Executable Protocol Models as a Requirements Engineering Tool

Ashley McNeile,
Metamaxim Ltd,
48 Brunswick Gardens,
London W84AN UK,
ashley.mcneile@metamaxim.com

Ella Roubtsova,
Open University of The Netherlands,
Postbus 2960, 6401DL Heerlen,
The Netherlands
ella.roubtsova@ou.nl

Abstract

Functional prototypes and simulations are a well recognized and valued tool for building a shared understanding of requirements between users and developers. However, the development of such artifacts does not sit well with traditional modeling techniques, which do not lend themselves to direct execution. Consequently building prototypes and simulations becomes a diversion from the mainstream development process, and sometimes even competes with it.

We propose that the resolution to this conflict lies in promoting the role of executable *behavioral models, so that artifacts supporting behavioral simulation are a by-product of the mainstream modeling process. We discuss why conventional modeling techniques are not suited to this, and we describe an innovative behavioral modeling technique, Protocol Modeling, that is well suited to direct execution.*

Using Protocol Modeling, a behavioral entity (business object or process) is modeled in terms of its event protocol: the conditions under which it accepts or refuses events. Such models capture the behavioral integrity rules at the level of business events; and can be composed using the semantics of Hoare's CSP, allowing concise and incremental representation. Direct execution of the model is achieved using a tool that simulates a normal user interface, so that non-technical stakeholders can review and explore behavior while requirements are being solidified.

1. Introduction

Our domain of interest is the development of business information systems, using object-oriented modeling and development methods. In particular, we are interested in the requirements engineering activities where the aim is to achieve a shared understanding of the required external behavior of a proposed system with the user community. A central issue here, and the subject of this paper, is the choice of a suitable medium to use for describing and demonstrating proposed behavior in support of reaching this shared understanding between the developer and user communities.

1.1. Review Behavior Models

The traditional medium of discourse between developers and users in requirements analysis is the review and discussion of behavioral models, represented as diagrams. This is not, in general, a satisfactory approach as these models are generally intended to inform the detailed design and development process, and not as a means of communication with the user. Users are not normally comfortable with the language and conventions of object-oriented software design at the level at which these models are normally expressed (i.e., software objects and the messages passed between them), nor do they view it as their role to understand models at this level. In particular, without training and practice, the mental translation from a static representation (a diagram) to dynamic behavior (of a system interacting with a business) is hard – just as the mental translation of musical notation into how the music would sound when played, is hard. This is evidenced in surveys of the way UML is used in practice, which show that behavior models are normally aimed at the technical (developer) audience rather than business users [3]. Our experience bears this out, and we do not believe that review of models can ever serve as a basis for achieving a proper shared understanding of proposed system behavior.

1.2. Review of Working Software

Increasingly, best practice in requirements engineering is based on the presentation and exploration of working software: some kind of early prototype or simulation of the final system. The value of providing stakeholders with working (executable) artifacts early in the development process

is well recognized. By presenting a representation of the intended system *in a form that makes sense to users*, simulation/execution offers the potential to make early design ideas accessible to stakeholders *who are not familiar with technical modeling notations*, and thus widen the circle of participants in design 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.

1.3. Focus and Structure of this Paper

The focus of this paper is an an examination the how such early presentation of working software is best achieved. We argue that, of various possible possibilities, *Executable Modeling* provides the best route to early execution in the context of requirements analysis. However, we also argue that mainstream modeling practice (based on the Unified Modeling Language, UML) does not support this well, and that other approaches offer more promise. We illustrate this by describing an executable modeling technique based on *event protocols* which has proved valuable as a basis for early model execution.

The paper is organized as follows:
Section 2 presents a short commentary on current state-of-the-art in early execution using various approaches.
Section 3 describes an executable behavior modeling approach, Protocol Modeling, based on event protocols.
Section 4 describes how execution of Protocol Models is used to support early user feedback.
Section 5 is a conclusion.

2. Commentary on Current Practice

There are various potential means of providing early executability, which we categorize under three headings:

- Executable Modeling
- Functional Prototyping
- Extreme Programming

In this section we look at these three routes in turn and justify our view that all of these, in their current form, have shortcomings.

