriate for the end-user.

Monitorability: we should be able to match constructs with program fragments.

Ability to discern system behavior: it should provide insight into the dynamic behavior of the system.

For this purpose, we considered four schemes in UML: the Use Case View, the Interaction View, the State Machine View, and the Activity View. For shortage of space, we omit the details of this exercise [13] and simply observe that only the Activity View satisfied each of the three criteria, and therefore we selected it to serve as our Dynamic Model.

In general, the dynamic behavior of the system would be described by several activity diagrams, each of which we call a *flow*; a flow describes the realization of a certain functionality. Figure 2 shows the dynamic model for XMLStore (one flow may be enough for simple applications). The numbers next to the activity states in the figure are used only for identification. In state 2, a connection is established with a database. Then, the existence of the target tablename is checked; if it exists, it is emptied, else it is created, and then the XML file is opened. The four-way branch (the branch conditions are given within parentheses) is a response to each of four possible events caught by the SAX (Simple API for XML) parser employed: *start-element found, end-element found, character data found,* and *end of document reached*. Attributes found with a start-element are stored in a loop. In states 10 (store attribute) and 13 (end element), values read are written into a row of the table in the database.

Perspective

The activity diagram can not only describe the dynamic behavior of the system at a very high level, such as a workflow in an information system, but also at a lower level, such as a procedure implemented in a programming language. As with the static model, we have found it useful to associate a *perspective* capturing a level of detail with activity diagrams: workflow perspective, functional perspective, and algorithmic perspective. The workflow perspective describes how multiple high-level tasks cooperate to achieve a system goal. The functional perspective elaborates on one of those tasks by describing a sequence of functional units. The algorithmic perspective explains how one of those functional units will be implemented step by step, and it can be mapped directly to the implementation in code. Among these three perspectives, the workflow perspective diagram is closest to the requirement

analysis, and the algorithmic perspective diagram is closest to the implementation phase of software development. Generally, an end-user will be more interested in the details of the module he/she is using rather than a big picture of the tasks of the whole system. Also, the end-user may not understand many algorithmic perspective implementation details of the module. Thus, it would be appropriate to focus on the functional perspective activity diagram.

PROBING THE SYSTEM

In order to embed probes in the software system and allow end users to monitor the execution, two questions have to be answered:

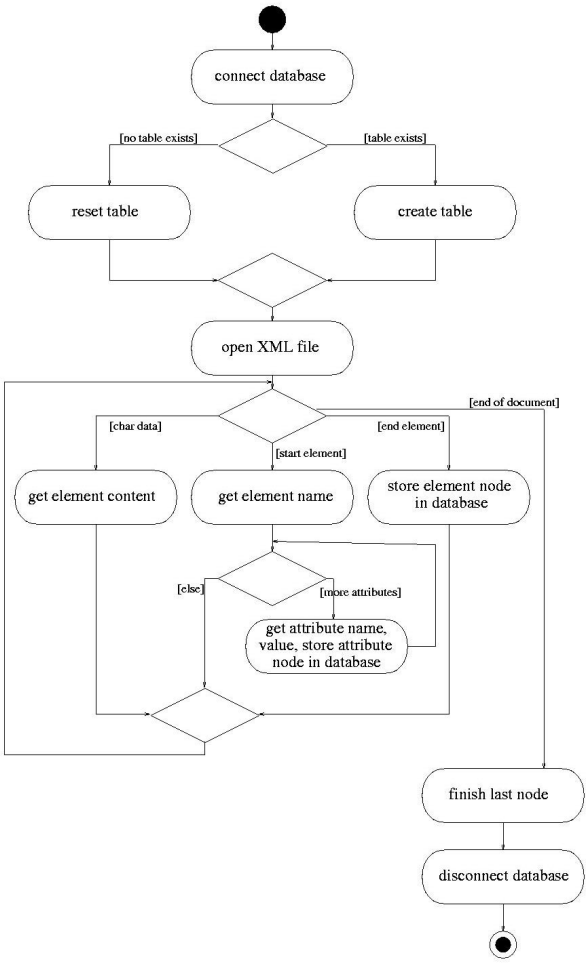
1. *what* should probes track, and
2. *where* (in the source code) to probe.

Our answers are

1. every entity (class), relationship (association), and attribute in the static conceptual model needs to be tracked because they are legitimate targets of the end user's queries, and
2. a probe must correspond to a step in the dynamic model.

In other words, the probes should monitor the sequence of activities described in the dynamic model and report how each such activity affects elements of the static model.

Figure 2: *Dynamic Model for XMLStore*



In order to express for each activity of the Dynamic Model exactly which elements of the Static Model are being used, the dynamic model needs to refer to the static model explicitly. But while the static model represents the conceptual objects in a system and a dynamic model describes the activities of the system and both describe the same system, it is not easy to discern the relationship between them, i.e., which objects of the static model are involved in an activity state or how attributes of objects affect the result of an activity. The activity view does not link actions with objects. The UML suggests swimlanes and object flow to overcome this disadvantage. Swimlanes are used to organize responsibility for actions according to class. They often correspond to organization in a business model. However, the swimlane does not describe what will happen to the object during the action. The object flow shows the flow of the objects, which are passed through the activities as the inputs and outputs. Generally, the object flow depicts the state-changing of an essential object during the workflow perspective activity diagram, but ignores the execution environment and the change in other objects. Similar inadequacies exist in the other dynamic views of UML. In summary, we found a need to link the static and dynamic models explicitly. We did so by adding a table called the *Activity Access Table* (AAT).

