System-Level Object-Orientation in the Specification and Validation of Embedded Systems

João M. Fernandes
Dep. Informática
Universidade do Minho
4710-057 Braga, Portugal
miguel@di.uminho.pt

Ricardo J. Machado
Dep. Sistemas de Informação
Universidade do Minho
4800-058 Guimarães, Portugal
rmac@dsi.uminho.pt

Abstract

The main aim of this paper is to present how the Unified Modeling Language (UML) can be used as the notation to specify the requirements of an embedded system. By using a relatively small, but real, system (a supervision application) as a running example, the paper illustrates the design flow that can be followed during the analysis phase of complex control applications. To assure the continuous mapping of the models, the authors propose some guidelines to transform the use case diagrams into a single object diagram, which is the main diagram for the next development phases (design and implementation). The Java programming language is used for developing a system's prototype, to allow the system's validation by the customers.

1. Introduction

Embedded systems are predominantly control-dominated systems and usually designers specify them using state-oriented models, such as FSMs or Petri Nets [13]. However, for modeling more aspects of the systems (namely, data and function), it is critical to consider the use of multiple-view models. For this purpose UML was adopted, since it is a notation that covers the most relevant aspects of systems and is an industrial standard.

UML is a general purpose modeling language for specifying, visualizing, constructing and documenting the artifacts of computer-based systems, as well as for business modeling and other non-software systems [3].

UML presents many advantages for modeling embedded systems at the system-level. It is a standard, can be used to communicate with the customer, is suitable to object-oriented design, is totally platform independent, and possesses an extension mechanism to deal with non-standard modeling issues that is being used to define various UML application-domain profiles.

Among its disadvantages, UML can be criticized for having too many diagrams, to not have a precise semantics, and for introducing a new layer in the project.

2. The modeling process

The design flow in fig. 1 is proposed by the authors for modeling an embedded system. Although the process is presented in a sequential way (waterfall model), in practice, it must follow a more iterative and incremental flow. During a project, discussions amongst the team members must occur and information must be fedback to previous phases, to maintain the documentation updated.

The main views for specifying the system, suggested by the design flow of fig. 1, are captured by the following UML diagrams: (1) *Use case diagrams* are used to capture the functional aspects of the system as viewed by its users; (2) *Object diagrams* show the static configuration of the system, and the relations among the objects that constitute the system; (3) *Sequence diagrams* present scenarios of typical interactions among the objects that constitute the system or that interact with it, and allow system-level test-bench operationalisation; (4) *Class diagrams* store the information of ready-made components that can be used to build systems and specify the inheritance and hierarchical relationships among them; (5) *State-chart diagrams* are used to specify the dynamic behavior of some classes.

The information that is represented in state-charts, object diagram, and class diagram must be transformed into an unified representation. Two alternatives can be considered:

- *Oblog*. Oblog is a UML-based (extended subset) system-level object-oriented modeling language that allows the system to be simulated and whose CASE tool has automatic code generation capabilities [1]. The Oblog environment generates sequence diagrams,

Proceedings of the 14th Symposium on Integrated Circuits and Systems Design (SBCCI'01)
0-7695-1333-6/01 $10.00 © 2001 IEEE

IEEE COMPUTER SOCIETY

as a simulation output, that can be compared with those previously created to specify the system behavior in order to validate the system's requirements [14].

- *Java*. Java is an object-oriented programming language and is used to generate software prototypes of the system being specified. In this paper, we show only the use of Java for system prototyping in order to validate the user's requirements. Java can be used as an SLDL (system-level design language).

