modelscope™

Version 2.0

Modellers' Guide

CONTENTS

1 Introduction

1.1 ModelScope

ModelScope is a tool for exploring and defining the requirements and logical architecture of transactional business applications. Using ModelScope it is possible to create testable application models from simple logical descriptions of the objects in the application and how they behave.

The heart of ModelScope is a state transition diagram interpreter which understands how events change the states of objects, what events an object can accept based on its state, and how events create and destroy relationships between objects. To provide a testable application model, ModelScope also provides a default user interface that allows events to be entered and processed, and a database that provides persistent storage of the created object instances.

ModelScope can be used to analyse a number of different kinds of application systems issues, for instance:

- To explore and define the logical scope and design of a new application or redesign of an existing application.
- To explore and define the behavioural requirements that a package must meet by customisation or parameterisation.
- To explore possible reconfiguration or wrapping of existing applications, e.g. to remove duplication of data or functionality, or to decouple workflow from transactional processing.
- To explore and define the requirements for the systems support required for new or redesigned business processes.
- To communicate aspects of application business purpose and logical design to a wider community.

The behaviour definitions used by ModelScope to create an executable and testable model are stored as metadata. This Guide describes the modelling concepts used by ModelScope and the syntax of the language used to make models executable.

2 How to Use this Guide

2.1 Purpose

This Guide is intended to provide an overview of ModelScope modelling, and detailed information about the format and syntax of the ModelScope modelling language. It is intended to be used as a reference guide when constructing models to execute using ModelScope.

It is not intended to be a tutorial in modelling in general or ModelScope modelling in particular. If you are not familiar with ModelScope modelling, please visit our website (www.metamaxim.com) to find out how you can learn about it.

2.2 Organisation of this Guide

This Guide is organised as follows:

- Section 3 describes the fundamental concepts used in ModelScope.
- Section 4 works through an example to illustrate and explain how a model is represented.
- Section 5 describes general conventions used in ModelScope.

These three sections are intended to be read from beginning to end to obtain background understanding.

- Sections 6 through 11 are reference material for use when constructing and executing models.

2.3 Bank Examples

Example models, based around a simple Banking application, are provided to help understand ModelScope modelling. These are to be used with Section 4, Worked Example.

The example models build progressively, starting with a simple model that only uses basic features and ending with a model that illustrates most of the range of ModelScope's capabilities.

The example models have been constructed to match the topics of each part of Section 4, as follows:

- Bank1 covers Section 4.1.
- Bank2 covers Section 4.2.
- Bank3 covers Section 4.3.
- Bank4 covers Section 4.4.
- Bank5 covers Section 4.5.

Please review these models and run them as you read Section 4 of this guide.

3 Fundamentals

3.1 Behaviour Modelling

ModelScope supports the capture and agreement of the behavioural requirements of an application as part of the requirements gathering stage of a systems project. The technique is applicable to transactional business systems, whether front office or back office, distributed or centralised. The technique is independent of the technology and platform that will be used to implement the final application.

Traditionally, models have been captured as text and diagrams, often using CASE (Computer Aided Systems Engineering) tools. The method of sharing, reviewing and agreeing models has been by document review and walk-throughs. ModelScope allows models to be executed and this enables those developing the model to be sure that it is complete and coherent, and those reviewing the model to understand, test and interact with it.

Use of ModelScope does not assume or constrain the final architecture, implementation approach or technology platform used to build and deploy the application. After ModelScope has been used to develop, test, demonstrate and agree a model, the final system can be produced by a number of means including:

- Custom system development.
- Development using components or frameworks.
- Transfer of the model to another tool for code generation.
- Application package and/or workflow engine customisation or parameterisation.
- Modification or enhancement of an existing application.

Or some combination of these.

3.2 Objects, Events and States

Models are built from three main constructs:

- Objects — These represent the objects with which the application is concerned. An object is defined in terms of its states, behaviour (how it moves from one state to another) and data. In a simple Banking System, the objects might be Customer and Account.

