PROGRAMMING IN PROTOCOLS
A Paradigm of Behavioral Programming

Ashley McNeile
Metamaxim Ltd., 48 Brunswick Gardens, London W8 4AN, United Kingdom
ashley.mcneile@metamaxim.com

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

Keywords: Object-orientation Programming Modeling Composition CSP Protocols

Abstract: We present work in the creation of a programming paradigm based on the event protocols of objects. Our claim is this results in a high level executable language that bridges the gap between behavioral models and code for a large class of systems. The language is based on the idea of composing partial behavioral descriptions using process algebraic techniques. We show that the concepts forming the basis of this language shed light on questions relating to the description of behavior in object models, particularly in the areas of reuse, abstraction, and behavioral conformance.

1 BACKGROUND

The earliest techniques for object modeling appeared in the 1970s, before the mainstream appearance of object oriented programming. Two leading approaches were Jackson System Development (Jackson, 1983) and Shlaer-Mellor (Shlaer and Mellor, 1992). Both of these techniques had at their core the idea of *object life-cycle modeling*, in other words describing, in some form, the possible states of an object and how events cause the object to move from one state to another during its life.

When object orientated (OO) programming started to become mainstream in the 1980s (with Smalltalk), object modeling techniques and practice tended to become aligned to the features and capabilities of the emerging OO programming languages. The emphasis in object modeling then moved to classes, inheritance, attributes and relationships, features closely aligned to the capabilities of OO languages, and away from life-cycle modeling, which is not supported in any direct way by any OO language.

The Protocol Modeling paradigm described in this paper has its origins in the JSD, but we have both moved away from the particular notations used in JSD towards a more abstract formulation, and strengthened the semantic basis of the ideas, to the point where direct execution can (and has) been supported. Along the way, the formalisms and ideas have changed so much that any clue that JSD is its provenance has all but disappeared, and the result is a high level behavioral programming language.

2 PURPOSE AND ORGANIZATION

This paper does not intend to provide a full description of the Protocol Modeling language, this has been covered elsewhere (in particular in (McNeile and Simons, 2006)). Rather, we want to discuss those aspects of the approach that are innovative, and represent a departure from conventional thinking or shed light on existing ideas.

The paper is organized as follows:

Section 3 discusses the process algebraic basis of the Protocol Modeling language.

Section 4 describes the semantics of the language.

Section 5 provides a small example based on a Bank.

Section 6 discusses some of the theoretical implications of the ideas.

Section 7 discusses the practical implications of the ideas.

3 PROCESS ALGEBRAIC COMPOSITION

The Process Algebras (CCS, CSP, etc.) all deal with abstract processes (or machines) that offer their environment the ability to accept some events and refuse or ignore others, based on their "state". This ability is the basis for composing processes in various ways. In the abstractions used in process algebras, the possible states of a process are simply enumerated, as they are entailed in the algebraic description of the process. For instance, in the following example (using Hoare's CSP notation):

$$P = (w \rightarrow P) | (x \rightarrow R),\ R = y \rightarrow STOP,\ Q = z \rightarrow P$$

there is an implicit definition of the *state-space* of Q as $\{initial, P, R, stopped\}$. It also specifies the *protocol* of the process: which of the events $\{w, x, y, z\}$ used in the definition of Q it is prepared to engage in (accept) in each state. For instance, when in state P it will accept w or x, but refuse to engage in y or z. These formalisms are geared to formal algebraic behavioral analysis, but not to modeling systems that store significantly complex data.

However, the ability to compose processes requires *only* that they have a well specified protocol behavior, meaning that they offer their environment the ability to *accept, refuse or ignore* any presented event. In particular, two machines that both have such protocol behavior can be put in CSP parallel composition to form a *composite machine* as follows:

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

Based on this, we define a *Protocol Machine* as a machine that has a specified protocol whereby it will accept, refuse or ignore any event that is presented to it; and observe that the above definition means that the composition of two Protocol Machines is another Protocol Machine. We neither prescribe nor proscribe any particular notation for a Protocol Machine: either for its state-space or for the internal mechanisms it uses to decide its protocol behavior. In addition, we allow a Protocol Machine to have arbitrary local storage, in same manner as a class in conventional object oriented programming. A Protocol Machine may update its own local storage, but only when moving to a new state as a result of accepting an event presented to it; and it may read *but not update* the local storage of other machines composed with it.