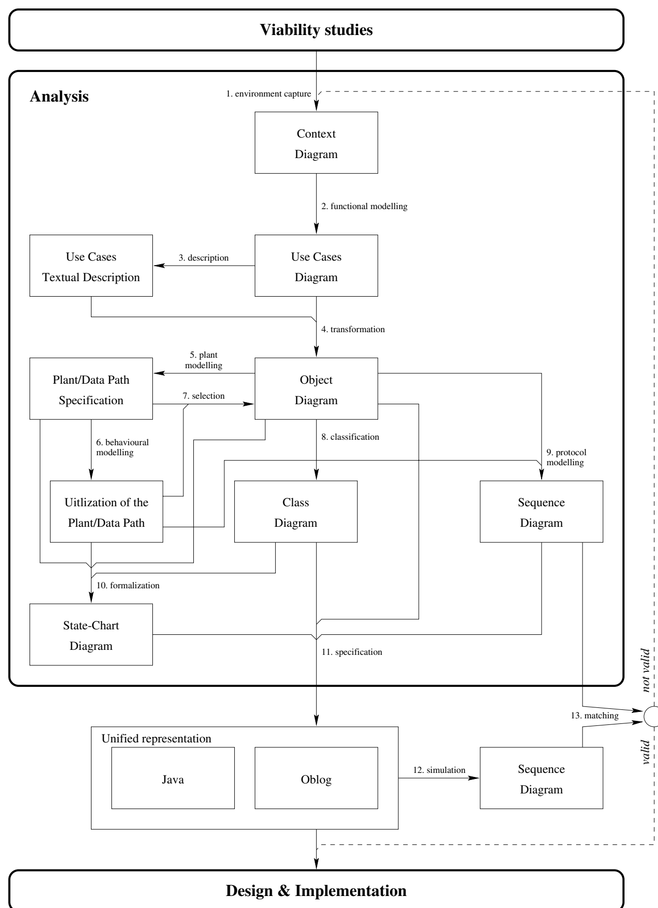

Figure 1. The design flow for developing embedded systems.

3. Context and use case diagrams

In this paper, a Lighting Supervision System (LSS) is used as a running example for showing the requirements engineering steps performed in the pre-synthesis operational phases. Although LSS is a relatively-small, but real, system that is used for explanation purposes, the approach has already been applied to develop complex embedded systems [7, 14, 6], where it is illustrated that a complete methodology can be based on the analysis approach presented here to support all the issues related to the design and implementation phases. These two phases are important to ensure that the user's requirements captured as this paper suggests can be really synthesized in a continuous model transformation approach.

The LSS is responsible for controlling the state of all the lamps that are used to decorate a building. The first diagram to be built is the context diagram of the system, that shows which actors interact with the system (not shown here).

The next task consists on the definition of the use cases of the system. A use case diagram is a powerful and useful technique for capturing the user's requirements. It is an easy-to-read diagram that divides the system in its functional points. A use case can be seen as a functionality or service that is offered by the system to its users.

Fig. 2 shows the system-level (or top-level) use case diagram, where it is possible to visualize which actors perform which functionalities. Since use cases have different impact on the final system, they must be ranked taken in consideration its importance to the main functionality of the system. This allows the project to follow a risk-driven process, where the most important or complex functionalities of the system are first tackled, leaving the less important ones to be treated later.

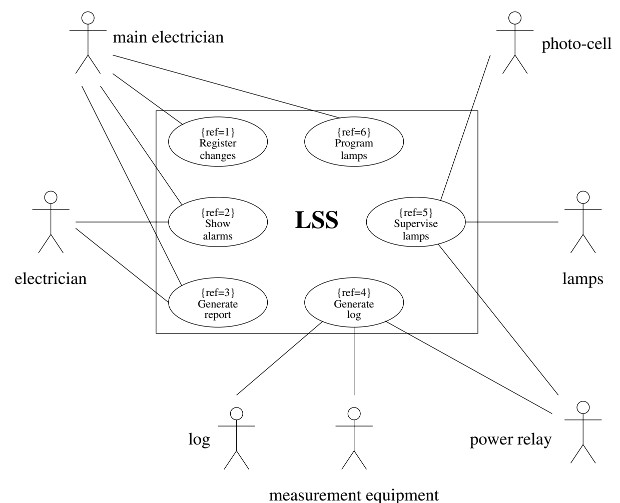

Figure 2. The use case diagram.

After identifying all the use cases of the system, the next step is to describe their behavior. As the next example shows for use case 5, the descriptions for the use cases were made with informal text, which is one of the possible alternatives [17].

"5. Supervise lamps. This function runs contin-

0-7695-1333-6/01 $10.00 © 2001 IEEE
IEEE COMPUTER SOCIETY

uously to turn on/off the lamps, according to the program in use. It is also responsible for detecting damaged lamps and registering this occurrences as an alarm for further processing."

