Active Object Modeling

U. Brockmeyer, W. Damm, C. E§mann, J. Klose, I. Schinz
OFFIS/University of Oldenburg, Germany

Overview

During the last years OFFIS/UoO have developed concepts and tools for automatic test case generation and formal verification of Statemate designs. Statemate is a set of tools from I-Logix, Inc., supporting the paradigm of SA/SD. Recently we have started to transfer and extend these concepts to the object-oriented UML context. For formal treatment of UML models we have to have a precise semantics of the considered diagrams. We intend to demonstrate the concepts of automatic test generation and formal verification by exploring UML models designed with the Rhapsody tool of I-Logix. In the following we sketch the execution model which our work is based on.

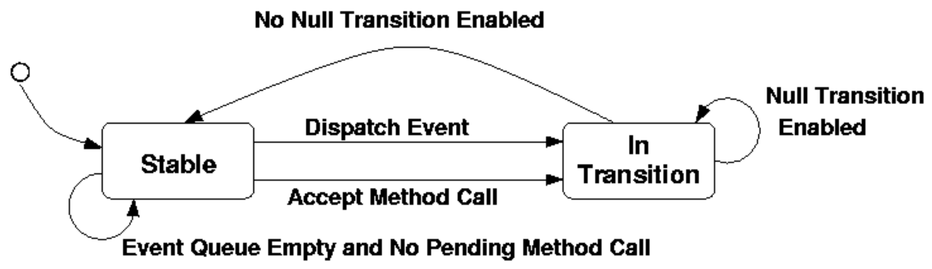
Introduction

We consider an object system created from a finite number of classes C_i. Each class has associated a set of typed attributes. For the sake of this discussion, types can be either class identifiers, in which case attributes point to instances of the class defined as its type, or some predefined types, like floats, integers, boolean, etc. Attributes are private and other objects can manipulate these attributes only by executing method calls.
With each class we associate a set of operations, which the object is willing to serve. Operations may be parameterized, and are seen as always returning a value. Operation calls are executed synchronously, that is, the caller is blocked until reception of the return value. We require operations to be guarded, therefore all objects are guarded.
Objects can also asynchronously communicate through exchange of events. Events are directed and are equipped with parameters. An object will process incoming events only one at a time, entailing a queuing mechanism for incoming but not yet processed events.
Each *active object* thus has its own event queue, whereas *passive objects* share their event queue with some active object. We declare classes as *reactive*, if their dynamic behavior is specified by Statecharts. As an additional attribute, priorities are associated with active objects, determining the order in which incoming method calls are served.

Execution model

To define a model of execution, several design issues have to be considered:
- single flow of control in each object instance has to be guaranteed
- level of interference
- conflict between accepting method calls and dispatching events

Since we require all operations to be guarded our proposed execution model ensures that there is **at most a single thread of control** active in each object at each point in time. We feel that violating this assumption leads to execution models, which are complex and incomprehensible. We take a standard interleaving semantics, in which all active objects are running asynchronously. In interleaving object executions, we have to decide on the **level of granularity of interference** between different threads of control. Our execution model interleaves execution at a coarse level, and does not allow preemption of run-to-completion steps. This — in combination with the restriction to guarded operations — increases understandability and analyzability of the model. In contrast to a finer grained interference model, the coarse grained interleaving should allow significant optimizations for the analysis tools, since no external communication — neither through operations, nor events — occurs during a completion run.



Our execution model is shown in the state diagram in the figure above. This state diagram does not state anything about situations where both method calls are pending and events can be dispatched. The **decision to dispatch an event or to accept a method call is based on priorities**. The priority of a method call is equal to the priority of the object calling the method. If the priority of the caller is higher than the priority of the object providing the method, then the method call is accepted. Otherwise an event is taken out of the event queue and will be processed. In case of ties the situation is solved by randomly choosing the event or the method call.

The proposed execution model should be seen as a base model, but it is still necessary to further investigate some of the more subtle semantical issues, for instance:

- The priority resolution is done irrespective of an objects current Òready setÓ which is the set of method calls and events occurring as transition guards. Considering a situation where the priority of a pending method call and the object priority are identical, the ready set could be used as a tie-breaker.
- A general problem with UML events is that they may be dispatched when they are not in the current ready set, and thus are lost. A concept is conceivable which supports re-enqueing of events and thus give it a second chance.
- Method calls of objects with priority lower than the current object *O* will only be served if *O«s* event-queue is empty. Thus, it is possible that low priority method calls will never be accepted by *O*. This will make it very hard to predict response times.
- Another possible extension of the execution model is to add priorities also for events and to modify the dispatcher.

Applying the execution model

The typical unit of automatic test case generation and formal verification are complete tasks. On the UML model level, we consider tasks to be compound objects, each containing one active object and a collection of passive objects, which all share the same thread of control and the same event queue. Moreover, since operations are assumed to be guarded, there would also be a monitor associated with each composite object, maintaining all pending method requests.

As a formalism for specifying verification requirements and as a basis for automatic test case generation we will use an extension to UML sequence diagrams (live sequence charts), allowing to express progress properties. It enables us to:

- formally state pre-and post-conditions of events and methods,
- formally state invariant properties for classes
- formally state acquaintanceship relations
- perform scenario-based test case generation and verification