Aspect-Oriented Development Using Protocol Modeling

Ashley McNeile[1] and Ella Roubtsova[2]

[1] Metamaxim Ltd, 48 Brunswick Gardens, London W8 4AN, UK
ashley.mcneile@metamaxim.com
[2] Open University of the Netherlands,
Postbus 2960, 6401DL Heerlen, The Netherlands
ella.roubtsova@ou.nl

Abstract. The aim of this paper is to explore the modeling of crosscutting behavioral abstractions. We argue that behavioral aspects can be seen as a particular kind of more general behavioral abstraction called a "mixin". Mixins support a compositional style of modeling, whereby a complete class definition is constructed by composing one or more mixins each of which represents a partial definition of the class. If used as the replacement for inheritance, mixins can provide an expressive power equivalent to multiple inheritance.
In this paper we use a modeling semantics called *Protocol Modeling* to illustrate how mixins can be used to represent behavioral aspects. We use the Crisis Management System case to illustrate the Protocol Modeling approach, and describe how the model can be executed to give early validation of its behavior.
We discuss the extent to which the Protocol Modeling approach is scalable to large problems, is suitable for evolutionary development and supports correctness analysis and testing.

1 Introduction

Behavioral modeling languages can be seen as the next generation of abstract programming languages. This suggests that they are executable and capable of representing all the abstractions that have to be implemented in a final system. Objects, Services, Aspects, Components and Agents have to find their place and be distinguished in models. But a unified framework that supports all of these different types of abstraction using a common semantic base has yet to be found, perhaps because the behavioral semantic constructs currently used in modeling are not appropriate to this challenge.

The Protocol Modeling approach [7] experiments with new semantic constructs for describing system behavior in terms of event protocols. The approach supports a highly compositional style of modeling, whereby a complete class definition is constructed by composing one or more partial definitions called *mixins*. If used as the replacement for inheritance, mixins can provide an expressive power equivalent to multiple inheritance [15]. But while mixins have been supported as constructs in programming languages[3] they have not generally been seen as a construct in modeling languages, which have tended to employ hierarchical inheritance structures. However, as we demonstrate, mixin based modeling is a viable alternative with the potential for wider expressive power.

In Protocol Modeling, behavioral mixins are used as the building blocks of a model. They are used both to construct inheritance like relationships (A is a "kind of" B) and also to separate out common behavior that is shared by a number of different entity classes. The modeling constructs of Protocol Modeling allow one mixin to abstract over the events, states and data of others; so that a single element of one mixin matches, or joins to, a number of different elements in other mixins. This gives the compositional calculus a flavor of "weaving" that is a feature of aspectual styles of software design. We suggest, therefore, that mixins composed in this way can be seen as a technique of aspectual modeling.

In this paper we illustrate and justify this claim using a protocol model of a *Crisis Management System* [20], a case study developed by Jörg Kienzle, Nicolas Guelfi and Sadaf Mustafiz for the comparative study of aspect-oriented modeling techniques.

This paper is organized as follows:
Section 2 provides an overview of the semantics of Protocol Modeling.
Section 3 describes how Protocol Modeling is used to describe the classes of a system, their behavior and associated business process.
Section 4 describes how a Protocol Modeling is built in stages using the information provided in a typical requirements specification.
Section 5 describes our protocol model of the Crisis Managements System case and shows how Protocol Modeling embodies an aspectual style of modeling.
Section 6 describes briefly how, in general, a complete protocol model is converted into a final, deployable, system.
Section 7 classifies Protocol Modeling among other aspect-oriented approaches and emphasizes the features of Protocol Modeling that make it aspect-oriented.
Section 8 discusses some practical considerations in Protocol Modeling, including some barriers to its use and the support it gives for model execution and testability, evolutionary development, scalability and determination of correctness.
Section 9 contains related work and conclusions.

2 Protocol Modeling

In this section we provide an overview of the semantics of Protocol Modeling, sufficient to enable understanding of our solution to the Crisis Management System (CMS) case. A complete account of Protocol Modeling can be found in McNeile and Simons [7].

[3] For example: Lisp, Simula, Python and Smalltalk.

2.1 Events

In Protocol Modeling, the basis of behavior specification is identification of occurrences of interest in the environment (or domain). These occurrences are taken to be atomic and instantaneous. An "event" (more properly "event instance") is the data representation of an occurrence in the environment as a set of data attributes. Every event is an instance of an event type. The type of an event determines its metadata (or attribute schema), this being the set of data attributes that completely define an instance of the event type. This approach to modeling occurrences in the domain as events is identical to that used in other event based modeling approaches, such as those described by Jackson and Cook et al. [21, 30].

2.2 Protocol Machine

A *Protocol Machine* (hereafter referred to as just "machine") is a reusable behavioral component of the model. A machine has a defined alphabet of event types that it understands. When a machine is presented with an event it will either *ignore* it, *accept* it or *refuse* it as follows:

- If the event *is not* in its alphabet, the machine ignores it.
- If the event that *is* in its alphabet, it will either accept it or refuse it.
- Acceptance or refusal of an event by the machine is determined by *protocol rules* that are owned by the machine and that the machine evaluates both *before* and *after* the event.

The protocol rules of a machine are normally depicted using a state transition diagram, where the transitions exiting a state show the event types that the machine accepts in that state. It is important to understand that "refusal" means that the machine is unable to handle the event at all in its current state, and this normally means that some kind of error message is generated back to the environment. How or where such an error is generated is not of concern for modeling purposes.

Between handling events, a machine has a well defined quiescent state, meaning that it can undergo no further change of state unless and until presented with a new event.

2.3 Local Storage

A machine has *local storage* which only it can alter, and only when moving to a new state in response to an event. The local storage of a machine is represented as a set of attributes associated with the machine.

A machine may read, *but not alter*, the local storage of other machines with which it is composed. A machine may use its accessible storage (its own local storage and the local storage of other machines composed with it) to compute updated values of its own attributes when processing an event, and to compute the value of derived attributes. In addition, a machine may use its accessible storage to compute its own state, as we describe next.

2.4 Derived States

A protocol machine may have a state is *derived* on-the-fly rather than being stored as part of the machine's local storage. This is exactly analogous to the concept of a *derived attribute*, as defined in modeling approaches such as UML [24]. The states of a derived state machine determine what events it accepts and what events it refuses and in this sense the fact that states are derived makes no difference to their use as determinants of the machine's behavior.

For clarity of semantics, we do not mix derived and stored states in the definition of the same machine (although it is possible to compose derived and stored state machines using CSP as described later in Section 2.6). Note that machines with derived state are not "topologically connected" as the new state that results from a transition firing is not determined as the end point of a transition. Further discussion can be found in the literature on Protocol Modeling [7].

2.5 Modal Semantics

We shall see later when discussing Use Cases in Section 4 that derived states enable the definition of machines that have *modal semantics*, describing what is *desired* rather than simply what is *possible.* These machines are used to model the fact that the system may decide that a particular event is required and may solicit or prompt the event from the actor responsible for performing it. This gives protocols the expressive power to express *workflow.* The idea of using modal machines to describe workflow rules has been described by McNeile and Simons [8].

2.6 Composition

A protocol model of a system is a non-empty finite set of protocol machines. Each machine presents a partial behavior and these partial behaviors work together to create the behavior of the system.

The machines of a model are composed using a parallel composition operator (P‖Q)[4] essentially the same as that defined by Hoare in his process algebra, Communicating Sequential Processes [10]:

- The alphabet of the composed machine is the union of the alphabets of the constituent machines; and the local storage of the composed machine is the union of the local storages of the constituent machines.
- When presented with an event the composed machine behaves as follows:
  1. If both machines *ignore* the event, the composed machine *ignores* it;
  2. If either machine *refuses* the event, the composed machine *refuses* it;
  3. Otherwise the composed machine *accepts* the event.

[4] Protocol Modeling approach also supports other composition forms [5, 14] for composition of distributed abstractions. However, these composition forms are beyond the scope of this paper and we do not discuss them.

It is clear from this definition that the concept of *event refusal*[5] is critical to composition.

2.7 Concurrency and Determinism

Protocol machines are deterministic in the sense that executions of a protocol model are repeatable. If a given sequence of events is presented to a model twice starting from the same initial state, the final state of the model will be the same, as will be the set of events that it could accept next. The reason for requiring determinism is the ability this gives to argue about behavior based on traces[6].

There is, however, no assumption that non-determinism may not be introduced at physical design time if it is decided to distribute the model across multiple (real or virtual) processors; but this must be done in such a way that the behavior of the model is not broken by such a distribution.

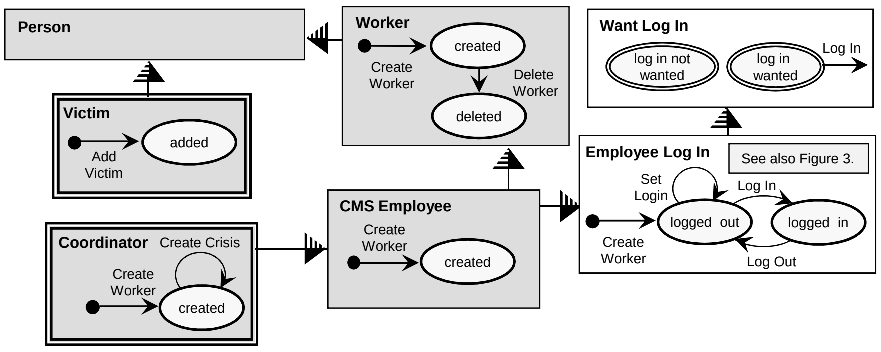


Fig. 1. Graphical Representation (Fragment of CMS)

3 Representation of a Protocol Model

In this section we illustrate how models are described in ModelScope, the tool that we have used for developing and testing our model of the CMS case study. Using examples from the CMS case study, we illustrate how the various features of Protocol Modeling are used to address the requirements of the CMS system. Later, in Section 8.2, we discuss how ModelScope is used to explore and validate models.

[5] Note that event refusal is **not** supported by the conventional semantics of UML statecharts [12].

[6] Note that determinism is **not** guaranteed by the conventional semantics of UML statecharts [12].

3.1 Events and Machines

The event `Create Worker` is described as follows:

```
EVENT Create Worker
# Brings a new worker into existence
   ATTRIBUTES Worker: Worker, Person Name: String,
      Address: String, Date of Birth: Date,
      Expertise: Mission Type
```

This lists the `ATTRIBUTES` for this event type and describes each using *attribute name*: *attribute type*.