4 PROTOCOL MODELING

A *Protocol Model* is a model built from Protocol Machines. Such models are event driven and deterministic, and the language we use to build them belongs to the class of languages termed *synchronous reactive* (see (Benveniste and Berry, 2002)). In this section, we outline the main concepts of this language [2]. The description given here is not exhaustive, but sufficient as a basis for the subsequent discussion of the wider implications of the ideas. In particular, we do not define a syntax for the language, such as might be presented to a compiler.

4.1 Events and Machines

The language deals with event driven systems, where an *event* (properly an "event instance") is the data representation of an occurrence in the real world domain that is of interest to the system. 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 attribute schema, this being the set of data attributes that completely define an instance of the event-type. For instance, the attribute schema for a `Transfer` event (to move funds from one account to another) in a banking system might be:

```
Event-type:          "Transfer",
Date-of-transfer:     Date,
Source:              acct-id,
Target:              acct-id,
Amount:              Currency
```

The system moves from one state to another as a result of accepting events presented to it. This approach to modeling events is identical to that used in other event based modeling approaches (Jackson, 1983; Cook and Daniels, 1994).

4.2 Objects and Repertoires

Our aim is to model systems that comprise a population of objects, which are separately instantiated and which each have their own, unique, identity - in other words, an *Object Model*. We achieve this by defining a Protocol Model as a collection of *composed Protocol Machines* (using the CSP composition described above), each with an OID (object identifier) which ties the machine to the object whose partial definition it represents. The OID of a machine is an immutable property of the machine, given to it when

[2] We use the epithet "Modeling Language" as the behavioral semantics resemble those normally associated with a modeling language.

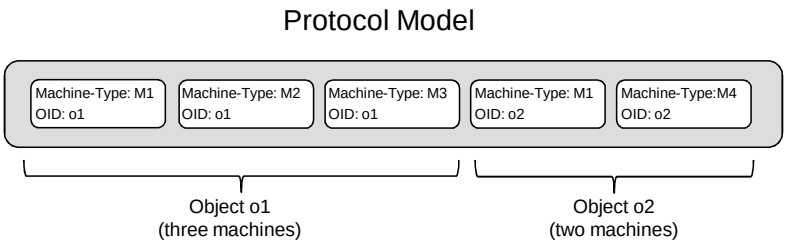

Figure 1: Machines and Objects

it is instantiated. The relationship between objects and machines is depicted in Figure 1 which shows two objects, one (with OID $o1$) comprising three machines and the other (with OID $o2$) comprising two. It is crucial to understand that *all* of the machines in Figure 1 are composed using the parallel composition construct of CSP, both *within* and *across* objects.

As in CSP, every Protocol Machine divides the universe of possible events into those that it understands (can accept or refuse) and those that it does not (and ignores). In CSP, this is done on the basis of the *alphabet* of a process: the set of event symbols for which the behavior of the process is defined. Thus the alphabet, $\alpha(Q)$, of the process Q in the earlier example would be the set $\{w,x,y,z\}$. Protocol Machines have a similar but slightly more complex construct, called the *repertoire* of a machine, which we now describe.

We define the repertoire of a machine[3] m, denoted by $\lambda(m)$, as a set of triples each of the form $[E,o,R]$, with the following meaning:

An event instance e is in the repertoire of a machine $m \Leftrightarrow [E,o,R] \in \lambda(m)$, where:

- E is the event-type of e,
- the schema for E has an attribute with name R [4],
- The value of R in e is o, the OID of m.