4. Object diagrams

Object diagrams are also an important technique to show the components that constitute the system. Transforming the use cases that divide the system in a functional way into objects is a critical task, since there is no direct mapping from use cases to objects. Several use cases can give origin to one single object, a single use case can give rise to a couple of objects, and sometimes there are intersections among the use cases and objects.

The big question is how to identify the objects to be included in the object diagram? The first solution would be to consider each use case as an object [11]. This solution does not ease the localization of the modifications that must inevitably be executed during the system's life cycle (either during the development, or during its usage).

Another alternative, not considered in this work, is to use data objects to just store information and to totally put the dynamic behavior in control objects. It is better to avoid this solution, since it would be built a structure similar to those that result from applying any structured method, where the separation between data and functions is quite evident. In order to avoid the well-known problems associated with the structured approach, it is recommended, among other guidelines, to associate behavior to data objects.

The authors have developed a new method (known as the *4-step rule set*) [6] to obtain the objects from the use cases, assuring the models continuity in the mapping from the user's to the system's requirements. The object diagram presented in fig. 3 was obtained after applying the 4-step rule set to the use case diagram depicted in fig. 2. The object diagram illustrates the usage of active and passive objects [15]. An active object is continuously executing (has its own thread) and is autonomous, which means that it exhibits some behavior without the command of another system. In the example, objects *5.c supervise* and *clock* are active (notice that they are represented by a rectangle with wider borders). The clock object was introduced since the system deals with time information. On the other hand, passive objects do not initiate computation by their own and so they just perform some action after the command of other objects. Typically, the active objects are the basis for the control of the system under consideration, while the passive objects offer services to the active objects when asked for.

It is important to stress that the object diagram of fig. 3 is semantically compatible with the previously captured use cases, because the former has been rigorously "calculated" by using the 4-step rule set. This approach is a partial solution to the problem of obtaining the objects from the use cases, because there still exist some subjective interpretation in the execution of the steps. This is natural, since, although analysis and design can be systematically supported, it is important to leave some freedom for design space exploration. Typically, the object diagram is not directly related with the user's requirements.

Notice that use cases specify the functionality of the system (i.e. they divide it functionally), whilst the object diagram is related to the structure of the system, which is used as the foundation for the design and implementation phases. The object diagram represents an ideal architecture for the system, because its construction was completely independent of any implementation issue (platform, programming language, operating system, processor, etc.).

5. Class Diagrams

The majority of the OO methodologies do not pay too much attention to the object diagram. Usually, the class diagram is built firstly, but in this project the order was reversed. To develop the system under consideration and, more generally, embedded control systems, the authors believe that it is more important to have a good object model than a good class diagram, because the elements that do constitute the system are the objects and not their classes. This was the main reason to first identify the objects and to later classify them, that is, to select the classes to which those objects belong. We are not advocating to not work out the class diagram or even to ignore it. Obviously, the best situation is having good object diagrams and good class diagrams. What the authors promote is that the focus should be directed towards the construction of the object diagram.

It is possible that this object-driven perspective that puts classes in an apparently secondary role might be classified by some specialists as object-based rather than object-oriented. Notwithstanding, the approach that firstly defines the objects and later the classes is somehow consistent with the bottom-up discovery of inheritance, as defined in [16], to organize the classes in a hierarchical structure.

Additionally, and without sub-estimating the benefits and the utility that classes give to any object-oriented project, some self-designated OO methodologies start to relegate inheritance to a less important position [4].

Thus, it seems completely valid the approach that is proposed here: identify first the objects and then classify them. During the classification of objects the class structure is built, modified, or ideally just used. Reuse, a buzzword in all OO projects, can be achieved in three different ways, during the classes discovery. First, if there are more than one object of the same class, their definition is specified in just one place. Second, if classes with similar properties are found, hierarchical relations among those classes can be de-