The protocol machine `Worker` is described as follows:

```
BEHAVIOUR Worker
# A worker, either Internal or External
   INCLUDES Person
   STATES created, deleted
   TRANSITIONS @new*Create Worker=created,
      created*Delete Worker=deleted
```

This description gives the possible states, {`created`, `deleted`}, for a `Worker` and describes the behavior rules as a set of `TRANSITIONS`. Each transition is represented as *starting state***event type*=*resulting state*. The state `@new` represents the initial state of the machine, the "black dot". It can be seen that this textual representation describes the state transition diagram shown in the central box at the top of Figure 1. (This diagram is a part of the overall protocol model of CMS that we describe later in the paper.) Note that the graphical representation is partial, in that it does not show the attributes.

The `INCLUDES` entry specifies that whenever the `Worker` machine is instantiated, an instance of the `Person` machine must be created too and composed with it. The `Person` machine is:

```
BEHAVIOUR Person
# Attributes shared by all types of Person
   ATTRIBUTES Person Name: String, Address: String, Date of Birth: Date
```

In this case, the included machine defines local storage attributes but does not define any behavior (transitions). The result of the composition is that a `Worker` will include the attributes defined in `Person`. The `INCLUDES` is the Protocol Modeling analogue of inheritance, but is a *composition* operator and to emphasize this difference we use a semi-hatched arrow (as opposed to the open arrow used for inheritance in UML Class Diagrams). The `Person` machine can be seen as more abstract than `Worker` as it is used by other machines: for instance it is also used by `Victim`:

```
OBJECT Victim
# A Victim of a Crisis
   NAME Person Name
   INCLUDES Person
```

```
ATTRIBUTES Crisis: Crisis, Medical Condition: String
STATES added
TRANSITIONS @new*Add Victim=added
```

As can been seen here, a machine can be defined with both `ATTRIBUTES` and `TRANSITIONS`. When a `Victim` is instantiated, its local storage is the union of the set of attributes in the `Victim` and `Person` machines, following the semantics of composition in Section 2.6. The Victim machine is concrete (instantiable) and this is indicated by the use of the keyword `OBJECT` instead of `BEHAVIOUR`, and the inclusion of the `NAME` entry which specifies which attribute to use to identify instances of `Victim` in the ModelScope user interface. In the graphical view, as in Figure 1, we use the double outline for the enclosing box to indicate that a machine is concrete.

3.2 Structure of a Model

The `INCLUDES` structure of the mixins forms a partially ordered set[7] (or "poset"). This is normal structure for model elements that represents multiple inheritance, as a given element may include more than one child and may be included in more than one parent.

Note that this structuring is not apparent from the diagrams (Figures 1, 5 and 6) which have been drawn to fit in the limited space of a page. Given more space, these diagrams could be refactored to show the semi-hierarchical form of a partially ordered set, with all of the `INCLUDE` arrows pointing upwards.

As an example, consider `Coordinator`. This is instantiable (it uses the keyword `OBJECT`) and creation of an instance entails instantiation of one of each of {`Coordinator`, `CMS Employee`, `Worker`, `Person`, `Employee Log In`, `Want Log In`}: this being the transitive closure of its `INCLUDES` relationships shown in Figure 1. With the exception of `Person`, all of these machines have behavior and so, in accordance with Section 2.6, the total behavior for `Coordinator` is the CSP composition of the behaviors of these machines.

3.3 Attribute Handling

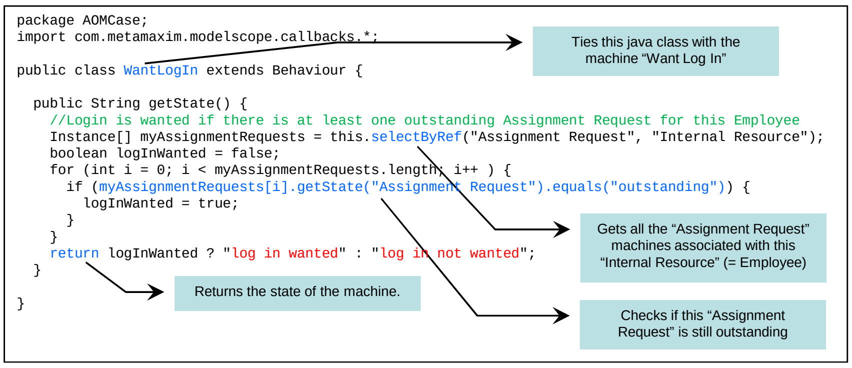
The convention in Protocol Modeling is that when an object (an assembly of machines) accepts an event the values of the event attributes transfer to matching attributes of the object; so when a `Create Worker` event is accepted by the model the values of {`Person Name`, `Address`, `Date of Birth`} will be given to the new `Worker` instance created. It is possible to define more complex updates with explicit update logic where required. We shall show examples of this later in the paper.

[7] A poset consists of a set together with a transitive binary relation that indicates that, for certain pairs of elements in the set, one of the elements precedes the other.

3.4 Derived States

As described in Section 2.4, some of the machines in a model can have derived rather than stored states. The `Want Log In` machine at the top right of Figure 1 is an example, as is the machine `Employee Log In` in Figure 3. The `!` prefixing the name of the machine tells ModelScope that it has to execute a function to determine the state of the machine. The code that calculates the state for the `Want Log In` machine is shown in Figure 2 and that for `Employee Log In` in Figure 3 (below the diagram).

In the graphical form we use the double outline to the state icons to indicate that the state is derived. As mentioned in Section 2.4, machines with derived states are not "topologically connected" as the state value is not driven by transitions. The state value `@any` means "any state".



```
package AOMCase;
import com.metamaxim.modelscope.callbacks.*;

public class WantLogIn extends Behaviour {

  public String getState() {
    //Login is wanted if there is at least one outstanding Assignment Request for this Employee
    Instance[] myAssignmentRequests = this.selectByRef("Assignment Request", "Internal Resource");
    boolean logInWanted = false;
    for (int i = 0; i < myAssignmentRequests.length; i++ ) {
      if (myAssignmentRequests[i].getState("Assignment Request").equals("outstanding")) {
        logInWanted = true;
      }
    }
    return logInWanted ? "log in wanted" : "log in not wanted";
  }

}
```

Ties this java class with the machine "Want Log In"
Gets all the "Assignment Request" machines associated with this "Internal Resource" (= Employee)
Checks if this "Assignment Request" is still outstanding
Returns the state of the machine.

Fig. 2. State Derivation of the "Want Log In" Machine

3.5 Modal Machines

As described in Section 2.5, some machines may have modal semantics for describing workflow. An example is the machine `Want Log In` which determines that an employee has outstanding requests for mission assignments and should therefore log into the system to attend to them. The ModelScope definition the machine is below. The entry `TYPE DESIRED` defines this as a modal machine.

```
BEHAVIOUR !Want Log In
# Makes a login desired if there is an
# outstanding Assignment Request for the Employee
   TYPE DESIRED
   STATES log in wanted, log in not wanted
   TRANSITIONS log in wanted*Log In=@any
```

As already noted, this machine has a derived state as indicated by the `!` prefixing its name.

3.6 Actors

The machines of a model represent the objects of the domain and capture how events change the states of these objects. We also have to consider who is responsible for an event and this is modeled using the concept of *Actors*[8]. Different events in the same machine may be the responsibility of different actors. As an example, consider the events in the `Employee Log In` machine in Figure 1. Responsibility for the `Create Employee` event, which sets up a new `Employee` in the system, falls to the *System Admin Department*[9]. However, the `Log In` event, which is also in the alphabet of this machine, is done by the employee him\herself in order to access the system.

Actors are modeled completely separately from the machines and their behavior, and serve to configure the user interface so that each Actor gets an interface containing only the behaviors and events from the model that are appropriate to his\her responsibility. An example of an `ACTOR` definition is given below.

```
ACTOR System Admin
   BEHAVIOURS CMS Employee, Crisis Type, Mission Type, Needed Mission
   EVENTS Create Worker, Set PDA Number, Set Login, Reset,
      Create Crisis Type, Create Mission Type, Make Needed
```

Actors form a separate layer that configures the user interface for the different users of the system. The Actor layer does not change the behavior of the underlying protocol machine model in any way whatsoever.

4 Building a Protocol Model

The starting point for building a protocol model can vary, depending on the nature and stage of the project. Sometimes a general understanding of the area and the purpose of modeling is all that is available; and sometimes a detailed requirements document has been prepared. We will describe the process that is followed when starting from a written document such as that provided for the Crisis Management System [20] (hereafter referred to as "the CMS Spec."), as this is quite typical.

Before describing the process, one aspect of Protocol Modeling needs to be emphasized as it differs significantly from traditional approaches: the intention and best practice in building a protocol model is that it is *executable and testable throughout the process of building an evolving model.* This is useful and desirable even if the final code of the system is being hand-crafted rather than generated. We discuss the reasons for doing this later, in Section 8.2, and it is also well explored in the authors' paper [4].

The process of building a protocol model is described below as a number of steps, but this is somewhat artificial. While the ordering of the steps does give the flavor of the normal strategy for building a model, in practice the process is very iterative and decisions can be revisited and changed at any time.

[8] Essentially the same as the concept of Actor in UML Use Cases.

[9] See Section 2.4.2 of the CMS Spec.

4.1 Step 1: Model the Domain

The starting point for building a protocol model is typically a Domain Model, such as that shown in Figures 4 and 5 of the CMS Spec. Using this, we proceed as follows to get to a first version of the model:

- Create an instantiable machine (using keyword `OBJECT`) for every concrete object in the model.
- Specify the `ATTRIBUTES` for each machine. These are normally given as part of the Domain Model.
- Define the lifecycle of each object, in terms of its `STATES` and the `EVENTS` that bring an instance into existence and change its state. The lifecycle will not be part of a standard Domain Model, which is normally just a static model. To find out the lifecycle, other artifacts such as Use Cases or Business Processes (Activity Diagrams) are used, or discussions held with domain experts.

This will result in a first cut protocol model, which can be executed. At this stage the executable model allows domain objects to be instantiated and taken through their lifecycles.

4.2 Step 2: Model Associations

As well as describing the dynamics of the creation and state changes of objects, a protocol model also describes the dynamics of associations between objects: how associations between objects are created and dissolved. Associations are created by events that appear in the lifecycle of more than one object. For example, the Domain Model in Figure 4 of the CMS Spec. shows associations between `Crisis` and `Crisis Type`, and between `Crisis` and `Coordinator`. These associations are created by the event `Create Crisis` that appears in the lifecycles of `Crisis`, `Crisis Type` and `Coordinator`.

The second step of building a model is to add events to the basic model formed in Step 1 to create and dissolve the associations in the model. Often, many of these will be already be there as a result of Step 1 but this is not always the case. In addition, the attributes of the objects in the model are checked to ensure that they contain appropriate foreign keys for the associations: thus the `Crisis` object should have attributes to hold the identifier of the associated `Crisis Type` and `Coordinator`. The result is a model in which the associations between objects can be created and dissolved.