The Activity Access Table

In an activity state, some objects are accessed in a particular environment with a possible resultant change in their attributes. We define the *activity access attributes* to be all class attributes (from the static model) that are accessed in the given activity state (in the dynamic model). Activity access attributes provide the link between the static and dynamic models.

We associate with each activity access attribute one of two access types:

- *Dirty*: the value of the attribute is potentially changed during the activity.
- *Read-only*: the attribute is accessed but without any change in value.

We tabulate the above using an Activity Attribute Table (AAT). The AAT contains four columns: *activity state, accessed class, accessed attribute*, and *access type*. The first corresponds to an activity state in an activity diagram. The second and third together pinpoint the attribute of a class in the static model (the exact instance of the object can be handled by the underlying database and its query facility). Of course, an activity state can influence several attributes of several objects. The last column specifies one of the two access types: *D* stands for a dirty attribute, and *R* stands for read-only.

Table 1 shows a fragment of the AAT for the XMLStore application. In the activity state *connect database*, values of attributes *URL, JDBC* of class *Database* are read (i.e., these are read-only attributes)

Table 1: *AAT of XMLStore*

Activity State	*Class*	*Attribute*	*Type*
connect database	Database	URL	R
		JDBC	R
		status	D
reset table	Database	table	R
		status	D
create table	Database	table	R
		status	D
open XML file	XMLDocument	filename	R
…	…	…	…

while attribute *status* of the same class is updated (i.e., this is a dirty attribute).

Construction of the AAT can be semi-automated. When the designer specifies for each attribute of the static model a corresponding program object (which could be a variable or a component of a record, or a function), it is possible to analyze the code to find direct/indirect references to them. However, finding all such references can be complex in certain programming languages; hence, it cannot always be completely automated.

By itself, the table is not novel as a data structure. It falls in the genre of CRUD matrices; such a matrix basically associates one or more of C (Create), R (Read), U (Update), and D (Delete) with a data-process pair. In Object-Oriented modeling, such matrices have been shown to be useful at the specification and implementation perspectives by pairing, for example, a class with a Use Case [4].

On the other hand, in our approach, which is based on a conceptual perspective, such a structure is crucial and serves more interesting purposes: the table

- specifies the task of probing (which augments the original software system) by associating *what* needs to be probed with *where*;
- can be used to automate the process of insertion of the probes;
- itself can be queried to find a sequence of activity states (including instances of the same state such as in a loop) that affect one or more given attributes; this helps explain to the end user the reason for system behavior; and
- helps check integrity constraints defined on the static model by identifying at which activity states, the involved data attributes appear as dirty attributes.

CONCLUSION

We have explained our desire to facilitate on-line monitoring: such monitoring is geared to empower end-users with insight into software they execute. The key component is a conceptual model that remains an integral part of the final implementation providing abstraction and comprehension about the system state and its dynamic evolution. We have prototyped our approach on two application programs and two device drivers. In this paper, we have reported on our experience with UML in expressing our conceptual model requirements. We were compelled to add a structure to make the dynamic model refer to the static model explicitly. The structure is simple but it is both crucial and has interesting uses. This suggests that augmenting UML with an explicit link between the static and dynamic models could be very useful.

ACKNOWLEDGMENTS

We are indebted to anonymous reviewers for their helpful comments.

REFERENCES

1. C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design*. Benjamin/Cummings, 1992.

2. G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, New York, 1994.

3. J. Bowen and M. Hinchey. *High-Integrity System Specification and Design*. Springer-Verlag, 1998.

4. D. Brandon. CRUD Matrices for Detailed Object Oriented Design. *Journal of Computing Sciences in Colleges*, 18(2):306-312, December 2002.

5. M. Brodie, J. Mylopoulos, and J. Schmidt. *On Conceptual Modeling*. Springer-Verlag, 1984.

6. P. P. Chen. The Entity-Relationship Model — Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), March 1976.

7. S. Cook and J. Daniels. *Object-Oriented Modeling with Syntropy*. Prentice Hall, 1994.

8. T. Demarco. *Structured Analysis and System Specification*. Prentice-Hall, 1978.

9. E. Dijkstra. The End of Computing Science? *Communications of the ACM*, 44(3):92, March 2001.

10. G. Engels, M. Gogolla, and U. Hohenstein. Conceptual Modeling of Database Applications using an Extended ER Model. *Data and Knowledge Engineering*, 9(4):157-204, 1992.

11. M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1999.

12. C. Hoare. An Overview of Some Formal Methods for Program Design. *IEEE Computer*, 20(9):85-91, September 1987.

13. W. Hu. Inspecting Software Execution using Conceptual Model and Database. Master's thesis, New Mexico Institute of Mining and Technology, 2001.

14. I. Jacobson. *Object-Oriented Modeling and Design: A Use Case Driven Approach*. Addison-Wesley, New York, 1994.

15. C. Larman. *Applying UML and Patterns*. Prentice-Hall, 1998.

16. S. Mazumdar. Tackling the Software Crisis via Conceptual Modeling. In *Proc. 6th World Multiconf. on Systemics, Cybernetics and Informatics. SCI 2002*, volume 1, pages 205-210, 2002.

17. H. Mills, M. Dyer, and R. Linger. Cleanroom Software Engineering. *IEEE Software*, 4(5):19-25, September 1987.

18. OMG. Object Management Group. OMG Unified Modeling Language Specification, version 1.3., 1999. *http://www.omg.org*.

19. J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

20. J. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8-24, September 1990.