Consider machines in a banking model that represent Accounts and handle Transfer events. Suppose that there is one of these machines (m_1) with OID 12345 and one (m_2) with OID 34567, and consider their roles in an event to transfer £250.00 from account 12345 to account 34567, as shown below:

```
Event-type          =  "Transfer",
Date-of-transfer    =  20-Jul-2007,
Source              =  12345,
Target              =  34567,
Amount              =  250.00
```

The machine m_1 will have the entry `[Transfer, 12345, Source]` in its repertoire and will understand, and either accept or refuse but will not ignore, the *Source* side of the Transfer transaction. The machine m_2 will have `[Transfer, 34567, Target]` in its repertoire and will handle the *Target* side of the transaction. Under the rules of CSP composition, only if both machines accept the event will the model as a whole accept it, and in this way the composition enforces *atomicity*, as required by the rules of ACID transaction handling [5].

Account machines in model with OIDs other than 12345 or 34567 will ignore this Transfer. Incorporating the OID in the repertoire entry makes the object structure of the model transparent to the composition scheme.

4.3 Machines Types and Representation

Like events, machines have types and instances, so that every machine instance has a type, and behaves according to its type. As stated earlier there is no standard representation of the behavior of a machine type, but the graphical notation described below has proved both usable and useful in practice.

The notation uses states and transitions and there are two variants of the notation, as shown in Figure 2. In both variants, the diagram represents a successful event acceptance scenario. In both cases the circles represent states of the machine, and the arrow represents the effect of an event. The arrow (transition) is marked with the repertoire entry to which the scenario applies. As we are at the type level here, the OID is omitted from the repertoire entry leading to the form $[E,R]$ shown (the OID is added to create the full

[3] We use "machine" to mean a *machine instance* and use a lower case symbol, m, to stand for a machine instance. When we mean *machine type* we will say so explicitly and use an upper case symbol, M. Similarly with events and event types.

[4] R here stands for Role, as the attribute name normally signifies the role the object is playing in handling an event.

[5] Described,for instance, in (Gray, 1981).

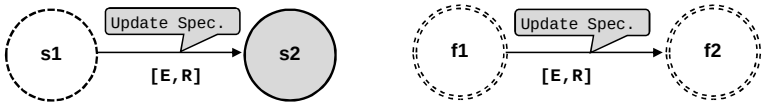

Figure 2: Transitions

form $[E,o,R]$ when a machine instance is created). The solid (shaded) parts of the diagram represent the way the machine updates its own local storage, and the dotted parts represent the tests that must succeed for the event to be accepted, as described below.

In the left hand form, the state of the machine is represented by a distinguished variable (the *state variable*) in the local storage of the machine. This variable has an enumerated type of which $s1$ and $s2$ are two possible values. This diagram has the following semantics:

- The scenario is applicable if the value of the state variable before the event is $s1$.
- The scenario results in a set of updates, specified by `Update Spec`, being applied to the local storage of the machine.
- In addition, the scenario results in the machine's state variable being set to $s2$ after the event, this being the only mechanism whereby the state variable can be changed.

In the right hand form, the circles (this time with a double outline) represent values that are computed by the machine, using a distinguished function called the machine's *state function*. This domain of this function is the local storage of the machine and all composed machines and the function returns an enumerated type, of which $f1$ and $f2$ are two possible values. Again, the diagram represents a successful event acceptance scenario, but with the following semantics:

- The scenario applies if the value returned by the state function *before* the event is $f1$ and the value returned by the state function *after* the event is $f2$.
- The scenario results in a set of updates, specified by `Update Spec`, being applied to the local storage of the machine.

The behavior of a machine type is specified as a *set of success scenarios*, with the following rules:

- The two variants are not mixed within a given machine type. A machine type is either *stored state* in which case it has a single, distinguished, state variable as part of its local state and only uses left hand variant scenarios; or it is *derived state* in which case it has a single, distinguished, state function that returns its state value and only uses right hand variant scenarios.
- The success scenarios of a given machine type are mutually exclusive, so that it is not possible that two (or more) scenarios can succeed for the same event [6].