2.1. Executable Modeling

Here we take the current state-of-the-art in modeling to be the *Unified Modeling Language* (UML) [12], and we examine the degree to which the UML modeling notations support executability. We will not be looking at Use Case Models or Class Models, and motivate these exclusions as follows:

- **Use Case Models** are used to model a system from an external perspective, to capture requirements. A Use Case Model treats a system as a black box, and the interactions with the system are perceived as from outside the system [1]. Our focus, in contrast, is the internal behavior of the system being developed. While simulation/animation techniques make sense for validation of the emerging *internal* design of a system against the *external* required behavior defined in a Use Case Model, it doesn't make logical sense in this context to animate/execute the Use Case Model itself.
- **Class Models** are purely structural in nature and have no behavioral semantics[1].

UML provides a number of notations aimed at behavior modeling which are intended to be used in combination to describe the required behavior of a system. These notations are: Interaction Diagrams, Activity Diagrams and State Machine Diagrams [12]. Our thesis is that **none** of these notations provide a suitable basis for simulation/execution of the kind of business object based models that are the core of modeling business information systems, and we justify this assertion for each notation in turn.

- **Interaction Models** The interaction diagrams in the UML2 notation set are: Sequence Diagram, Communication Diagram, Interaction Overview Diagram, Timing Diagram, and Interaction Table. These focus on the communications between instances, using message passing to express operation invocation and signal sending. These diagrams are scenario based, or "exemplaric" [13], in that an interaction aims to describe a set of possible traces of the system, often aligned to a particular Use Case, but not to describe *all* traces. As the UML2 specification [12] puts it: "The traces that are not included [in an Interaction Model] are not described by this Interaction at all, and we cannot know whether they are valid or invalid". This characteristic of interaction models makes them unsuitable for execution or code generation, except in the weak sense of animation of a scenario (i.e., graphically highlighting the flow of messages between the participants if the interaction).
- **Activity Models** Activity Diagrams describe flow behavior, using Petri Net-like semantics, and are suited to

[1] There is a school of thought according to which behavior can be added to a Class Diagram by adorning the operations (methods) of each class with pre- and post-conditions. We do not hold with this view, which seems to confuse the concepts of "Contract" and "Protocol". See discussion of this in [10]

the description of business workflows. They are complete and can, in principle, be executed: executable processes languages based on the BPEL standard exploit this possibility. Activity Diagrams are not, however, capable of representing the behavior of an object model. As the UML specification puts it: "The focus of activity modeling is the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers [i.e., classes] own those behaviors" [12].

- **State Machine Models** State Machine Diagrams are complete behavior descriptions and can, in principal, provide model based execution of an object model by describing the behavior of each class in the model. UML describes executable semantics for State Charts [12] aimed exactly at this kind of use. The semantic model used for State Machine Execution in UML2 (which was first included in UML at version 1.5) is based on the "Recursive Design" method of Sally Shlaer and Steve Mellor [15], deriving from work by these authors in the real-time/embedded systems domain. The Recursive Design approach addresses the modeling of so-called "active objects" whose instances execute asynchronously (i.e., as if executing on independent threads) resulting in system behavior that is inherently non-deterministic [14]. It is very 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 rather than asynchronous execution of the underlying business objects, and deterministic behavior of business logic is important to ensure repeatability and therefore testability. For this reason, the commercial tools currently available that support execution of UML State Machines (such as those from Telelogic, Kennedy Carter and Mentor Graphics) are unsuitable for use in the business information systems domain and are *explicitly positioned* by their vendors to target the real time/embedded market.

Our conclusion is that, at least in the domain of business system development, there is no UML notation that supports early (model based) execution for behavioral validation by users.

In the absence of suitable executable modeling formalisms within UML, the earliest point at which execution is available is when code is written. If early validation of behavior by execution is to be achieved, therefore, code has to be written early. The options are either to write code as an adjunct to the model (Functional Prototyping), or to write code instead of the model (Extreme Programming). These are discussed in the following paragraphs.