- Events — These represent things that happen in the business or domain. Events are instantaneous and result in changes to the states and data of the objects. In a simple Banking System, key events might be register (a new Customer registers with the Bank) and open (a Customer opens an Account).

- States — Events cause objects to move from one state to another. Between events, states and data do not change. In the Banking System, the states of Account might be active and closed, and of Customer might be registered and left.

This colour coding for Objects, Events and States is used in this guide to help explain the concepts and examples.

3.3 State Transition Diagrams

State transition diagrams are used to show how objects move from state to state as a result of events. For the Banking System, the state transition diagrams are shown in Figure 1.

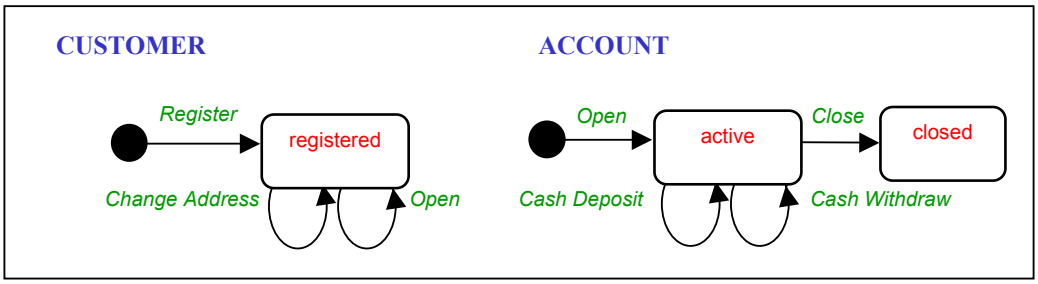

Figure 1

These diagrams say that:

- A Customer may only open an Account once registered.
- A Customer may change address any number of times while registered.
- Any number of Accounts may be opened while the Customer is registered.
- Once an Account is active, any number of cash deposits and cash withdraws may be made until the Account is closed.

4 Worked Example

4.1 Bank1

Please refer to the example model Bank1 for illustration of all the ideas and syntax introduced in this section.

4.1.1 ModelScope

ModelScope uses a modelling language to represent a model. As shown in Figure 2, a model is defined by creating:

- A *Model File* containing ModelScope Metadata.
- *Callback Files* containing Java code.

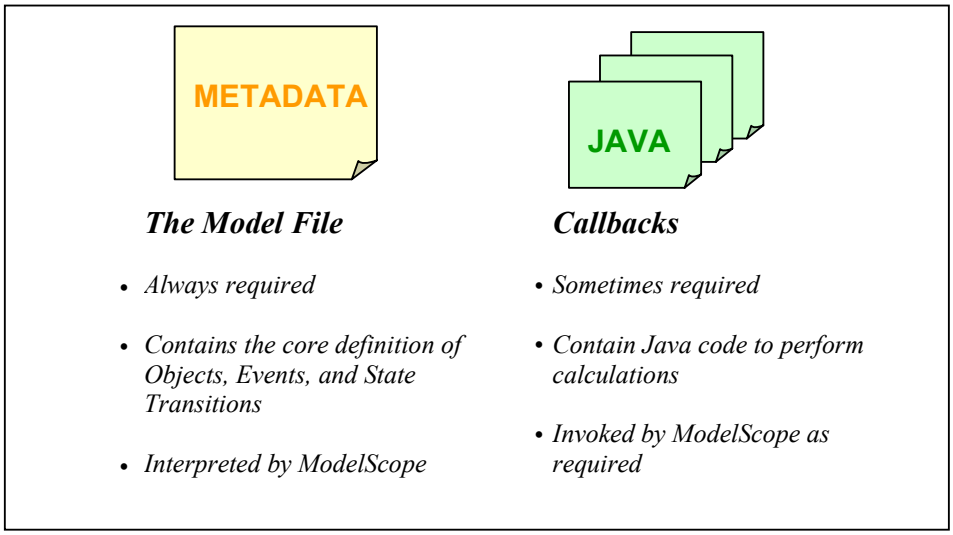

Figure 2

The main definition of the model is in the Model File. This is where objects, states, events and attributes are defined. The Callback Files supplement the Model File by defining how calculations are performed. Callbacks are not always needed – the simplest models do not need them at all.