4.3 Step 3. Model Inheritance

The third step is to model the inheritance. This involves refactoring the model by using abstract, non-instantiable, machines (using keyword `BEHAVIOUR`) to factor out attributes and/or behaviour that is common to multiple objects. Very often, the Domain Model will have identified the candidate inheritance relationships and these can be copied in the protocol model. Thus the inheritance structure for various kinds of people and workers shown on the left hand side of Figure

5 in the CMS Spec. has been used to create an identical mixin structure in the protocol model shown in Figure 6. Note the following:

- It is not necessarily a good idea to copy all inheritance relationships in the Domain Model. For instance, the Domain Model shows (Figure 4 in the CMS Spec.) *Car Crash* as a specialization of *Crisis Type*. However, copying this would "hard wire" the different types of crisis that the system can handle into the model. Instead, we have made the Crisis Types "soft", so that new types can be introduced into the system at run-time (using the `Create Crisis Type` event).
- Protocol Modeling has natural support for multiple inheritance, whereas Domain Models are commonly built using only single inheritance and are correspondingly constrained in the specialization/generalizarion relationships that can be shown. For instance, *Mobile Employee* and *Vehicle* should (in our view) both be specializations of *Resource*, but this would require Mobile Employee to specialize from two parents. This constraint does not exist in Protocol Modeling, and in our model `Mobile Employee INCLUDES` (= specializes) both `CMS Employee` and `Internal Resource`.

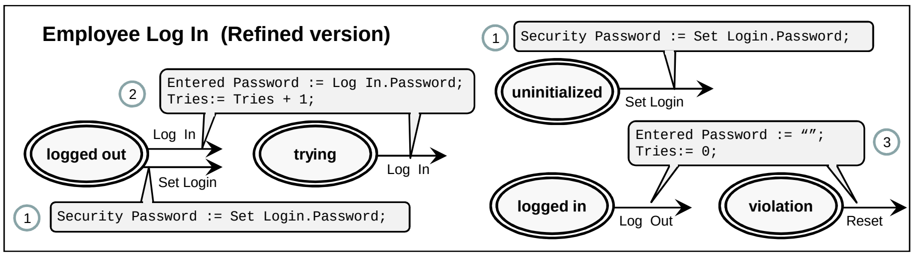
4.4 Step 4. Refine Machine Behavior

Normally the behavior built into a model in the early versions is a simplification of the actual requirement, and is refined and improved in later steps. This refinement can involve refining the definition of a machine already defined in the model, or adding new machines.

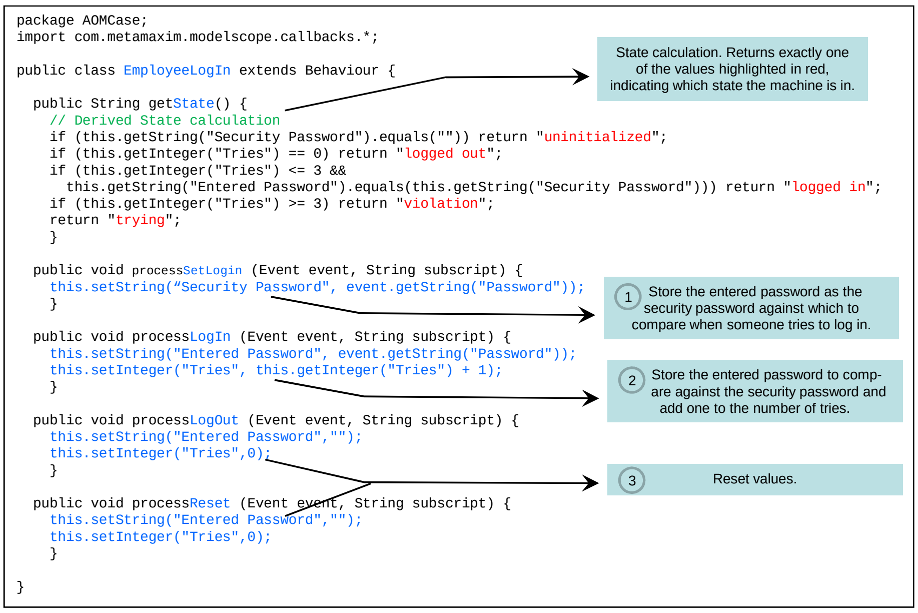
For instance, in Figure 5 the machine `Employee Log In` (near the top right of the diagram) is a naïve description of logging on, as it assumes that the `Log In` event will always work. This may be a reasonable assumption at early stages, so that basic testing of the model can take place, but does not take into account the fact that the password must be checked, or that there is a limit on the number of attempts that may be made, or the possibility that no password has yet been setup. To model these, the simple machine shown in Figure 5 is replaced by the one shown in Figure 3. This has the same two states as the original (`logged in` and `logged out`) but adds new states `unitialized` (for the case where no password has been set up), `trying` (for the case where the user is trying to enter a correct password) and `violation` (for the case where the user has exceeded his\her attempt limit). The new machine has a derived state rather than a stored one, but whether a state is derived or stored is private to a machine so does not complicate the substitution. Finally, note that the new machine requires a new event `Reset` to reset the machine if it is in the violation state.

4.5 Step 5. Model Event Automation

The steps so far have assumed that all events are initiated from outside the system. However, systems commonly initiate some events internally. For example,



Employee Log In (Refined version)
1
Security Password := Set Login.Password;
uninitialized
Set Login
2
Entered Password := Log In.Password;
Tries:= Tries + 1;
logged out
Log In
Set Login
trying
Log In
1
Security Password := Set Login.Password;
Entered Password := “”;
Tries:= 0;
3
logged in
Log Out
violation
Reset



```
package AOMCase;
import com.metamaxim.modelscope.callbacks.*;

public class EmployeeLogIn extends Behaviour {

  public String getState() {
    // Derived State calculation
    if (this.getString("Security Password").equals("")) return "uninitialized";
    if (this.getInteger("Tries") == 0) return "logged out";
    if (this.getInteger("Tries") <= 3 &&
      this.getString("Entered Password").equals(this.getString("Security Password"))) return "logged in";
    if (this.getInteger("Tries") >= 3) return "violation";
    return "trying";
    }

  public void processSetLogin (Event event, String subscript) {
    this.setString(“Security Password", event.getString("Password"));
    }

  public void processLogIn (Event event, String subscript) {
    this.setString("Entered Password", event.getString("Password"));
    this.setInteger("Tries", this.getInteger("Tries") + 1);
    }

  public void processLogOut (Event event, String subscript) {
    this.setString("Entered Password","");
    this.setInteger("Tries",0);
    }

  public void processReset (Event event, String subscript) {
    this.setString("Entered Password","");
    this.setInteger("Tries",0);
    }

}
```

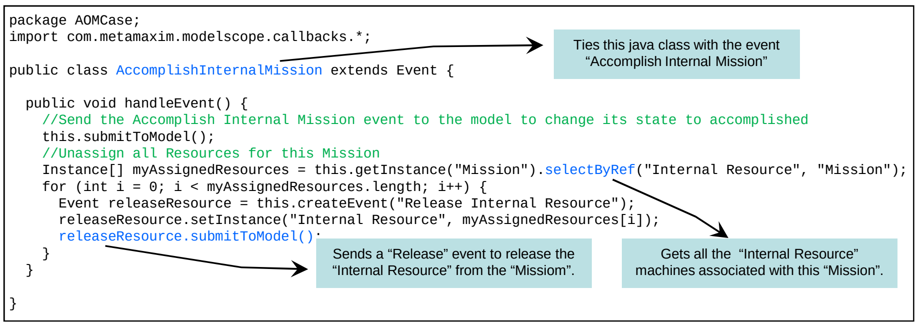
State calculation. Returns exactly one of the values highlighted in red, indicating which state the machine is in.

1 Store the entered password as the security password against which to compare when someone tries to log in.

2 Store the entered password to compare against the security password and add one to the number of tries.

3 Reset values.

Fig. 3. Refined Employee Log In Machine



```
package AOMCase;
import com.metamaxim.modelscope.callbacks.*;

public class AccomplishInternalMission extends Event {

  public void handleEvent() {
    //Send the Accomplish Internal Mission event to the model to change its state to accomplished
    this.submitToModel();
    //Unassign all Resources for this Mission
    Instance[] myAssignedResources = this.getInstance("Mission").selectByRef("Internal Resource", "Mission");
    for (int i = 0; i < myAssignedResources.length; i++) {
      Event releaseResource = this.createEvent("Release Internal Resource");
      releaseResource.setInstance("Internal Resource", myAssignedResources[i]);
      releaseResource.submitToModel();
    }
  }
}
```

Ties this java class with the event “Accomplish Internal Mission”

Sends a “Release” event to release the “Internal Resource” from the “Mission”.

Gets all the “Internal Resource” machines associated with this “Mission”.

Fig. 4. Automated Events to Release Resources on Mission Accomplishment

when a Mission is reported as accomplished, the *System* could automatically release all resources assigned to the mission. This kind of event automation requirement is addressed by associating a piece of logic with an event, executed when an event is accepted by the model that generates further automated events. This is indicated by prefixing the `EVENT` entry with `!`. Thus `!Accomplish Internal Mission` has logic associated with it as shown in Figure 4.

Other examples of use of this technique in the CMS model are:

- Adding a Witness if the initial report of the Crisis contains a witness report (attached to `!Create Crisis`).
- Determining what kinds of Internal and External Mission are needed for a Crisis and creating recommendations to the Coordinator for such Missions (attached to `!Make Recommendations`).
- Finding a suitable Internal Resource for a newly confirmed Mission and creating a Request Allocation (attached to `!Confirm Mission`).
- Finding another suitable Internal Resource if a request for assignment is declined (attached to `!Decline Request`).

4.6 Step 6. Define the Actors

As pointed out Section 3.6, the actor responsibility for events is modeled separately from the underlying protocol machine model, using `ACTORS`. This is normally done once the model is substantially complete but before validating against Use Cases.

4.7 Step 7. Validate against Use Cases

The final stage is to validate the model against Use Cases. Generally, we take the view that Use Cases should be viewed as test cases whose satisfaction is necessary (but not sufficient) for acceptability of the model.

Sometimes validation of the protocol model behavior against Use Case description can reveal the need to nuance the behavior of the system by adding *modal machines*, as described in Section 2.5. For example, Use Case 3 Step 1a.1 in the CMS Spec. says that “System requests the CMSEmployee to login” and this is represented as a modal machine, `Want Log In`, which determines that the employee needs to log into the system in order to either accept or reject requests to be assigned to a mission. In ModelScope, an event that has been determined to be needed by a modal machine is highlighted in green in the user interface.