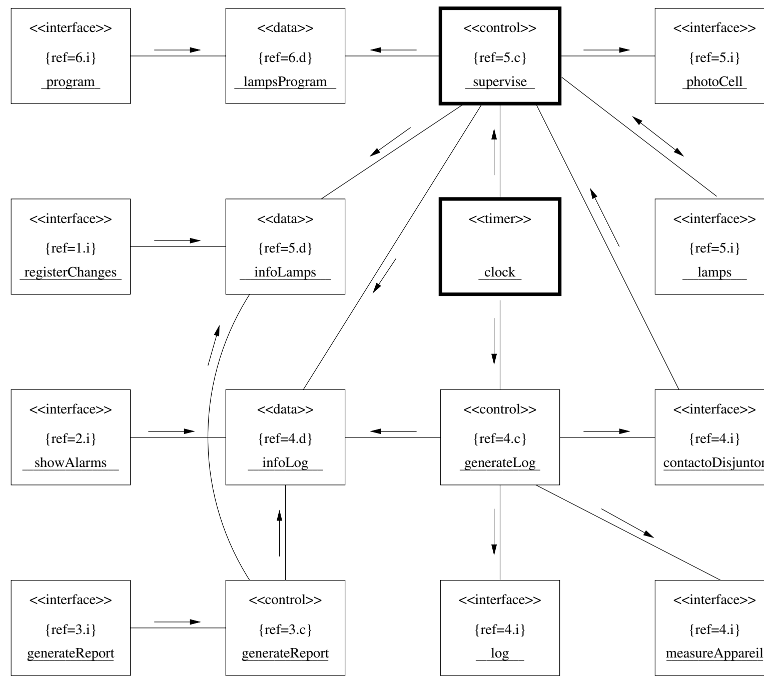

Figure 3. The Object diagram.

fined. Finally, when the class of an object is described, it is possible that the developer recognizes the existence of that class in a library, which allows it to be immediately reused.

Usually, the class diagram is understood as a template for a set of applications that can be obtained from it. In other words, the class diagram is a high-level generalization of the system. Whenever the developers define the way classes are interrelated, they are indicating all the systems (or, in a different perspective, all the configurations of a system) that can be obtained from those classes. So, it can be said that the class diagram is like a cookie-cutter for the object diagram.

With this perspective, it is common, in several methodologies, to not build the object diagram, since it automatically results from the class diagram. Whenever an object diagram is constructed, it is necessary to guarantee that the relations expressed in the class diagram between two classes also exist between instances of those classes. This is the main reason that methodologies usually impose (or suggest) class diagrams to be elaborated first than object diagrams.

There is an additional task in which it must be assured that there is consistency between the information that is described by both diagrams [5]. This fact can be interpreted as a symptom that some information is being unnecessarily replicated. For instance, the existence of the ≪singleton≫ stereotype in UML, which indicates that a given class can only have one single instance corroborates the perspective that sees the class diagram as a pattern for the systems, within a given application domain. This stereotype clearly indicates that if the object diagram is to be made consistent with the class diagram, it must satisfy the restriction of just presenting one single instance of that ≪singleton≫ class.

This kind of approach seems quite adequate and popular to develop, for example, business information systems or, more generally, any data-dominated system, where the objects are created and destroyed during the system life cycle. For example, in a system for bank accounts management, it is common that each account is always associated with, at least, one customer (this fact is indicated in the class diagram by associating the account class with the customer class). Thus, whenever an account object is created, it is mandatory to link it to, at least, one customer object. This approach is quite useful for business information systems, but does not offer many benefits for developing embedded systems (industrial control-based information systems), since normally the objects that constitute the system are not created and destroyed on the fly. An embedded system, as the one considered in this project, is generally, composed of a set of fixed objects that usually must be linked in an irregular way. Thus, it does not seem truly important to indicate, for example, that objects of the controller class need to be linked with objects of the sensor class. If in some applications this information can be quite pertinent, in others it may be completely inadequate or even wrong.

IEEE COMPUTER SOCIETY

Thus, the approach presented in this communication sees the class diagram as a repository of previously defined objects' specifications ("a raw material store"), that can be used to develop any application.

6. State Diagrams

For those objects that have a complex or interesting dynamic behavior, a state diagram should be specified. Since UML was chosen as the notation for all the documentation of the project, UML's state-charts were used to describe state machines [10].

The crucial component of the control application is object *5.c supervise* and for this object a state-chart was produced in order to specify its dynamic behavior (fig. 4).