Finally, the success scenario diagrams for a single machine type can be "stitched together" to form a single graphical state transition diagram that represents the behavior of the machine.

5 EXAMPLE

Figure 3 is an example of a Protocol Model, showing the behavior of the machine types of the model. This model has two object types, `Customer` and `Account`, whose behavior is defined using one and four machine types respectively.

Note the following about this model:

- `Account Machine 2` allows the repertoire entry `[Transfer, Source]` when in the state `Unfrozen` but not in the state `Frozen`. This means that it is not possible to transfer funds out of a frozen Account. However `Account Machine 2` does not have the entry `[Transfer, Target]` in its repertoire, so whether an account is frozen or not has no affect on its ability to act as the target of transferred funds.
- Some transitions are labeled with two repertoire entries, for instance `[Withdraw, From]` and `[Transfer, Source]` in `Account Machine 1`. Using a single transition arrow and specification of updates for both is possible because both are treated identically by this machine. This is discussed further in Section 6.4.
- `Account Machine 3` and `Account Machine 4` (both of which use the right hand form in Figure 2) have no ending and starting states respectively. This means that there is no protocol test on the event(s) in these machines after and before the event respectively.

[6] This is typically arranged by ensuring that, if two success scenarios in the same machine type apply to the same repertoire entry, they have mutually exclusive starting states.