5 Protocol Model of the CMS Case Study

Our protocol model of the CMS case study is illustrated graphically in Figures 5 and 6. This graphical representation is intended as an overview and shows all the machines of the protocol model, their states and their protocol (what events are allowed in each state). It does not, however, show how data are handled:

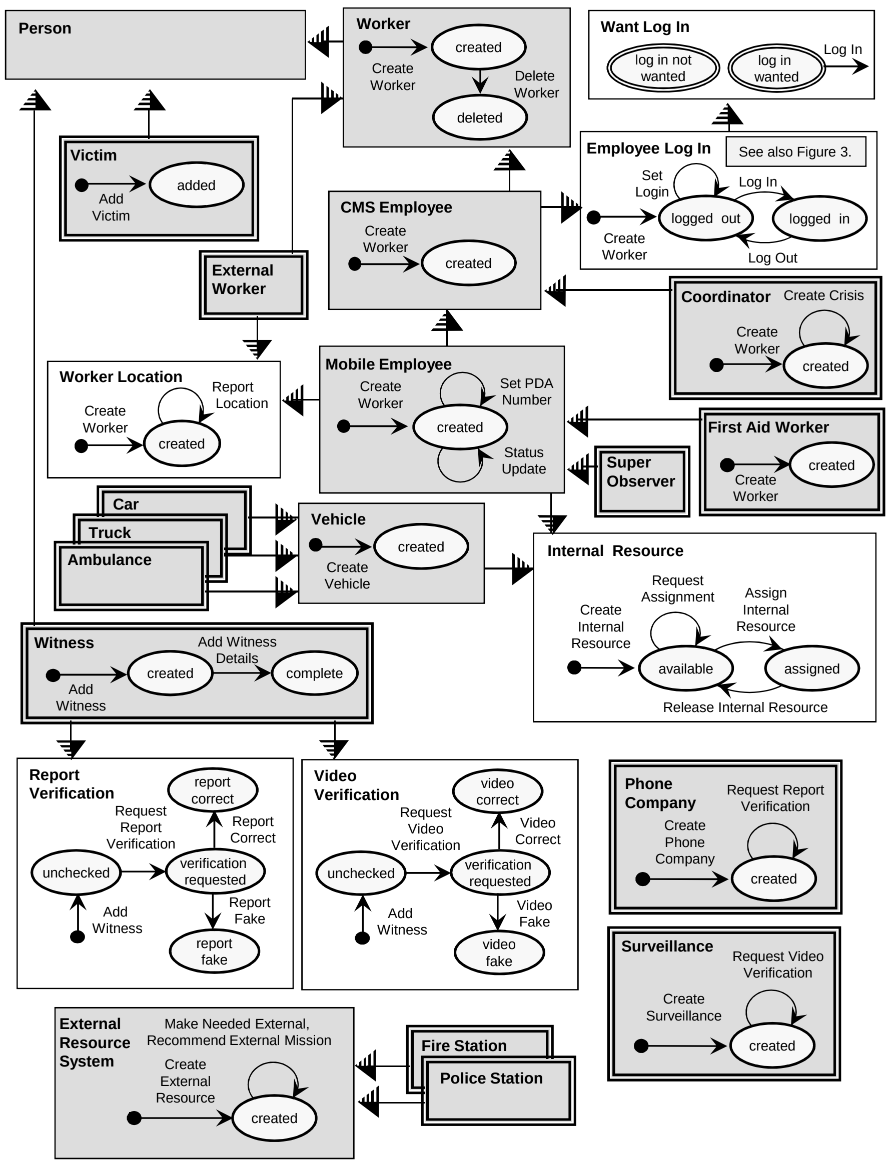


Fig. 5. Protocol Model of the Crisis Management System (1 of 2)

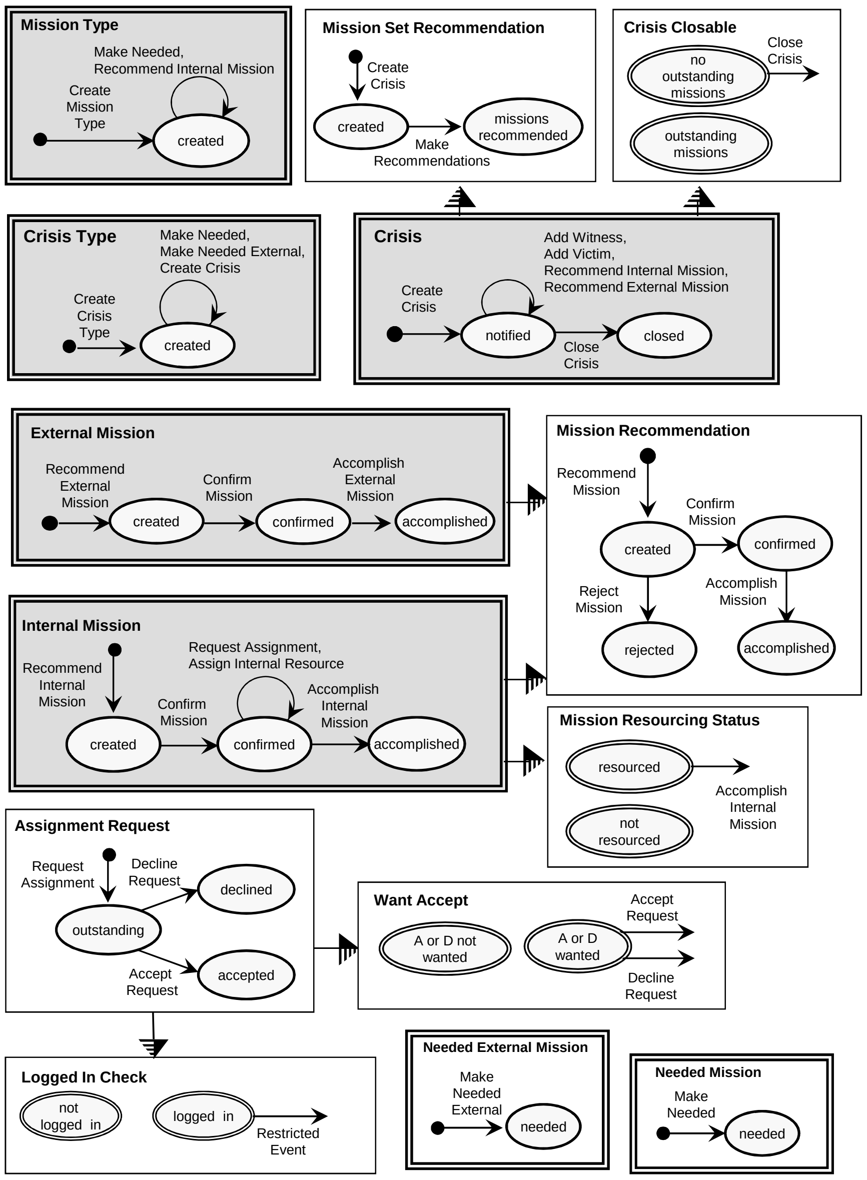


Fig. 6. Protocol Model of the Crisis Management System (2 of 2)

it omits attribute definitions, calculations for derived states and attributes, and the algorithms for generation of automated events. The full source of the model, including the full definition of data handling, can be downloaded from the Metamaxim website [6][10]. The ModelScope execution tool is also available from this site.

5.1 CMS Domain Model

Most of the machines in our model correspond exactly to the classes in the Domain Model given in [20]. These are the *shaded* machines shown in Figures 5 and 6 of this paper, and their `INCLUDE` structure, shown using semi-hatched arrows, corresponds to the inheritance hierarchy of the Domain Model.

The other, *non-shaded*, machines Figures 5 and 6 of this paper fall into two categories. Some are further domain objects that we have identified as needed to support functionality required in the CMS application. Others represent behavior that is sensibly modeled separately from the main behavior of an object and then included as a separate machine. In some cases this separated behavior is shared by more than one domain object, in which case it is included in all those to which it relevant. Two examples from Figure 5 in the CMS model are:

- The machine `Internal Resource` that is included in `Mobile Employee` and `Vehicle`. This machine models the ability of an object to be requested for, and then possibly assigned to, a mission.
- The machine `Worker Location` that is included in `Mobile Employee` and `External Worker`. This machine models the ability of a mobile worker to report his\her current location.

In both cases, the included machine handles an aspect of behavior that is separate and essentially orthogonal to that of the host machine. Because a protocol machine recognizes events from its own alphabet and ignores other events, it is both possible and natural to identify such *behavioral aspects*, concerned with different subsets of the overall alphabet of the object, and model them as separate machines. The resulting structure of behavioural aspects supports multiple inheritance of both data and behavior. In this sense, behavioral aspects in Protocol Modeling are not a "bolt on" added to model cross-cutting concerns, but the prime tool of modeling.

5.2 CMS Use Cases

As we described in Section 4.7, we use Use Cases to validate that the behavior in the model is able to address the various ways in which it is intended that the system will be used. We have verified our protocol model against all the Use Cases using this technique. We illustrate below how our protocol model meeds the requirements expressed in the three of the key Use Cases in the CMS Spec.

[10] The model is available at: www.metamaxim.com/download/models/CMS.zip

UC1: "Resolve Crisis" .

UC1.1 *Coordinator captures witness report.*
Event `Create Crisis` allows a *Coordinator* to set up a `Crisis` and capture a `Witness` report at the same time. Further reports may be added using event `Add Witness`.

UC1.2, UC1.3 *System recommends to Coordinator the missions that are to be executed based on the current information about the crisis and resources. Coordinator selects one or more missions recommended by the system.*
When the *Coordinator* creates a `Crisis`, the included machine `Mission Set Recommendation` is also created. Using the event `Make Recommendations` the *Coordinator* can ask the System to create an appropriate set of `Internal Mission` and `External Mission` machines. (The types of mission recommended for a crisis is based on information previously set up to define the needed `Mission Types` for each `Crisis Type`.) Each mission machine created includes a `Mission Recommendation` machine that keeps track of whether the mission is confirmed or rejected. This machine offers two events, `Confirm Mission` and `Reject Mission`, allowing the *Coordinator* to select which of the system recommended missions are actually to be executed. Because the behavior for recommending or rejecting missions is identical for both internal and external missions, we use a common behavior `Mission Recommendation` and generic events `Confirm Mission` and `Reject Mission` for both.

UC1.4 *For each internal resource required by a selected mission, System assigns an internal resource.*
When the *Coordinator* uses `Confirm Mission`, the system finds an available `Mobile Employee` with the right expertise (found in his attributes) and creates an `Assignment Request` machine to request assignment of the employee to the mission. This machine allows the *Employee* to either `Accept Request` or `Decline Request`. If the *Employee* declines the request, the *System* automatically finds another candidate and instantiates a new `Assignment Request` machine for this candidate.

UC1.5 *For each external resource required by a selected mission, System requests an external resource.*
This step is similar to UC1.4.

UC1.6. *Resource notifies System of arrival at mission location.*
When an instance of machine `Mobile Employee` is created an instance of the included machine `Worker Location` is created too. This machine allows an *Employee* to use event `Report Location` to notify the system of his\her arrival at mission location.