2.2. Functional Prototyping

Producing a Functional Prototype as an adjunct to the model leads to dual representation of the system - in the prototype and in the model. There is no formal way of ensuring that they are aligned and, in practice, they can diverge and compete as being the "true" representation of requirements. Managing dual representations is problematic. Because a prototype is built quickly and amended frequently to reflect review comments, its internal structure is prone to degradation. However the user community, familiar with the prototype and impatient to see a production version delivered, is tempted to lobby for the prototype to be repositioned as the basis for delivery. This perverts the intended development process, as the model (however good it is) becomes sidelined by the prototype (often with a degraded structure) with adverse consequences for the quality of the delivered software.

2.3. Extreme Programming

The alternative to prototyping as an adjunct to the model is to position code as the main representation from the early stages of the project. This approach is seen in eXtreme Programming (XP). While this can deliver executability quickly the problem is that, by eschewing modeling, the level of abstraction is forced down to a low level from the beginning of the project. Using code as the shared medium for understanding of the system in the large is hard with significant team sizes, and in practice there are challenges when scaling this kind of approach to big projects [16].

3. Protocol Modeling

We believe that executable modeling provides the most promising basis for achieving early behavior validation. The challenge is to devise a *complete* (as opposed to "exemplaric") behavior modeling paradigm that works at a level of abstraction that is meaningful to the user: i.e., at the level of the business event interactions between the system and its users. The remainder of this paper describes a modeling paradigm based on *event protocols* which we have been developing and which we believe shows promise in this area. This form of modeling borrows its key ideas from Process Algebras [6, 11], which have, at their core, the idea that a behavioral entity has state and, based on its state, offers its environment the ability to *accept* some events and *refuse* others.

A business information system always has a protocol, as it has to maintain a state (normally represented by its database) that is synchronized with the state of the business it is there to serve. To preserve its own integrity as a meaningful representation of the business reality, a system may

only accept events that allow it to move from one meaningful state to another and so it must refuse events that do not do this. However, the protocols of a system are not normally explicitly modeled but are instead *emergent properties* of the code: sometimes invented by the programmer leading to inconsistent or incoherent behavior. Protocol modeling entails **explicit modeling** of protocols, to yield an executable model that may be used to explore and validate behavior during requirements analysis.

A central theme of our approach is *composition* of protocols to describe and refine behavior. Protocol descriptions can be composed both *within* and *across* objects, using the composition rules of Hoare's CSP [6] (the method of composition is described later in the paper). This compositional approach has a number of benefits:

- modeling can be incremental, as protocols are refined by composition;
- individual descriptions can be kept simple, as many can be composed to model complex protocols;
- derived (or calculated states) may be used, giving much greater expressive power than is provided by conventional state machines;
- integrity of the event handling [5] is guaranteed by the compositional semantics.

The building blocks of a Protocol Model are abstract machines called `Protocol Machines (PM)` [10]. A PM is a machine that has a defined alphabet of `event types` that it understands, and the ability to `accept`, `refuse` or `ignore` any event that is presented to it by its environment.

When a PM accepts an event, it will normally perform some kind of update to its internal state. Refusal of an event means that the machine recognizes the event from its alphabet but is unable to handle the event 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.

This section explains the basic concepts of PMs and how they are composed. To illustrate the basic concepts, we use the machines for a very simple Banking system shown in Figure 1. This model comprises two object-types: Customer (modeled by *Customer Machine*) and Account (modeled by *Account Machines 1 - 4*). Note that the example shown here is very simple for ease of understanding.

3.1. Events

An event (properly an "event instance") is the data representation of an occurrence of interest in the real world business domain. Examples of such real world occurrences are "Customer Fred places an order for 100 widgets to be delivered on 12th August" or "Policy holder Jim makes a claim for £250 against policy number P1234". These occurrences are considered to be atomic and instantaneous in the domain.

An event represents such an occurrence as a set of data attributes. Every event is an instance of an event-type, and 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 events is identical to that used in other event based modeling approaches [7] [2].

The alphabet of the PM *Account Machine 1* (in Figure 1) is the following:

- **Open** with attribute schema (Account Id, Customer Id);
- **Close** with attribute schema (Account Id);
- **Deposit** with attribute schema (Account Id, Amount);
- **Withdraw** with attribute schema (Account Id, Amount).