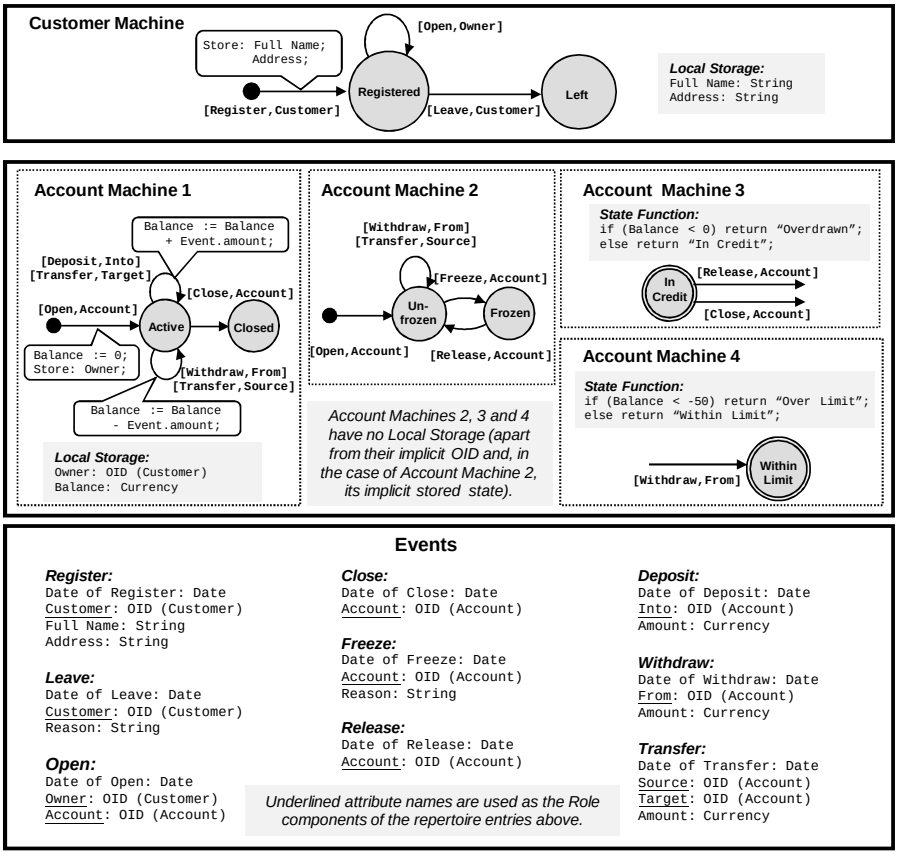

Figure 3: Bank Model

- Two of the event types in the model, `Open` and `Transfer`, require the engagement of two objects: a `Customer` and an `Account`, and two `Accounts` respectively. For these events, the CSP composition rules ensure that the model accepts the event iff both objects involved accept it, as required by ACID event handing.
- Presentation of an event containing an OID that does not exist in the model causes a new object to be instantiated with the new OID. Such an event will only be successful (accepted) if it is also the creation event for the new object: as `Register` is the creation event for `Customer`, and `Open` is for `Account`.

6 THEORETICAL OBSERVATIONS

In this section, we discuss some of the theoretical observations stemming from our work.

6.1 Derived States and Composition

Most modeling formalisms recognize the concept of a *derived attribute*[7]. However, the use of a derived quantity as the state of behavioral machine (as we do in the right hand form of Figure 2) is, as far as we know, unique to our approach. Without this capability, states determining protocol behavior that involve derived quantities have to be expressed in other ways: typically by the use of guards on transitions, or by omitting such rules from the model and handling them in code. It would not be possible, for instance, to define `In Credit` and `Overdrawn` as states of an account (as we have in `Account Machine 3`), or show the protocol rules that depend on these states in diagrammatic form.

Describing complex behavior as a composition of separate machines, some with stored and some with derived states, helps to ensure that descriptions remain as simple as possible, and to avoid the prob-

[7] In UML, a derived attribute is indicated by prefixing its name with "/".

lem of combinatorial state-space explosion that results from representing complex behavior in a single diagram. In addition it is the basis for reuse, as described in the next Section.

6.2 Inheritance, Reuse and Behavioral Conformance

A cornerstone of Protocol Modeling is the idea that the behavior of an object is the composition of partial behavior descriptions. If we add to this the idea that these partial behavior descriptions can be *reused* across different object types, we arrive very naturally at a pure *mixin* approach to behavioral reuse (in the sense described in (Bracha and Cook, 1990)). Thus, in a banking context where it is desired to define a variety of different types of account product, a catalogue of machines such as those shown in Figure 3 can be created and assembled in different combinations to form different account product variants with different behavioral rules. The machine `Account Machine 1`, as it defines the core protocol of an account and the maintenance of the account balance, is likely to be a common mixin to all variants.

The use of CSP as the means of combining the mixin components of an object guarantees that the composite has *Observational Conformance* (a term coined by Ebert and Engels in (Ebert and Engels, 1994b)) with the components contributing to it [8]. As far as we know, Protocol Modeling is the only behavioral modeling approach that guarantees a degree of behavioral conformance by virtue of its semantics, in contrast to other published approaches which require that (sometimes complex) formation rules be followed to achieve behavioral conformance [9]. Moreover, most published approaches which address this matter focus on *substitutability* as the measure of conformance required. In our view substitutability is the wrong measure of behavioral conformance in context of model reuse, as it applies to *subtyping* rather than the creation of *behavioral variants* [10].

It seems that our compositional approach leads very naturally to *mixin* based reuse. However, the mainstream of object modeling, particularly as embodied in UML, has followed OO programming technologies down the *inheritance* path. We think that the natural way that mixins "fall out" of the Protocol Modeling approach casts doubt on whether hierarchical inheritance is the appropriate reuse paradigm for modeling behavioral variants.

[8] As discussed in (McNeile and Simons, 2003).

[9] See, for instance, (Cook and Daniels, 1994), (Ebert and Engels, 1994a), and (Schrefli and Stumptner, 2000).

[10] The same point is made in (Cook et al., 1990).

6.3 Local Reasoning

A concern in compositional approaches to software development is that of *Local Reasoning*: the ability to argue about the behavior of the whole based on examination of only a part. Without the ability to reason locally, all parts of the design have to be viewed in order to reach any conclusion about behavior, making the retention of intellectual control hard in all but the simplest of problems. This issue has been discussed particularly in the context of Aspect Orientation, for instance by (Dantas and Walker, 2006).