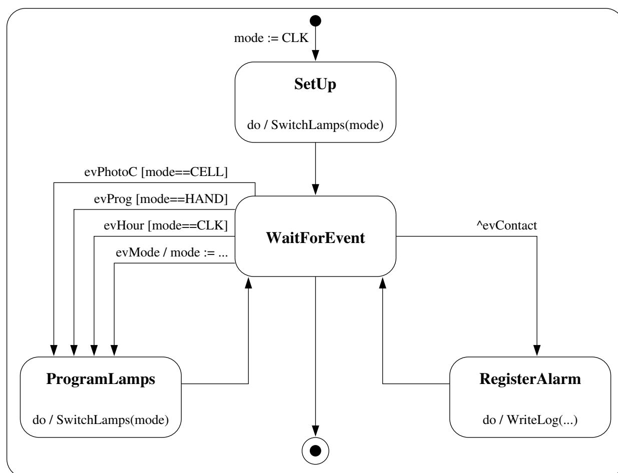

Figure 4. The state-chart for object 5.c.

In object-oriented modeling, the characteristics of a class are influenced by all its superclasses. Thus, when an object behaviour is defined by a statechart, inheritance issues among the classes must be considered. A solution to this problem is to completely ignore the superclasses' statecharts and draw a completely new statechart for the class. This solution is not considered, since it is contrary to the object-oriented modeling principles.

The statechart of a class must be inherited by its subclasses. For this purpose, some rules, similar to those that are used for code inheritance, must be fulfilled.

Due to the nature of the graphical modifications that can be done to a class' statechart, it is not possible to indicate those modifications in a simple and incremental way. Our proposal is to consider the statechart of a class to be graphically a complete diagram. There are some proposals to highlight the modifications made to a class statechart in relation to the superclass' statechart: one suggests dashed symbols for inherited elements and normal symbols for the new elements [18], while another uses gray symbols for inherited elements and black symbols for the new elements [19]. Whatever notation selected, the user has always to draw a completely new statechart from scratch, which means that if some modifications are made to a class' statechart diagram, they must also be made (by hand and not automatically) to all its subclasses.

7. Java Prototype

The authors developed a prototype in Java (fig. 5), based mainly on the object and state-chart diagrams created previously. Some rules were used for obtaining the code from the diagrams, which implies that there is a semantical continuity on the specification, allowing the designer to know which requirement gave rise to a given line of code. Fig. 6 shows the Java code for the Clock class.

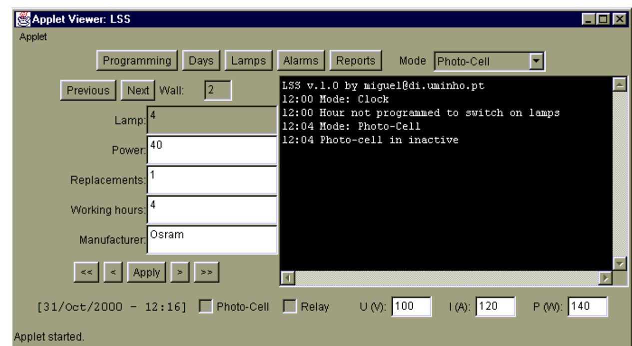

Figure 5. The system prototype developed in Java.

```
class Clock implements Runnable {
  private Thread t;
  public void run() {
    while (true) {
      if (new Date().getSeconds()==0) {
        showClock();
        if (new Date().getMinutes()==0)
          supervise.switchOn()
        try {
          t.sleep(2000); // waits 2 sec.
        } catch(InterruptedException ex) {;}
      }
    }
  }
}
```

Figure 6. The Java code for the class Clock.

The prototype execution allowed the users to experience the system behavior and to validate their requirements. This facility is extremely important, because it guarantees that the final system to be developed fulfills the users needs. If the user finds that something is not adequate, the team can still introduce the proper changes with minor costs. If the

IEEE COMPUTER SOCIETY

mismatches are found during the system's usage, the price to correct them is considerably high.