3.2. Notation

Protocol Machines can be represented in any notation that can support the semantics of an event alphabet and the ability to accept or refuse a presented event that is in its alphabet. Possible notations include: state transition diagrams, Colored Petri Nets, flow languages (such as BPEL) and program code.[2] In this paper we use state transition diagrams. Note, however, that the notational similarity to UML statecharts is superficial: UML statecharts do not have semantics for event refusal, and so do not support CSP composition for which the concept of event refusal is crucial. In our use of state transition diagram notation, which does support event refusal, it is to be given the following interpretation:

- Any event-type that is not in the alphabet of the PM is not shown in the diagram at all;
- The event-types that are accepted are those for which there is a transition from the machine's current state;
- If a machine is presented with an event that is in its alphabet but not allowed in its current state, it refuses the event.

Thus a machine of type *Account Machine 1* in the state *Active* will accept events of type *Deposit*, *Withdraw* or *Close* but will refuse an event of type *Open*.

[2] [9] shows how the semantics of popular notations can be redefined to PM semantics.

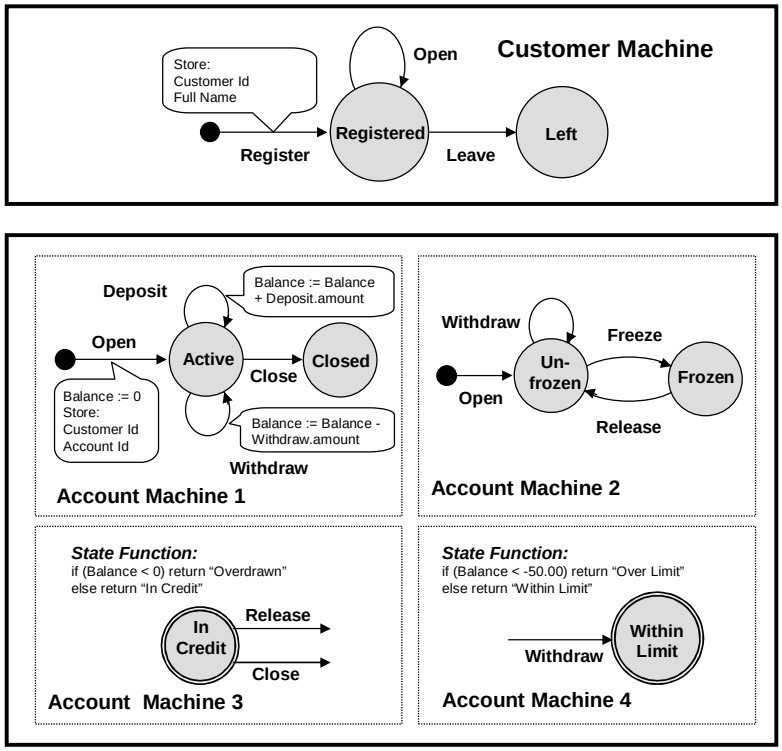

Figure 1. Bank Protocol Model

3.3. Local Storage

A PM has a `local storage` which only it can alter, and only when moving to a new state in response to an event. For example, the local storage of *Account Machine 1* is the set of attributes *Account Id, Customer Id, State, Balance* where the values of *State* can be "New", "Active" or "Closed". Note that "New" means that the machine has not yet received an event: it is still at the "black dot".

When it accepts an event, a PM can update its own local storage. The updates that each machine in the example makes to its local storage are shown in bubbles attached to the transitions; although note that updates to the *State* are implicit in the diagram. A PM may read (but **not** update) the local storage of other composed PMs.

3.4. Composition

Composing two PMs yields another PM. The alphabet of the composed PM is the union of the alphabets of the constituent PMs; and the local storage of the composed PM is the union of the local storages of the constituent PMs. The rules for whether the composed machine accepts, refuses or ignores a presented event are:

- If both constituent machines ignore the event, the composed machine ignores it;
- If either constituent machine refuses the event, the composed machine refuses it;
- Otherwise the composed machine accepts the event.