We argue that the use of CSP composition enables local reasoning at a behavior level, on the following grounds (this argument is sketched only). Suppose that:

- m is the composition of two machines m_1 of type `M1` and m_2 of type `M2`.
- $traces(m) \upharpoonright_{\lambda(m_1)}$ is $traces(m)$ restricted to the repertoire of m_1.

where $traces(m)$ is the set of sequences of *event instances* that m can accept. Then we have, as a property of CSP composition, that:

$$traces(m) \upharpoonright_{\lambda(m_1)} \subseteq traces(m_1)$$

Using this we can determine from examination of `M1` *alone* that an event sequence that purports to be a trace of m is not actually so, because it fails to meet the above condition. This ability to argue about m from knowledge of `M1` alone constitutes a form of behavioral Local Reasoning.

6.4 Abstraction

Protocol Modeling enables abstraction in models, in the sense that one machine can contain constructs that represent a set of similar constructs in another machine, with the differences that distinguish the individuals of the set abstracted away. In a modeling approach, the ability to abstract is essential to achieving economy and simplicity of expression.

The two forms of abstraction supported are by *event* and *state*.

By Event: In the Banking Model shown in Figure 3, the repertoire entries `[Withdraw, From]` and `[Transfer, Source]` appear in both `Account Machine 1` and `Account Machine 2` attached to the same transition. This means that, in both of these machines, the two repertoire entries are treated identically: including the updates performed to the local storage of the machines. This is equivalent to saying that the difference between these repertoire entries is immaterial for

these transitions, and we can use a single name `Debit` defined as follows:
`Debit = {[Withdraw,From], [Transfer,Source]}`
instead. Note that the `Debit` abstraction could not be used in `Account Machine 4`, as this machine does not constrain Transfers.

By State: The ability to abstract by state stems from use of derived states, as described earlier in Section 4.3. For instance, the state `In Credit` in `Account Machine 3` can be viewed as an abstraction of all non-negative values of the local storage attribute `Balance` in `Account Machine 1`.

The use of these abstraction mechanisms has no affect on the semantics of machine composition, and is therefore orthogonal to the ability to perform local reasoning about behavior.

6.5 Behavioral Types

So far we have taken the protocol rules specified in a Protocol Model to be non-discretionary statements about when an event can take place. If an event is refused, no update or change of state takes place. In effect, the event is rejected as an error. Other classes of behavioral rules are weaker than this, and should not prevent an event but only provide guidance on whether it should happen or not. 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.

In the model shown in Figure 3, it may be that the Bank's policy is not to *prevent* its Customers from going over their account overdraft limit but merely *warn* them if a withdraw does so. A violation of the rule in `Account Machine 4` should not then cause a `Withdraw` event to be rejected, but instead should cause an alert to the Customer that the event breaches the account limit. To model this, we give a different *Behavioral Type* to `Account Machine 4` to reflect this weaker protocol effect. A range of such weaker Behavioral Types is possible, two being:

Allowed: Refused events are flagged as not desired or in violation of policy (but are not prevented).

Desired: Accepted events 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 required in the domain. Complete discussion of this topic is beyond the scope of this paper, and more information can be found in (McNeile and Simons, 2007).

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 hinted at by others, for instance Fielding with his idea of *Hypermedia as the engine of application state* (Fielding, 2000)) 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.

7 PRACTICAL OBSERVATIONS

The aim of a language such as the one described here must be to close the gap between the way software is described in the analysis and design stages of a project, and the way it is described in executable form. The more this gap can be closed, the better the software development process can be engineered to avoid the familiar problems associated with human interpretation and translation of one description into another. Ideally, in our view, the software development should involve executable artifacts *from the beginning*, and the process should be based around incremental review, testing and refinement (and, perhaps, automated transformation) of such artifacts. The early focus of the process is on the behavioral requirements, the later focus on the implementation environment (target platform) and such issues as performance.