UC1.7. *Resource executes the mission.*
An *Employee* uses event `Status Update` to update his\her own status and the status of the mission.

UC1.8. *Resource notifies System of departure from mission location.*
An *Employee* uses the event `Report Location` to report departure.

UC1.9. *In parallel to steps 6-8, Coordinator receives updates on the mission status from System.*
The information available from the events `Report Location`, `Status Update` and `Accomplish Mission` are available to the *Coordinator*.

UC1.10. *In parallel to steps 6-8, System informs Resource of relevant changes to mission (crisis) information.*
The information available from the events `Report Location`, `Status Update` and `Accomplish Mission` are available to the *Employees*.

UC1.11. Resource *submits the final mission report to* System.
The *Employee* uses event `Accomplish Mission` to submit a final report.

UC1.12. *In parallel to steps 4-8,* Coordinator *receives new information about the crisis from* System.
The events `Add Witness` and `Add Victim` of machine `Crisis` are used to add further information about the crisis.

UC1.13. *Coordinator closes the file for the crisis resolution.*
The *Coordinator* uses event `Close Crisis` in machine `Crisis` to close the file. Event `Close Crisis` is possible when there are `no outstanding missions`. This state is derived by machine `Crisis Closable`.

UC2: "Capture Witness Report".

UC2.1, UC2.2. *Coordinator provides witness information to System as reported by the witness. Coordinator informs System of location and type of crisis as reported by the witness*
These are modeled by the protocol machine `Witness` with the corresponding events `Add Witness` and `Add Witness Details`.

UC2.2a.1, UC2.2a.2. *System contacts PhoneCompany to verify witness information. PhoneCompany sends address\phone information to System* and the extended requirement UC2.5a. *PhoneCompany information does not match information received from Witness.*
These are modeled as machine `Report Verification` which is instantiated with acceptance of event `Add Witness`.

UC2.2a.2 and UC2.5a represent different outcomes of witness verification modeled as events `Report Correct` and `Report Fake` leading to the corresponding states `report correct` and `report fake`.

UC2.3a.1, UC2.3a.2 *System requests video feed from SurveillanceSystem. SurveillanceSystem starts sending video feed to System* and UC2.3a.3 *System starts displaying video feed for Coordinator.*
These are modeled by machine `Video Verification` which is instantiated with acceptance of event `Add Witness`.

UC2.5b.*Camera vision of the location is perfect, but Coordinator cannot confirm the situation that the witness describes or the Coordinator determines that the witness is calling in a fake crisis.*
This requirement represents a negative outcome of the verification. The video verification should definitely contain a positive outcome of verification which is omitted from the requirements.

UC10: "Authenticate User".

This use case presents a commonly recognized *Security* aspect, modeling password protected access to functionality of a system.

UC10.1. *System prompts CMSEmployee for login id and password.*
This is captured by the state `not logged in` of protocol machine `Employee Log In` either that shown in Figure 5 or its refined version shown in Figure 3.

UC10.2. *CMSEmployee enters login id and password into System.*
This requirements specifies attributes of event `Log In`, namely `Login` and `Password`.

UC10.2a. CMSEmployee cancels the authentication process.
This is modeled by two possibilities. One possibility is that the employee has not yet succeeded in supplying a correct password. In this case he is not logged on and can just walk away. The other possibility is that he has succeeded in supplying a correct password. He is now `logged in`, so can `log out`.

UC10.3.*System validates the login information.*
These requirements are modeled by the refined version of `Employee Log In` shown in Figure 3. The `Log In` event is allowed provided the system is not states `logged in` or `violation`. The state of `Employee Log In` is computed by comparing the security password in the system with the password entered in the `Log In` event. If they do not match but the maximum number of tries (3) has not been exceeded, `Employee Log In` is in the state `trying`. If the passwords are identical and the maximum number of tries has not been exceeded `Employee Log In` is in the state `logged in`.

UC10.3a, UC10.3a.1, UC10.3a.1a. *System validates the login information. System fails to authenticate the CMSEmployee. CMSEmployee performed three consecutive failed attempts.*
If the security password in the system with the password entered do not match and the maximum number of tries (3) has been exceeded, `Employee Log In` is in the state `violation`. The state can only be reset to `logged out` by using the `Reset` event.

5.3 Some Observations

A requirements document is never perfect, and building a protocol model helps identify gaps and inconsistencies. In particular, the discipline of identifying the protocol states of every object, and determining the events that cause entry and exit from every identified state, helps to ensure the completeness and coherence of the behavioral requirements. The illustrations of this from the CMS case are:

– No means is provided for defining how the system should determine what missions to recommend for a crisis (UC1.2, UC1.3). The system must have knowledge of the relationship between crisis types and mission types to do this.
– Video verification (UC2.3a.2 and UC2.5b) should require a positive confirmation of verification. No means of doing this is given.
– The need for a `Reset` (or a time out) (UC10) if the number of password tries is exceeded, is not discussed in requirement document.

6 From Model to System

While the protocol model of a system is an executable artifact, it is not in general deployable as the final system. Work is required to create an implementation that has the appropriate physical characteristics. Information from the protocol model can be directly used in this process, and an example is the extraction of an Entity Relationship Model.

The attributes of the machines, given in the `ATTRIBUTES` entries of the machines in the model, are examined to identify those that represent "foreign key" pointers. These are the attributes that are typed using other machines: for instance `Expertise: Mission Type` in `Mobile Employee`. Such foreign key attributes can be used to create the "entity relationship" model shown in Figure 7, for instance the `Expertise` attribute results in the relationship marked with a star. This model can then be used as the basis of database design using traditional design techniques. The extraction of this model from the protocol model can be automated.

The techniques and mechanisms used to convert a protocol model into a final system design depends on a number of factors, including the technology platform and the need to integrate with a pre-existing software architecture. Sometimes the process can be entirely mechanized and sometimes it cannot. Discussion of this is beyond the scope of this paper.

7 Aspect-Orientation of Protocol Modelling

In this section we discuss the claim of Protocol Modeling to be an aspect-oriented approach, and the way in which aspects have been used in our solution to the case study.

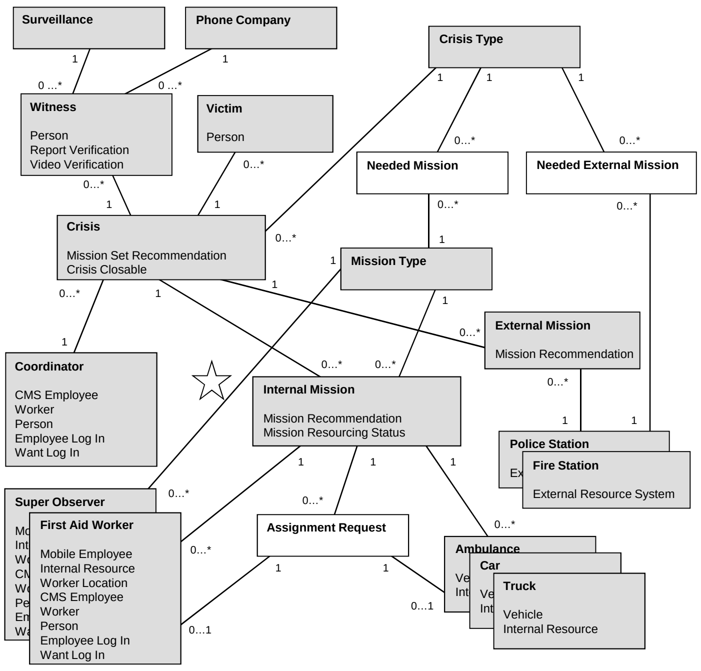


Fig. 7. Entity-Relation Model of the Crisis Management System

7.1 CSP Composition as Weaving

The basis for any claim to support aspects is a notion of *weaving*, whereby two independently defined parts of a model are combined but without explicit invocation, such a method or subroutine call. The mechanism for composition in Protocol Modeling is CSP composition, as described in Section 2.6. The semantics of CSP composition can be thought of as “trace weaving” whereby the sets of traces of individual protocol machines are woven to form the set of traces of the system as a whole. Because it works by synchronizing machines on events that are in the alphabets of both, CSP weaving uses *events* as join points. It is in these terms that we discuss the aspect-orientation of Protocol Modeling.

7.2 Quantification of Event

Events in a protocol model may be described at different levels of abstraction using `GENERIC` events. This has two uses:

– Where a number of different event types have the same treatment and effect in a given context. The difference between those event types may be abstracted away in contexts where the difference is immaterial.
– To facilitate re-use, by allowing the creation of a generic definition that has different interpretations in different contexts.

An example is of the first of these is `Restricted Event`, defined as:

```
GENERIC Restricted Event
# Events only allowed when an Employee is logged in
   MATCHES Accept Request, Decline Request
```

This abstraction is used in the machine `Logged In Check` at the bottom left of Figure 6 to model the fact that certain events are restricted (not possible) unless an `Employee` is logged into the system, but `Logged In Check` doesn't itself need to differentiate between the different kinds of restricted event.

An example of the second is `Create Internal Resource`, defined as:

```
GENERIC Create Internal Resource
# Events that can create an Internal Resource
   MATCHES Create Worker, Create Vehicle
```

When the model of is run in ModelScope the following messages relating to it are generated as the model is compiled:

```
Generic 'Create Internal Resource' in object 'Ambulance' expands to: Create Vehicle
Generic 'Create Internal Resource' in object 'Car' expands to: Create Vehicle
Generic 'Create Internal Resource' in object 'First Aid Worker' expands to: Create Worker
Generic 'Create Internal Resource' in object 'Super Observer' expands to: Create Worker
Generic 'Create Internal Resource' in object 'Truck' expands to: Create Vehicle
```

Here the compiler is interpreting this generic event appropriately for the context[11].

Most of the weaving that takes place in a protocol model is defined in terms of the event types of the domain. In this case, the weaving is completely symmetric and there is no formal way of distinguishing the "base behavior" and the "advice". However, as these examples show, the use of generic events provide for a degree of *quantification*, whereby a single join point in the model, specified as the label on a transition, can match multiple labels in other machines and/or can match a different label depending on context. When generic events are used, there is asymmetry between the machines that are woven and we can (perhaps) identify the machine that uses the generic as representing the "advice".

In the CMS case study there are 4 uses of generic events.