These rules correspond to the parallel composition operator $(P \parallel Q)$ of Hoare's process algebra, Communicating Sequential Processes [6]. It is crucial to understand that **all machines** in a model are composed, both **within** and **across** objects (see [10] for further discussion of this).

As an example of composition **within** an object, consider *Account Machine 1* and *Account Machine 2*. The composition of these two adds two additional events, *Freeze* and *Release*, to the alphabet of Account beyond those listed above for *Account Machine 1* and defines the rule that a *Withdraw* cannot occur on an Account that is in the state "Frozen". This is because, for a *Withdraw* to be accepted, *Account Machine 1* must be in the state "Active" **and** *Account Machine 2* must be in the state "Unfrozen", otherwise the event is refused by the second rule above.

As an example of composition **across** objects, consider *Customer* and *Account*. This composition requires that, for an *Open* event to be accepted, the *Customer* must be in the state *Registered* and the *Account Machines 1 and 2* must both be in the state *New*. By requiring that all objects that

understand (i.e., do not ignore) an event must accept it, the semantics of composition across objects ensures that event handling is atomic (i.e., that the event is accepted by all objects that are affected by it) as is required by the rules of "ACID" event handling [5].

3.5. Derived States

The definition of a PM does not require that its state is stored, and so it is possible to have the state of a PM returned by a function (called the machine's *State Function*). This is exactly analogous to a derived or calculated attribute, where the attribute value is calculated on-the-fly when it is required.

As an example, *Account Machine 3* has states *In Credit* and *Overdrawn* calculated from the *Balance* in *Account Machine 1*. This machine defines the additional protocol rules that an Account will only accept event-types *Close* and *Release* if the Account is in credit. The composition rules now mean that an Account can only be closed if it is currently both *"Active"* and *"In Credit"*; and only released if it is currently both *"Frozen"* and *"In Credit"*.

In our state transition diagram notation, derived states are shown with a double outlines. Note that the arrows in *Account Machine 3* do not lead to a new state, because the state values of this machine are calculated on-the-fly when needed rather than being updated as the result of a transition.

A machine that has a derived state may also define a constraint on events based on the post-state: the state that results from the event. *Account Machine 4* will only accept a *Withdraw* that results in the Balance of the Account being within a predefined limit.

3.6. Behavioral Types

So far we have taken the rules specified by PMs to be non-discretionary statements about when an event can take place. If an event is refused by a PM, it is rejected and no update or change of state takes place. Sometimes, however, rules are weaker than this, and should not prevent an event but only discourage it. An example is attempting to access a website in Microsoft Internet Explorer v.7 (IE) when the security certificate held by site is not recognized or is invalid. IE alerts the user (with an alert and by coloring the header bar red) but does not prevent access to the site.

Similarly, it may be that violation of a protocol rule should not prevent an event but only discourage it. For instance, it may be that the Bank's policy is not to prevent its Customers from going overdrawn but merely warn them if a Withdraw will do so. A violation of the rule in *Account Machine 4* should not then cause a Withdraw to be rejected, but instead should alert the Customer that by going ahead he/she will breach the account limit. To model this, *Account Machine 4* would be given a different *Behavioral Type* to reflect this weaker protocol effect. A range of such *Behavioral Types* is possible:

- **Essential**: Refused events are rejected as errors (This is the type we have been assuming until now).
- **Allowed**: Events that would be refused are flagged as not desired or in violation of policy (but are not rejected).
- **Desired**: Events that would be accepted are flagged as desired or required (i.e., similar to a workflow rule).

Combining machines with different Behavioral Types so as to prevent, encourage and discourage different events under different circumstances is a way of building behavioral intelligence into the model, so that the system is active in policing the rules and policies of the business. Complete discussion of this topic is beyond the scope of this paper, and more information can be found in [8].

Building behavioral intelligence into objects in this way means that business processes become emergent properties of the combined behavior of the participating objects, rather than being defined as process flow models using languages such as BPEL. This idea (which has been hinted at by others, for instance Fielding with his idea of *Hypermedia as the engine of application state* [4]) involves less redundancy of rule definition than the process flow modeling approach, which normally requires that behavioral rules about an object be repeated in each process that involves the object. Making an object the owner of its own behavioral rules ensures that any change is automatically propagated to all business process contexts in which that object participates. This makes processes easier to modify and hence enhances business agility.