If the Java code is to be used for the final implementation, some modifications must be introduced to the prototype code, since Java is not yet a suitable language for developing embedded real-time systems, mainly due to the code size, the non-deterministic nature of the behavior and the low performance [9]. In 1999, the RTJEG (Real-Time for Java Experts Group) started the RTSJ (Real-Time Specification for Java) [2], which defines a platform that allows the designers to predict the temporal behavior of the software during run-time. Taking into account the Java language specification [8] and the JAVA Virtual Machine (JVM) specification [12], the RTSJ specification has revised the topics related to scheduling, memory management, and synchronization and has added four items related to asynchronous events reaction, asynchronous transfer control, asynchronous thread termination and physical access to memory. These topics were introduced with some judicious care in order to maintain the WORA (write once, run anywhere) principle, that gave Java a great popularity.

8. Conclusions

This paper has presented how UML can be utilized in real projects to analyze, model and specify industrial control systems. Since UML is intuitive for non-technical people, it covers the main views of a system, it is independent of the platform and it is an OMG standard, the authors are using UML as an adequate modeling language in industrial projects.

The transformation from use cases into objects is a critical task within the development process. For this reason, this paper proposes an rigorous holistic approach during this transformation, to obtain, in a semi-automatic migration step, the object diagram that best maps the user's requirements into the system's requirements.

The approach presented in the paper puts more emphasis on objects rather than on classes (true object-driven), which is one of its main divergences in relation to the traditional object-oriented approaches (true class-driven).

Since some objects of the system present dynamic behavior, state-charts were used because they are adequate for modeling that view of the systems. Finally, taken into consideration the various UML diagrams, a prototype in Java was built in order to validate the user's requirements. Building a prototype is an important issue, since it allows users to experiment the system's services before the final system is constructed.

References

[1] L. F. Andrade, J. C. Gouveia, and P. J. Xardoné. Architectural Concerns in Automated Code Generation. In *OOPSLA Midyear Conference*, 1998.

[2] G. Bollella and J. Gosling. The real-time specification of java. *IEEE Computer*, 33(6):47–54, June 2000.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] D. Budgen. *Software Design*. Addison-Wesley, 1994.

[5] B. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.

[6] J. M. Fernandes and R. J. Machado. From Use Cases to Objects: An Industrial Information Systems Case Study Analysis. In *7th Int. Conf. on Object-Oriented Information Systems (OOIS '01)*, Calgary, Canada, Aug. 2001. Springer-Verlag.

[7] J. M. Fernandes, R. J. Machado, and H. D. Santos. Modeling Industrial Embedded Systems with UML. In *8th ACM/IEEE/IFIP Int. Workshop on Hardware/Software Codesign (CODES'2000)*, pages 18–22, San Diego, CA, USA, May 2000. ACM Press.

[8] J. Gosling and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] D. Hardin. The Real-Time Specification for Java. *Dr. Dobb´s Journal*, pages 78–84, Feb. 2000.

[10] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–74, 1987.

[11] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[13] R. J. Machado, J. M. Fernandes, A. J. Esteves, and H. D. Santos. *Hardware Design and Petri Nets*, A. Yakovlev, L. Gomes and L. Lavagno (eds.), chapter "An Evolutionary Approach to the Use of Petri Net Based Models: From Parallel Controllers to HW/SW Co-Design", pages 205–22. Kluwer Academic Publishers, Feb. 2000.

[14] R. J. Machado, J. M. Fernandes, and H. D. Santos. *Architecture and Design of Distributed Embedded Systems*, B. Kleinjohann (ed.), chapter "A Methodology for Complex Embedded Systems Design: Petri Nets within a UML Approach", pages 1–10. Kluwer Academic Publishers, Apr. 2001.

[15] P. J. Robinson. *Hierarchical Object-Oriented Design*. Prentice-Hall, 1992.

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

[17] G. Schneider and J. Winters. *Applying Use Cases: A Pratical Guide*. Addison-Wesley, 1998.

[18] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[19] W. Weber and P. Metz. Reuse of Models and Diagrams of the UML and Implementation Concepts Regarding Dynamic Modeling. In *The Unified Modeling Language: Technical Aspects and Applications*, M. Schader and A. Korthaus (eds.), pages 190–203, Heidelberg, Germany, 1998. Physica-Verlag.

0-7695-1333-6/01 $10.00 © 2001 **IEEE**

IEEE COMPUTER SOCIETY