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 recognised and valued tool for building a shared understanding of requirements between users and developers. However, the development of such artefacts does not sit well with traditional modelling processes which tend to focus on structure rather than behaviour. Consequently building prototypes and simulations is a diversion from the mainstream development process, sometimes even competing with it.

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

Using Protocol Modelling, a behavioural entity (business object or process) is modelled in terms of its event protocol: the conditions under which it accepts or refuses events. Such models capture the behavioural 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 behaviour while requirements are being solidified.

1 Introduction

A key goal of requirements engineering is to capture the required external behaviour of a system in a complete and precise form to provide a proper basis for development. We take it as given that feedback from the user community on the behaviour of a system under development is essential in achieving this. However, in current practice, the purpose of behavioural models of the internals of a system under design is primarily to inform the detailed design and development process, and not as a means of communication with the user. This is evidenced in surveys of the way UML is used in practice [1]. We believe that this is because users are not normally comfortable with the language and conventions of object oriented software design at the behavioural (i.e., message passing) level, and
that users do not normally view it as their role to understand the models at this level. Simulation/Execution, if it presents the model in a form that makes sense to users, is a means to make behavioural models more accessible and understandable to stakeholders who are not familiar with modelling notations, and thus widen the circle of participants in the model 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. Our thesis is that current modelling techniques provide poor support for this process, and we propose an alternative to behaviour modelling, based on the concept of event protocols, that supports early execution.

Our domain of interest is the development of business information systems using current object-oriented modelling and development methods, and our requirement is therefore to be able to model the behaviour of a population of business objects, and the business events and activities that change their state and data. We start with a short commentary on current mainstream, UML based, behaviour modelling techniques to explain why we do not think these can support this kind of model execution, and so set the stage for the ideas we want to present.

In this commentary we will not be looking at Use Case Models or Class Models, and we motivate these exclusions as follows:

- **Use Case Models** are used to model a system form 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 [?]. Our focus, in contrast, is the internal behaviour of the system being developed. While simulation/animation techniques allow validation of the emerging design of the internals of a system against the requirements defined in a Use Case Model, it doesn’t make logical sense to animate/execute the Use Case Model itself, as you cannot validate whether an artefact meets requirements specified by the same artefact.

- **Class Models** are purely structural in nature and have no behavioural semantics [footnote on OCL?]?

**Modelling the Behaviour of the System** UML provides a number of notations aimed at behaviour modelling which are intended to be used in combination to describe the required behaviour of a system. These notations are: Interaction Diagrams, Activity Diagrams and State Machine Diagrams [8]. 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 modelling business information systems, and we justify this assertion for each notation in turn.

- **Interaction Models** UML2 supports a variety of interaction based diagrams (Sequence Diagram, Communication Diagram, Interaction Overview Diagram, Timing Diagram, Interaction Table) that focus on the communications between instances, using message passing to express operation invocation and signal sending. These notations are scenario based, or “exemplaric” [9], in that an interaction aims to describe a set of possible traces of the system, often aligned to the a particular Use Case, but not to describe all traces. As the UML specification [8] 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 behaviour, using are Petri Net like semantics, and are suited to the description of business workflows. They are complete and can, in principle, be executed – executable processes languages based on the BPEL standard exploit the direct executability of such flow languages. Activity Diagrams are not, however, suitable for representation the the behaviour 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” [8]. In other words, activity (flow) based models is not intended to be tied to an object model.

- **State Machine Models** State Machine Diagrams are complete behaviour descriptions and can, in principal, provide model based execution of an object model by describing the behaviour of each class of object in the model. UML describes executable semantics for State Charts [8] aimed exactly at this kind of use. The semantic model used for State Machine Execution, imported into UML at version 1.5, is based on the ”Recursive Design” method of [11] which derived from work in the real-time/embedded systems domain. The Recursive Design approach addresses the modelling of so-called ”active objects” whose instances execute asynchronously (as if on independent threads), resulting in system behaviour that is inherently non-deterministic [10]. It is very hard to reconcile this semantic basis with the characteristics of the business information systems domain, where behavioural issues are related to transactional integrity and business rules rather than asynchronous execution of the underlying business objects. For thus 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.