4. Protocol Machines in Requirements Engineering

Our aim is to provide a means of modeling that supports early feedback from the user community. In this section we describe how Protocol modeling is used to do this.

4.1. Model Execution

Models built using the techniques described in this paper are directly executable using a suitable tool. The key features of such a tool are support for automatic composition of PMs, according to the CSP composition rules described above; and provision of a User Interface that allows a user

to see the model: to see what events are possible, and to enter events and see their effect.

The focus of executing the model is to exercise, explore and critique its behavior, and not to design the user interface (in contrast to a "User Interface Prototype"); so a highly standardized user interface suffices. An example of such a user interface (from a PM based modeling tool[3]) is shown in Figure 2. The richness of the behavioral semantics of the modeling language means that this kind of user interface, even though completely standardized, can provide acceptable usability by:

- Displaying, for a selected object, a list of the event types that the object understands.
- Color coding event types to indicate whether they are not possible ("Greying out"), possible but discouraged (e.g., in red), or possible and encouraged (e.g., in green).
- Creating "pick-lists" of suitable target objects for an event, being those objects that understand the event type and are in a state to be able to accept it.
- Creating suitable user-oriented error or warning messages when a protocol rule has been violated.

Note that the interface does not show or require any understanding of the internal structure of the model, in terms of the way objects are modeled as composed machines. This means that stakeholders who are unfamiliar and/or uncomfortable with formal behavior modeling notations can successfully engage in the model validation process.

4.2. Incremental modeling

In practice, it is not possible to collect all the requirements for a model at once, so models must be built up and validated incrementally. Typically, a behavior model is built in phases, each phase scoped by a set of Use Cases (or User Stories). The compositional nature of protocol modeling makes this kind of incremental approach attractive and natural. In the small example we have used, phases might be as shown below (although, in the context of a real system the increments would be bigger than in this illustrative example):

- **Phase 1**: Basic Customer events and data (Machines used: *Customer Machine*, but without the Open event).
- **Phase 2**: Basic Account data and balance maintenance (Machines added: *Account Machine 1*).
- **Phase 3**: Account Freezing (Machines added: *Account Machine 2* and *Account Machine 3*).

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. In the Bank example, questions that might emerge from this process are:

- Should it be possible to close a frozen account?
- Is it possible for an account to be opened by two (or more) Customers (a "Joint Account")?
- What about transferring money directly between Accounts?
- Should it be possible to reassign an Account from one Customer to another?

The act of trying out scenarios stimulates such questions. Moreover, the use of an integrated (rather than scenario based) model makes conflicts and inconsistencies between requirements (which is a common issue with Use Case Models as these are scenario based) impossible to ignore.

4.3. Large Models

In using this technique on real development projects, we have encountered model sizes in the following range:

- Number of Objects: 10 - 50.
- Number of Event types: 50 - 200.

When dealing with models at the upper end of this range it is not realistic to expect one user to understand and validate the entire model. In this case, it is necessary to support multiple "views" of the model, each corresponding to a different user role. Each view is defined in terms of a subset of the object and event types relevant to the responsibilities of the role. For the simple banking model, such roles might be defined as follows:

- **Customer Relationship Manager**: Objects (Customer, Account) Event types: (Register, Leave).
- **Credit Manager**: Objects (Customer, Account) Event types: (Open, Close, Freeze, Release).
- **Teller**: Objects (Account) Event types: (Deposit, Withdraw).

Each view is implemented as a "filter" put on top of the model, specified independently of the model itself. This means that the view definitions are not sensitive to the details of the underlying model, and can be changed without fear of "breaking" the model. With this kind of view facility, the techniques described in this paper can scale to large problems without making it hard for users to understand and review.