7.3 Symmetric approach

One view of aspect-oriented software development is that *every major feature* of the system: core concern (business logic), or cross-cutting concern (additional features), is an aspect, and by weaving them together (a process also called composition), you finally produce a whole out of the separate aspects. This approach is known as the *symmetric (or pure)* aspect approach. However *asymmetric (or hybrid)* approaches, where a base model is built using one technique and aspects are applied to it using a specialist aspect language, are more commonly

[11] This uses a technique in Protocol Modeling called *conditional repertoire entries*, which is described fully in [7]. This is a form of polymorphism.

```
package AOMCase;
import com.metamaxim.modelscope.callbacks.*;

public class LoggedInCheck extends Behaviour {

  public String getState() {
  //Determines whether the Employee is logged in or not
    return (this.getInstance("Internal Resource").getState("Employee Log In").equals("logged in"))
      ? "logged in" : "not logged in";
    }

}
```

Fig. 8. State Derivation in Logged In Check

used perhaps since then there is less of a paradigm shift between object- and aspect-orientation[12].

Within this classification, Protocol Modeling is clearly a symmetric aspect-oriented approach, as there is no difference in syntax and semantics between a protocol machine that models an aspect and any other protocol machine in the model.

7.4 Derived States as Join Points Refinements

It is worth mentioning that using derived states can help provide economy and clarity in the expression machine behavior and interaction under composition. For instance, the state derivation function of the machine `Logged In Check` shown in Figure 8 abstracts over the states of the machine `Employee Log In` (the refined version shown in Figure 3). Thus the state `not logged in` of `Logged In Check` represents any of the states {`logged out, trying, violation`} of `Employee Log In`. This machine can then be used to make a more succinct definition of the constraints that being logged in versus not logged in has on the behavior of the system as it only expresses these two states.

7.5 Mixins as Aspects

When modeling a system, the structure of the model will be dominated by the model architect's view of the key elements of the model and their relationships and connections. The more a core modeling language imposes and constrains the structure of a model, the less likely that all the elements of a problem and their relationships can be accommodated, as the structure chosen for some elements necessarily means that others cross-cut. This is sometimes referred to, for instance by Ossher and Tarr, as the "Tyranny of the Dominant Decomposition" [17]. This problem is immediately manifest with single inheritance structures, where the single hierarchy of the class structure is the tyrant, and the modeler has to resort to using a *asymmetric approach*, with a specialist aspect language being used to model those elements which clash with the main structure of the model.

[12] See the Wikipedia entry on "Aspect" at:
`http://en.wikipedia.org/wiki/Aspect_(computer_science)`.

This is, in general, not the case with a mixin based inheritance approach supporting multiple inheritance. As Filman and Friedman observe: "In using inheritance to achieve aspects, single superclass inheritance systems require all aspects to match the class structure of the original program, while multiple inheritance systems allow quantification independent of the programs dominant decomposition. Mixins with multiple inheritance are thus a full aspect-oriented programming technology"[25]. A similar observation is made by Apel et al. [28]. The use of mixin-based inheritance in Protocol Modeling qualifies it as aspect-oriented. This is true in particular because the means by which mixins are combined is CSP composition, and this entails weaving of trace behavior. However, the aspects are pervasive in the model rather than being invoked as an exceptional technique to handle cross-cutting concerns.

To avoid structural tyranny it is necessarily to minimize the structure constraints imposed by the inheritance scheme on a model, and the partial ordering of Protocol Modeling (see Section 3.2) is the minimum structuring possible. As applied to the CMS case study, examination of the structures in Figures 5 and 6 shows that multiple inheritance, measured formally as the definition of machine type that `INCLUDES` more than one other machine type, is used 7 times.

8 Protocol Modeling in Practice

This section describes various practical considerations in the use of Protocol Modeling, both positive and negative.

8.1 Barriers to Protocol Modeling

The difficulties with application of Protocol Modeling fall into two categories, one associated with the Protocol Modeling paradigm (1) and the other with the nature of the problem which is to be addressed (2).

1. Paradigm Related. The Protocol Modeling paradigm is not well aligned to the prevalent paradigm of current mainstream modeling languages (UML) and programming languages (Java, `C#`, `C++`):

- The use of pure mixin based approach, contrasting with the prevalent focus on inheritance;
- The use of process algebraic composition (CSP);
- The idea that the state of a state-transition machine may be calculated, rather than simply driven by transitions;
- The idea of using modal semantics to model workflow.

These entail different ways of thinking so becoming fluent in the technique requires time and effort, analogous to that required to convert from imperative programming to OO programming. Moreover, the conceptual distance between Protocol Modeling and current programming languages means that mapping a protocol model to an implementation is not trivial. For instance, if the target programming language does not support multiple inheritance, other means (which could include Aspect Oriented Programming techniques) may be required to implement a model.

2. Problem Related. The second consideration is the nature of the problem being addressed. Protocol Modeling has its own domain of applicability, which can be roughly characterized as discrete event based systems which are based on what Jackson, in his work on "Problem Frames" [22], refers to as an *analogic model.* Examples of systems that do not conform to this characterization are: a word processing system, a compiler and a chess game; and attempts to apply Protocol Modeling to such problems will be problematic and unlikely to give a useful solution.

8.2 Model Execution and Testability

Protocol models can be directly executed (tested) after any step of model evolution, and this is supported by the ModelScope [6] tool, providing run-time machine composition and a generic (metadata driven) user interface. The input of the tool is the textual presentation of a protocol model described in Section 3, and Figure 9 shows the user interface as it appears when executing the CMS case study. The tests are recorded in a test-file.

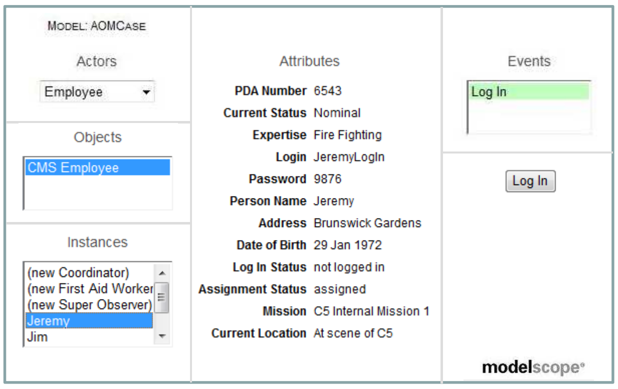
The value of providing stakeholders with working (executable) artifacts early in the development process is well recognized. By presenting a working representation of the intended system, model execution offers the potential to make early modeling accessible to stakeholders *even those who are not familiar with technical modeling notations*, and thus widens the circle of participants in review activities. Such widening of the review activities can help to expose and eliminate misunderstandings between the development and user communities early in the development lifecycle, when mistakes are still relatively cheap to correct, and thus reduce risk and improve quality. The use of ModelScope in this context is discussed fully in [4].

The tests are used to validate the model against use cases. Tests can also validate other properties formulated by the developer. For example, "A Crisis case can be closed only if all its missions have been accomplished". Testing can help to find counter examples to the properties formulated by the developer and this usually means that the model needs a correction.

8.3 Model Evolution

In practice it is not possible to collect all the requirements for a model at once, so models evolve. Evolution of a protocol model normally entails adding or deleting protocol machines. As all protocol machines are equally composed, the changes are local.

Typically, a behavior model is built in phases, each phase scoped by a Use Case or part of a Use Case. The compositional nature of Protocol Modeling



PDA Number	6543
Current Status	Nominal
Expertise	Fire Fighting
Login	JeremyLogIn
Password	9876
Person Name	Jeremy
Address	Brunswick Gardens
Date of Birth	29 Jan 1972
Log In Status	not logged in
Assignment Status	assigned
Mission	C5 Internal Mission 1
Current Location	At scene of C5

Fig. 9. ModelScope Generic User Interface

makes this kind of incremental approach attractive and natural. In the CMS case study, initial phases of modeling might be as follows:

– **Phase 1**: Events and machines for `Crisis Type`, `Crisis`, `Mission Type` and `Mission`. Events and machines for `Person`, `Worker`, `CMS Employee`, `Coordinator`, and `Mobile Employee`. Development of support for steps 1 - 2 of Use Case 1.
– **Phase 2**: Events and machines for `Needed Mission`, `Mission Recommendation` and development of support for steps 3 - 5 of Use Case 1.
– **Phase 3**: Events and machines for `Internal Resource` and `Assignment Request`. Development of support for steps 1 - 2 of Use Case 3.
– **Phase 4**: Events and machines for `Witness`, `Victim`, `Phone Company`, `Surveillance`, `Report Verification` and `Video Verification`. Development of support for the steps 1 - 2 of Use Case 2.

In general, each increment will go through the stages of development described in Section 4. After each increment the model is tested and, at key stages, validated with users for correct interpretation of requirements. The validation process allows users and other stakeholders to play scenarios through the model.

8.4 Scalability

Scalability concerns the ability of a technique to be applied to large, complex problems without exhibiting non-linear growth in complexity [1]. In a compositional modeling approach such as Protocol Modeling, scalability can discussed in terms of the following questions:

1. How does the size of the state space of the solution grow as more complex problems are modeled? In particular, can the approach prevent the phenomenon of "state space explosion" resulting from combinatorial effects?
2. Is the author of the model able to maintain intellectual control over a model as its size grows?

We discuss these in turn.

1. State Space. The ability of state transition approaches to scale to complex behavior requires avoidance of the "state-space explosion", whereby the number of states that have to be used to describe a problem increases geometrically rather than linearly with the size of the problem. Protocol Modeling is not vulnerable to this phenomenon, as the new states of an object needed to describe new behavior are added in new machines. This is in contrast with some other state-transition approaches, such as that suggested by Mellor and Balcer [32], which do not support composition so that all important state combinations of an object must be represented as a single machine.

The size of state space of individual protocol machines is dictated by the preferences of designers. Usually, the machines of a model follow the psychological restriction that a person cannot control and reason at one moment more than nine elements. This can be seen in the model in Figures 5 and 6 whose individual machines do not use more than five states.

2. Intellectual Control. In Protocol Modeling, the key to maintaining intellectual control over a model as it grows is the ability to do "local reasoning": to reason reliably about the behavior of the whole from examination of a part in isolation. If the use of aspect technologies results in specifications becoming distributed through multiple design artifacts (the base behavior or code definitions and separate aspects that have been added to them) in such a way that no reliable deductions about the behavior of the whole can be made from examination of a part, then there is little chance of maintaining intellectual control over a complex model.

Local reasoning in Protocol Modeling is based on the following property of CSP composition: If we take a sequence, S, of events that is accepted by the composition $(M_1 \parallel M_2)$ of the two machines M_1 and M_2, then the subsequence, S', of S obtained by removing all events in S that are not in the alphabet of M_1 would be accepted by the machine M_1 by itself. This property is sometimes known as *observational consistency* [18] between a composite and its component machines. The fact that CSP composition gives observational consistency was established by Hoare [10]; however, Hoare's formulation was based on the composition of algebraic processes and did not consider machines that access the local storage of other machines (in particular, to derive their state) and a proof extended to cover composed protocol machines is given in [3].