At least in the domain of business system development, then, there is no behavioural UML notation that offers the possibility of early (model based) execution for behavioural validation by users.

**Early Resort to Code** In the absence of suitable executable modelling formalisms, the earliest point at which execution is available is when code is produced. If early validation of behaviour by execution is to be achieved, therefore, code has to be produced early. The options are either to produce code as an adjunct to the model, or to produce code instead of the model. Both have drawbacks, as described below.

Producing code (in the form of 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, with adverse consequences for the quality of the delivered product.

The alternative is to focus on code as the central representation from the early stages of the project. This approach is seen in its most extreme form in eXtreme Programming (XP). The issue here is that, by eschewing modelling, 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 there are challenges when scaling this kind of approach to big projects [12].

**Protocols as the basis for Behaviour.** The challenge in realising early execution for behaviour validation can be rephrased as the challenge of finding complete (as opposed to "exemplaric") behaviour models that work 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. Our proposal is that modelling event protocols, namely the ability of the system to accept or refuse business events based on its state, provides a good basis for doing this. This form of modelling borrows its key ideas from Process Algebras [3],[7] which have, at their core, the idea that a behavioural entity has state and, based on its state, offers its environment the ability to accept 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 others. However, the protocols of a system are not normally explicitly modelled but are instead emergent properties of the code. Protocol Modelling entails explicit modelling of protocols, to yield an executable model that may be used to explore and validate behaviour.

A central theme of the approach we describe is that of composition. We show that protocol descriptions can be composed both within and across objects, using the composition rules of CSP. A compositional approach yields a number of beneficial features:

- modelling 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 [2] is guaranteed by the compositional semantics.

The paper is organized as follows. Section 2 describes the Protocol Modelling approach. Section 3 describes how Protocol Modeling is used to support early user feedback. Section 4 is a conclusion.

### 2: Protocol Modelling Approach and CSP Composition

We propose that behavioural models by built by composing abstractions called Protocol Machines (PM) [5]. A PM is a conceptual 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 modelling purposes.

This section explains the basic concept of PM and how PMs 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 (modelled by Customer Machine) and Account (modelled by Account Machines 1 - 4). Note that the example shown here is very simple for ease of understanding.

Figure 1. Bank Protocol Model

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 modelling events is identical to that used in other event
based modelling approaches [4].

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).

**Notation** Protocol Machines can be represented in any notation that can support the semantics. This includes state transition diagrams, Coloured Petri Nets, flow languages (such as BPEL) or program code. In this paper we use state transition diagrams, with 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*.

**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.

**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 [3]. It is crucial to understand that all machines in a model are composed, both within and across objects (see [5] 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 as required for ACID event handling rules [2].

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 the 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.

Behavioural 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 recognised or is invalid. IE alerts the user to the issue 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 Behavioural Type to reflect this weaker protocol effect. A range of such Behavioural 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 Behavioural Types so as to prevent, encourage and discourage different events under different circumstances is a way of building behavioural 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 [6].

### 3: Protocol Machines in Requirements Engineering

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

**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 behaviour, and not to design the user interface correct (in contrast to a "User Interface Prototype"); so a highly standardised user interface suffices. An example of such a user interface (from a PM based modelling tool) is shown in Figure 2.

![Figure 2. User Interface](image-url)
Note that the interface that does not show or require any understanding of the internal structure of the model, in terms of the way objects are modelled as composed machines. This means that stakeholders who are unfamiliar and/or uncomfortable with formal behaviour modelling notations can successfully engage in the model validation process.

**Incremental Modelling**  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 behaviour model is built in phases, each phase scoped by a set of Use Cases (or User Stories). The compositional nature of protocol modelling makes this kind of incremental approach attractive and natural. In the small example we have used, phases might be:

- **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, as noted in the Introduction, is a common issue with Use Case Models) impossible to ignore.

**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.

4: 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 behavioural 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.

References