[3] This example user interface comes from the ModelScope tool of Metamaxim Ltd, www.metamaxim.com

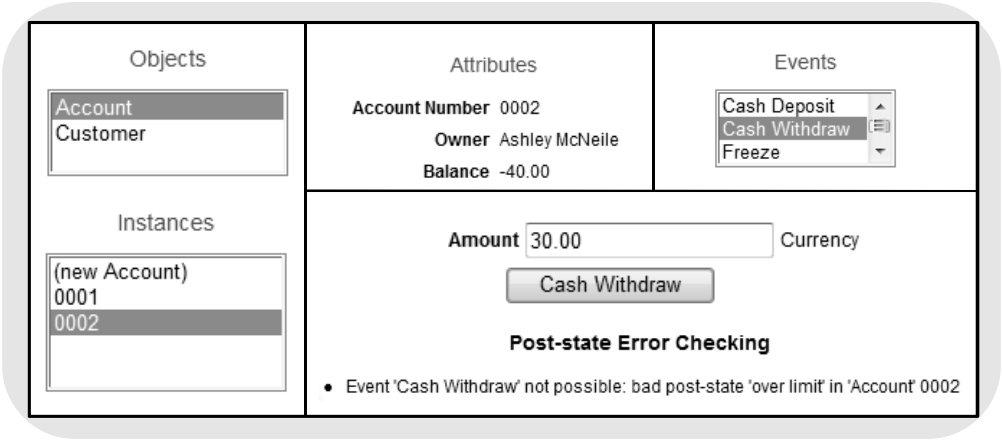

Figure 2. User Interface

5. Conclusion

Requirement engineering is key to successful system development. Lack of precision in requirements will result in difficulties later in development and implementation when it is expensive to make changes. In our view, an investment in behavioral model validation, involving key stakeholders, is a valuable communication tool in ensuring that ambiguity, inconsistency and misunderstanding are identified and eliminated at an early stage. In addition, the ability to extend and experiment with the executable model makes it an important tool in evolutionary system development. We believe that the use of behavior modeling approaches such as that described in this paper makes such early behavioral exploration and validation feasible, and thus can reduce the risk and waste in the system development process.

References

[1] Ambler S. *The Elements of UML(TM) 2.0 Style*. Cambridge University Press, New York, NY, USA, 2005.

[2] Cook S., Daniels J. *Designing Object Systems: Object-oriented Modelling with Syntropy*. Prentice-Hall, 1994.

[3] Dobing B., Parsons J. How UML is Used. *Communications of the ACM*, 49(5):109–113, 2006.

[4] Fielding R. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[5] Gray J. The transaction concept: Virtues and limitations. *Proc. of the 7th International Conference on Very Large Data Bases*, pages 144–154, 1981.

[6] Hoare C. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[7] Jackson M. *System Development*. Prentice Hall, 1983.

[8] McNeile A., N. Simons. A Typing Scheme for Behavioural Models. *Journal of Object Technology*, 6(10):81–94, November 2007.

[9] McNeile A., Roubtsova E. Protocol Modelling Semantics for Embedded Systems. *Proc. of IEEE Second International Symposium on Industrial Embedded Systems, Costa da Caparica, Portugal*, pages 258–265, July 2007.

[10] McNeile A., Simons N. Protocol Modelling. A modelling approach that supports reusable behavioural abstractions. *Software and System Modeling*, 5(1):91–107, 2006.

[11] Milner R. *Communicationg and Mobile Systems - the Pi-Calculus*. Cambridge University Press, 1999.

[12] OMG. UML 2.0 Superstructure Final Adopted Specification. *OMG Document reference ptc/03-08-02*, August 2003.

[13] Rumpe B. Executable Modeling With UML - A Vision or a Nightmare? *Issues & Trends of Information Technology Management in Contemporary Associations*, pages 697–701, 2002.

[14] Santen T., Seifert D. Executing UML State Machines. *Technical Report 2006-04,Fakultät für Elektrotechnik und Informatik, Technische Universität Berlin*, 2006.

[15] Shlaer S., Mellor S. *Object Life Cycles - Modeling the World in States*. Yourdon Press/Prentice Hall, 1992.

[16] Stephens M., Rosenberg D. *Extreme Programming Refactored*. Apress, 2003.