These ideas are very closely related to the categorization of aspects suggested by Katz [31]:

- *Spectative* aspects can change the values of variables local to the aspect, but do not change either the value of any variable or the flow of method calls of the underlying system.
- *Regulative* aspects can affect the flow of control of the underlying system by restricting operations, delaying some operations, or preventing the continuation of a computation.
- *Invasive* aspects can change the values of variables and therefore the behavior in the underlying system.

Katz was also concerned with the issue of how to reason about software behavior in the presence of aspects, and argues that Invasive aspects make reasoning hard or impossible. The fact that CSP composition cannot break trace behavior means that protocol machines **cannot** be Invasive.

Another way of stating the *observational consistency* property is: composing another machine with M_1 cannot "break its trace behavior" by overriding a constraint that M_1 says must be true. The property can be used to support local reasoning thus: If the sequence S' were *not* acceptable to M_1, the original sequence S *could not* have been acceptable to $(M_1 \parallel M_2)$. This means that we can use properties of M_1 (or M_2) *alone* to argue about the behavior of $M_1 \parallel M_2$. This is key to retaining intellectual control. When creating a model, the author need only ensure that the protocol *constraints in each machine of the model are true by examining each machine separately*, as composition preserves the constraints specified by each machine. This allows intellectual control to be maintained over a model even if it comprises a large number of fine grained machines.

8.5 Correctness

By correctness we mean the use of formal proof techniques to establish behavioral properties in a model. These are properties such as *liveness* or *absence of deadlock* and are normally addressed using *model proving* techniques. This is a large topic, and we only give a summary here.

Our view is that the approach needed to establish behavioral correctness depends on the nature of the system. The key determinant is whether the system is:

1. *Deterministic* in which case local reasoning about models combined with model execution (testing) is generally sufficient for ensuring correctness. This is because with a deterministic system you can repeat tests.
2. *Non-deterministic* in which case you must use **global reasoning** (model checking) to ensure correctness. This is because the execution path of the system cannot be determined in advance or controlled during execution, and consequently repeatability of behavior cannot be assured. This makes systematic testing hard or impossible.

Within the realm of Protocol Modeling, models fall into two classes corresponding to the above:

– Protocol models that only use the deterministic parallel composition of CSP[13].
– Protocol models that use composition operators that introduce non-determinism, such as the composition operator of Robin Milner's CCS (or the π-calculus) [26].

We discuss these in turn.

[13] As Hoare himself notes: "...the concurrent operator by itself does not introduce non-determinism." [11].

1. Deterministic Models. The CMS case study (at least as we have modeled it for this paper) falls into the first class of deterministic models. Here the ability to apply local reasoning to the model and use a tool such as ModelScope is generally enough to establish correctness. Both techniques are essentially needed. The CSP parallel composition and its property of local reasoning guarantees that traces of parts will not me reordered by the the composition. Model execution provides an extra check that the composition of Protocol Machines produces desired behavior and does not unnecessary behavioral constraints.

CSP composition always works, in that it never produces a composite that fails in execution (generates a run-time error). However, it is possible to create machines that interlock with each other. Consider the two machines:

$$P = x.y \text{ and } Q = y.x \text{ where “.” indicates sequencing of events.}$$

In this case $P \parallel Q$ will refuse both x and y and so will not be able to engage in any event. This won't generate a run time error: it just won't do anything.

It might be possible to devise a static analyzer that could detect such situations. However, as there are often cases in a model where refusal of the events is the intention (e.g., as `Logged In Check` purposefully refuses `Restricted Events` when the `Employee` is not authenticated by the system) such an analyzer could only detect candidate problems: for instance by using model checking to find states of the system in which composition causes all exit transitions to be refused. We have not found this to be necessary as, in practice, model testing is sufficient to detect such mistakes in the model by exercising the required Use Cases.

2. Non-Deterministic Models. When modeling software that is required to be distributed across multiple (real or virtual) unsynchronized processors it is necessary to use composition operators that result in non-deterministic behavior and then correctness must, in general, be established using *model proving* techniques. Model proving generally requires that the multiple machines of a protocol model be combined into a single machine for the purpose of global analysis of the possible execution paths: for instance to establish that no path ends in a state from which there no exit transition (a deadlock). This is usually done with the help of specialist model-proving tools.

This application of model proving to Protocol Modeling is a subject of current research and beyond the scope of this paper, but the following general statements can be made:

– Protocol models that comprise a fixed population of machines lend themselves well to traditional model proving techniques. The general approach is to form a state space of the model as whole as the Cartesian product of the state spaces of the constituent machines of model, and then draw the valid transitions between the states of this global state space according to the rules of the composition operators (normally both CSP and CCS) used in the model. The resultant overall machine can then be analyzed for the presence (or absence) of topological properties corresponding to desired (or pathological) behavioral properties. The proving is a mechanical process that can be carried out by a software tool.

- As described in Section 7, derived states can be used to abstract over the data and states of a model and hence maximize the economy of the state space required. This reduces the chance that model proving becomes untractable because of the size of the global state space that needs to be constructed and explored.
- Because, as noted in Section 2.4, derived state machines are not “topological” extra steps are sometimes needed in the model proving process to convert these machines into a form that allows them to be included in the analysis.

These techniques have useful application. For instance, the combination of model proving and the ideas described in Section 2.5 for representing modality makes it possible to conduct formal progress analysis on distributed (multi-party) collaborative workflows whereby it is possible to establish analytically that a collaboration will always reach a successful conclusion. Another example of formal proving techniques using protocol models is the work by McNeile on choreography realizability [2].

9 Related Work and Conclusion

In this section we make an overview of some related aspect based modeling approaches and draw some conclusions.

9.1 Related Work

We have chosen the workflow based Theme approach, the RAM approach using multiple behavioral views and approaches that use StateCharts for capturing of aspects, to show that composition semantics may enrich those approaches and make them more flexible.

Theme. The Theme [29, 13] approach is used both at requirements and the design level. At the requirements level Theme/Doc exposes the relationship between behaviors (features) in the model. The Design level Theme/UML supports modeling features and aspects in UML. Theme/Doc views are mapped onto the Theme/UML model allowing traceability of requirements.

At the requirements level a designer identifies actions as verbs in textual requirements. Each action potentially becomes a *theme.* The actions have relations via concepts. Actions found in different themes become potential aspects. An action view of each theme is a graph that contains actions and entities. The size of this view grows with the number of entities and actions in the model. The scalability is achieved by the separation major and minor actions. “Major actions become themes, while minor actions are slotted to become methods within a theme” [13].

The action view “drives composition semantics for design in Theme/UML” [13]. The action view is analyzed to produce the Theme/UML specification of actions-aspects as a combination of a class and sequence diagrams. So, actions in the Theme approach are not elementary, they are activities. If a Theme/action is reused in the model, the pointcuts are specified in the Theme/UML view.

The evolution of the model is handled by re-generating the views for the new set of requirements. This we see as a shortcoming of this approach.

Reusable Aspect Models (RAM). In the Reusable Aspect Models (RAM) [19] approach a model of a concern or functionality “contains up to three different kinds of views: a structural view, state views and message views - which are grouped together in an aspect model, a special UML package” [19].

The structural view is expressed using a UML class diagram where the specified public methods of classes are annotated with “+”. The structural view of a concern may present only classes and associations relevant to the concern. The structural view may contain “incomplete classes, i.e. entities that are not directly or indirectly bound to model elements of some other aspect models, and methods whose name and signature are yet to be determined”. These incomplete classes are called “mandatory instantiation parameters” and they are recognized by | character attached to their name and as UML template parameters on the right hand side of the structural view compartment [19].

The classes may later be composed by the weaver with other classes when the aspect is instantiated or bound to a base model. The class diagrams are composed using the algorithm proposed in France et al. [27]. The composed elements at an aspect model and a target model must be instances of the same metamodel class and have matched signature, i.e two elements with the same signature represent the same concept and composed.

For each class (complete or incomplete) defined in the structural view one state view is defined. “Using a UML state diagram, the state view of an entity describes the internal states of that entity that are relevant within the concern. A state is relevant if it affects the messages that the entity is capable of processing. In UML terms, the state view compartment describes the usage protocol of the entity. To be complete, the state diagram must contain each method defined in the structural view for the entity at least once [19]”.

For incomplete classes, an aspect state diagram consists of two parts: a pointcut and an advice. “The pointcut defines the states and transitions that have to exist in the target state diagram, i.e. the state diagram with which the aspect state diagram is composed.”

The advice defines the state diagram that replaces the occurrence of the pointcut in the target state diagram. “States that are not directly or indirectly bound to states defined in a standard state diagram are mandatory instantiation parameters of the state view [19]”. So far, weaving of state views is not supported.

For each public method defined in the structural view, there is at most one message view. “Each message view describes, using a UML sequence diagram, the sequencing of message interchanges that occur between entities when providing the functionality offered by the public method. Hence, if the functionality does not involve any message exchanges, but only computation internal to the entity, no message view compartment is shown for that method." A message view has two parts: a pointcut and an advice.

The message views are woven on the basis of the algorithm known as Generic weaving with Kermeta [9, 36]. The result of weaving of sequence diagrams is a sequence diagram.

The RAM approach generates aspect dependencies. The dependencies are declared in the aspect heading. "If A depends on B, A explicitly states that it reuses the functionality provided by B by instantiating B. If an aspect A depends on an aspect B, i.e. an incomplete class X (or | X) in the structural view of an aspect A needs to be composed with a complete class Y defined in B, the state view X in A might also need to refine the state view Y. In this case, A has to define a binding directive that maps the incomplete entities of As structural view, state view or message view into the structural view, state view or message view of the aspect defining Y. Instantiations and binding directives can be one-to-many or many-to-one, if needed" [19]. Aspect dependencies are kept unresolved until the aspects are woven with the final application model. An aspect A can have complex dependencies in form of a directed acyclic graph.

The weaving algorithm resolving aspect dependencies is recursive. It processes the directed acyclic graph of dependencies step by step in depth-first order. All aspects should be woven with a base application model.

Before weaving each aspect model goes through consistency checks, then the adherence of aspect models to the instantiation and binding rules is checked. The third of consistency checks is performed within the independent aspect model and within the final base model."For each object life line in the sequence diagram, the incoming messages to that object are presented in sequence to the state diagram describing the protocol of the corresponding class. If the state diagram refuses a message, consistency is violated" [19]. We should notice that the refuse semantics used for consistency checkers is different from the semantics of refuse in Protocol Modeling. In Protocol Modeling "refuse" is not a violation, it is a normal situation when a protocol machine being in its current state cannot accept an event.

RAM models are not executable. The authors plan to extend the approach "by adding yet another kind of view that describes the detailed execution paths for individual methods. Detailed method algorithms could be expressed, for instance, with UML activity diagrams or SDL. With this additional view, RAM would be capable of generating final application models that are fully executable [19]."