The language described here has proved valuable in the early stages of development, where the focus is on behavioral requirements. The richness of the behavioral semantics of the language makes it possible to provide a generic user interface which interprets the model to:

- Display, for a selected object, what events the object understands.
- Color code events to indicate whether they are not possible ("Greying out"), possible but discouraged, or possible and encouraged.
- Create "pick-lists" of suitable target objects for an event, being those objects that understand and can

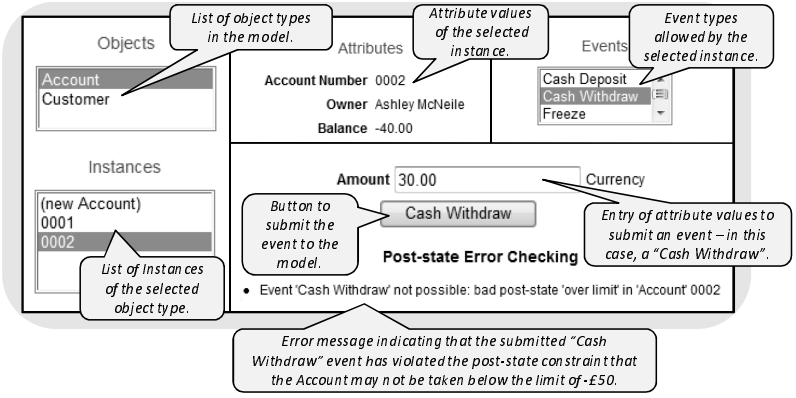

Figure 4: Generic User Interface

accept it.

- Create suitable user-oriented error or warning messages when a protocol rule has been violated.

An illustrative example of such a generic user interface is shown in Figure 4. The important point is that a generic user interface requires *no more information than is present in the model (for instance Figure 3) and the associated instance data*, so it is not necessary to model or program it. This means that when the model is changed the user interface changes according, and this makes it is easy to support a rapid cycle of *develop, review and revise* in the context of an iterative approach to model construction and validation. Because the user 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) stakeholders who are unfamiliar and/or uncomfortable with formal behavior modeling notations can successfully engage in, and contribute to, the model validation process.

REFERENCES

Benveniste, A. and Berry, G. (2002). *The synchronous approach to reactive and real-time systems*. Kluwer Academic Publishers.

Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *ASM conference on Object-Oriented Programming, Systems, Langiuages, Applications*, pages 179–183.

Cook, S. and Daniels, J. (1994). *Designing Object Systems: Object-oriented Modelling with Syntropy*. Prentice-Hall.

Cook, W., Hill, W., and P.Canning (1990). Inheritance is not subtyping. In *17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135. ACM.

Dantas, D. and Walker, D. (2006). Harmless advice. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 383–396. ACM.

Ebert, J. and Engels, G. (1994a). Dynamic models and behavioural views. *LNCS 858*.

Ebert, J. and Engels, G. (1994b). Observable or invocable behaviour - you have to choose. *Technical report 94-38. Department of Computer Science, Leiden University*.

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

Gray, J. (1981). The transaction concept: Virtues and limitations. In *7th International Conference on Very Large Data Bases*, pages 144–154.

Jackson, M. (1983). *System Development*. Prentice Hall.

McNeile, A. and Simons, N. (2003). State machines as mixins. *Journal of Object Technology*, 2(6):85–101.

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

McNeile, A. and Simons, N. (2007). A typing scheme for behavioural models. *Journal of Object Technology*, 6(10):81–94.

Schrefli, M. and Stumptner, M. (2000). On the design of behavior consistent specialization of object life cycles in obd and uml. In *Advances in Object-Oriented Data Modelling*, pages 65–104. MIT Press.

Shlaer, S. and Mellor, S. (1992). *Object Life Cycles - Modeling the World in States*. Yourdon Press/Prentice Hall.