Approaches using Statecharts. There are several approaches using statecharts for capturing aspects.

Mahoney et al. [23] use the semantics defined by D. Harel [12]: “When event a occurs in state A, if condition C is true at the time, the system transfers to state B”. Moreover the authors exploit the *AND-composition* of several independent (orthogonal) statecharts and “the key feature of orthogonal statecharts is that events from every composed statechart are broadcast to all others. Therefore an event can cause transitions in two or more orthogonal statecharts simultaneously” [23].

We should notice that this semantics does not define what happens if one of orthogonal statecharts is in a such a state where it cannot accept the broadcast event. This incomplete semantics does not allow use of CSP or CCS composition for orthogonal statecharts. The most that can be said is that the result of the composition of orthogonal statecharts is a computation tree that represents partial behavior of the system when the orthogonal statecharts are in suitable states to accept broadcasted events.

The UML Specification [24] includes two behavioral semantics for finite state transition systems: *Behavioral State Machines* (BSM) and *Protocol State Machines* (PSM) and several approaches [16, 34] use these as a basis for defining aspect semantics.

High-Level Aspects (HiLA) [16] uses UML State Machines with declarative specification of concerns such as synchronization of orthogonal regions or history-based behaviors. The authors use *Behavioral State Machines* (BSM) semantics. The semantic model used for Behavior State Machine Execution in UML2 (which was first included in UML at version 1.5) is based on the “Recursive Design” method of Shlaer and Mellor [33] whose work has been mainly in the real-time/embedded systems domain. The approach is based on using state machines to model so-called “active objects”: objects whose instances execute autonomously and asynchronously (i.e., as if executing on independent threads) resulting in system behavior that is inherently non-deterministic [35].

The authors of paper [16] notice that “UML state machines work fine as long as the only form of communication among states is the activation of the subsequent state via a transition. More often than not, however, an active state has to know how often some other state has already been active and/or if other states (in other regions) are also active. Unfortunately, behavior that depends on such information cannot be modeled modularly in UML state machines.” We agree with this observation, which our use of derived states avoids.

The asynchronous and non-deterministic composition semantics of BSMs makes reasoning about behavior difficult. Complete analysis of the behavior of the model must allow, in general, for arbitrary queuing of events between objects and for the accumulation of deferred events. If a model comprises a number of communicating objects this results in a large number of possible execution states for the system as a whole, and reasoning on models is impossible without model checking algorithms. This does not make sense when models are being developed, as they are in most projects, in an iterative manner and subject to frequent change; and it is hard to reconcile this semantic basis with the characteristics of the business information systems domain, where behavioral issues are related to transactional integrity and business rules, and strictly deterministic behavior of business logic is important to ensure repeatability, auditability and testability[14].

While there is some native support in Shlaer/Mellor for behavior abstraction through the use of "polymorphic events", this has not been included in the UML BSM standard; nor is there is any method to compose multiple machines to form the behavior of a single classifier. This places severe limits on the ability of BSMs to describe generalization/specialization of behaviors or to support behavior re-use. As described in [32], a single object class is modeled with a single state machine, and only concrete classes are modeled. This also means that crosscutting behaviors (aspects) have to be addressed by other means, potentially further complicating model analysis.

As noted above, UML also supports *Protocol State Machines* (PSM). These are not related to Shlaer/Mellor and have semantics that define the legal lifecycles of a classifier (an object, interface, or port) in terms of the allowable order of invocation of its operations.

PSMs can (to a limited extent) be composed. "A classifier may have several protocol state machines. This happens frequently, for example, when a class inherits several parent classes having a protocol state machine, when the protocols are orthogonal" [24]. In this context, "orthogonal" means that they have a disjoint alphabets.

The occurrence of an event that a PSM cannot handle is viewed as a precondition violation and the consequent behavior of the PSM is left open: "The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a *semantic variation point*: the event can be ignored, rejected, or deferred, an exception can be raised, or the application can stop on an error. It corresponds semantically to a pre-condition violation, for which no predefined behavior is defined in UML" [24]. Only if such a situation were interpreted as a refusal could CSP style composition be supported, but this does not seem to be intention of the specification.

PSM semantics are simpler and more abstract than the BSM semantics, and this makes them more widely usable and easier to analyze. However, as evidenced by the language used to describe them, PSMs are clearly positioned in UML as *contracts of legal usage*; and this gives it a different meaning and role from that of BSMs. While a contract must specify what is legal, it **not** concerned with the mechanism by which non-legal behavior is avoided, nor is it required to specify the effect of violation. In other words: a contract cannot be used as the instrument that guarantees its own satisfaction. It would therefore be a logical error to execute PSMs directly or to generate code from them; and this seems to us to make their use as a means of specifying behavioral aspects problematic.

[14] We note that the commercial tools that support this approach (such as those from Telelogic, Kennedy Carter and Mentor Graphics) are not well adapted for use in the business information systems domain and are positioned by their vendors to target the real time/embedded market.

9.2 Conclusion

In this paper we have shown an approach to aspect modeling using Protocol Modeling, a mixin-based behavioral modeling technique. Protocol Modeling embodies semantics needed to capture and compose behavioral aspects, including: events represented as data, machines with state and local storage, state derivation and formal parallel composition techniques. This enables behavioral aspects and behavior inheritance to be handled by a common mechanism and thus provides a unified view of these different abstraction techniques.

References

1. A. Bondi. Characteristics of Scalability and their impact on Performance. In *Proceedings of the 2nd international workshop on Software and performance, Ottawa, Ontario, Canada*, pages 195 – 203, 2000.
2. A. McNeile. Protocol Contracts with application to Choreographed Multiparty Collaborations. *Service Oriented Computing and Applications*, In Publication, 2010.
3. A. McNeile and E. Roubtsova. CSP parallel composition of aspect models. *AOM'08: Proceedings of the 2008 AOSD Workshop on Aspect-Oriented Modeling*, pages 13–18, 2008.
4. A. McNeile and E. Roubtsova. Executable Protocol Models as a Requirements Engineering Tool. In *Proceedings of the 41st Annual Simulation Symposium (anss-41 2008)*, pages 95–102, Washington, DC, USA, 2008. IEEE Computer Society.
5. A. McNeile and E. Roubtsova. Composition Semantics for Executable and Evolvable Behavioural Modeling in MDA. *BM-MDA'09: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 1–8, 2009.
6. A. McNeile and N. Simons. http://www.metamaxim.com/.
7. A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107, 2006.
8. A. McNeile and N. Simons. A Typing Scheme for Behavioural Models. *Journal of Object Technology*, 6(10):81–94, November 2007.
9. B. Morin, J. Klein, O. Barais and F-M. Jézéquel. A Generic Weaver for Supporting Product Lines. In *EA '08: Proceedings of the 13th international workshop on Early Aspects*, pages 11–18, New York, NY, USA, 2008. ACM.
10. C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
11. C. Hoare. Why ever CSP? *Electronic Notes in Theoretical Computer Science*, 162:209–215, September 2006.
12. D. Harel and E. Gery. Executable Object Modelling with Statecharts. *IEEE Computer, 30(7)*, pages 31–42, 1997.
13. E. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. *ICSE 2004. Proceedings. 26th International Conference on Software Engineering. IEEE*, pages 158–167.
14. E. Roubtsova and A. McNeile. Abstractions, Composition and Reasoning. In *AOM'09: Proceedings of the 13th workshop on Aspect-Oriented Modeling. Charlottesville, Virginia, USA*, 2009.

15. G. Bracha and W. Cook. Mixin-based inheritance. *Proc. of the ASM conference on Object-Oriented Programming, Systems, Languages, Applications, OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10*, pages 179–183, 1990.
16. G. Zhang, M. Hlzl and A. Knapp. Enhancing UML State Machines with Aspects. In *G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors. Proc. 10th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'07). LNCS 4735*, pages 529–543, 2007.
17. H. Ossher and P. Tarr. Multidimentional Separation of Concerns and the Hyperspace Approach. In *Proc. Architectures and Component Technology: The State-of-the Art in Software Development*, January 2000.
18. J. Ebert and G. Engels. Observable or invocable behaviour - You have to choose. *Technical report. Universitat Koblenz, Koblenz, Germany*, 1994.
19. J. Kienzle, W. Al Abed and J. Klein. Aspect-oriented Multi-view Modeling. *Proceedings of the International Conference on Aspect-Oriented Software Development, AOSD'09, Charlottesville, Virginia, USA*, pages 87–98, 2009.
20. J. Kienzle, N. Guelfi, and S. Mustafiz. Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. *Transactions on Aspect-Oriented Software Development*, 7:1 – 22, 2010.
21. M. Jackson. *System Development.* Prentice Hall, 1983.
22. M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
23. M. Mahoney, A. Bader, T. Elrad and O. Aldawud:. Using Aspects to Abstract and Modularize Statecharts. *In the 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004*, 2004.
24. OMG. Unified Modeling Language, Superstructure, v2.2. *OMG Document formal/09-02-02*, 2009.
25. R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In P. Tarr, L. Bergmans, M. Griss and H. Ossher, editor, *Proceedings of Workshop on Advanced Separation of Concerns, OOPSLA 2000.* Department of Computer Science, University of Twente, The Netherlands, 2000.
26. R. Milner. *A Calculus of Communicating Systems*, volume 92. 1980.
27. R. Reddy, S. Ghosh, R. France and B. Straw. Directives for composing aspect-oriented design class models. *TAOSD LNCS 3880* , pages 75–105, 2006.
28. S. Apel, T. Leich and G. Saake. Mixin-Based Aspect Inheritance. *Technical Report No. 10/2005, University of Magdeburg, Germany*, 2005.
29. S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach.* Addison Wesley, 2005.
30. S. Cook and J. Daniels. *Designing Object Systems. Object-Oriented Modelling with Syntropy.* Prentice Hall, 1994.
31. S. Katz. Aspect Categories and Classes of Temporal Properties. *Transactions on Aspect-Oriented Software Development, LNCS 3880*, pages 106–134, 2006.
32. S. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture.* 2002.
33. S. Shlaer and S. Mellor. *Object Life Cycles - Modeling the World in States.* Yourdon Press/Prentice Hall, 1992.
34. T. Elrad, O. Algawud and A. Baber. Aspect-oriented modelling-Briging the gap Between Design and Implementation. *Proceedings of the First ACM International Conference on Generative PRogramming and Component Engineering GPCE),Pittsburg*, pages 1189–202, 2002.

35. T. Santen and D. Seifert. Executing UML State Machines. *Technical Report 2006-04,Fakultt fr Elektrotechnik und Informatik, Technische Universitt Berlin*, 2006.
36. Z. Drey, C. Faucher, F. Fleurey, V. Mah and D. Vojtisek. Kermeta language. *http://kermeta.org/documents/manual/htm_chunked*, 2009.