In this Section, a simple Bank model is developed. As each part of the model is defined the Metadata (colour coded yellow) and Callback code (colour coded green) are illustrated.

4.1.2 Objects

The metadata entries needed to define an Object are:

- The NAME by which an object instance will be identified in the User Interface.
- Its ATTRIBUTES (a list of data items that the object stores and maintains).
- Its STATES (a list of the possible states of the object).
- A state transition diagram, represented as TRANSITIONS.

The metadata (contained in the Model File) for Customer is shown in Figure 3.

```
OBJECT Customer
      NAME Full Name
      ATTRIBUTES Full Name: String, Address: String
      STATES registered
      TRANSITIONS @new*Register=registered,
            registered*Open=registered,
            registered*Change Address=registered
```

Figure 3

The NAME entry says that an instance of Customer will be labelled by the attribute Full Name when a list of Customers is presented by ModelScope at the User Interface.

The TRANSITIONS represent the state transition diagram for Customer in Figure 1. Each transition has the form:

```
starting state * event  = ending state
```

and corresponds to one arrow on the state transition diagram.

The `@new` state in the first transition corresponds to the black dot on the diagram – i.e. the starting state of the transition that creates a new instance. `@new` does not need to be included in the STATES entry. (Nor do other special states that start with "`@`" that will be introduced later).

Every other state used in a transition as either a starting or an ending state must be declared in the STATES entry.

4.1.3 Object Attributes

In addition to its state, each object has a set of data attributes that store information about it. The attributes are defined in the ATTRIBUTES entry in Figure 3.

Each attribute is given a Type, in a similar way to most programming languages. ModelScope supports a range of standard types:

- Boolean.
- Currency.
- Date.
- Integer.
- String.

More details on these types is given in Section 7.

Attributes with these types are called *Value Attributes*. All the attributes for Customer in Figure 3 are value attributes.

Some attributes are pointers to other objects, Foreign Keys in database jargon. In this case, the type given to the attribute in the metadata is the name of the referenced object. These are called *Reference Attributes*. Figure 4 shows metadata for Account. The Owner attribute, shown in bold, is a reference attribute (pointer) to the Customer who owns the Account.

```
OBJECT Account
      NAME Account Number
      ATTRIBUTES Account Number: String, Owner: Customer, Balance: Currency
      STATES …
      TRANSITIONS …
```

Figure 4

The STATES and TRANSITIONS for Account will be discussed shortly.

4.1.4 Events

The second key element for defining models in the ModelScope are *Events*. The metadata for the events in the Bank example are shown in Figure 5.

```
EVENT Register
      ATTRIBUTES Customer: Customer, Full Name: String, Address: String

EVENT Open
      ATTRIBUTES Account: Account, Owner: Customer, Account Number: String

EVENT Change Address
      ATTRIBUTES Customer: Customer, Address: String

EVENT Cash Deposit
      ATTRIBUTES Target: Account, Amount: Currency

EVENT Cash Withdraw
      ATTRIBUTES Source: Account, Amount: Currency

EVENT Close
      ATTRIBUTES Account: Account
```

Figure 5

Each event has an ATTRIBUTES entry. Each attribute is given a type using the same set of types that has already been described for object attributes.

Events can also have both value attributes and reference attributes. Reference Attributes in events can be thought of as holders for the addressees of the event. Like emails, which can have a number of recipients, an event can be sent to a number of objects. This is discussed further below.

4.1.5 Event Transition Correspondence

The *Event Vocabulary* of an object is the set of events that appear in the state transition diagram. The Event Vocabularies of Customer and Account can be listed from the diagrams in Figure 1, and are shown in Table 1.

Table 1

OBJECT	EVENT VOCABULARY
Customer	Register, **Open**, Change Address
Account	**Open**, Cash Deposit, Cash Withdraw, Close

An event can appear in the vocabulary of more than one object, as Open does in this example. An event that is in the vocabulary of multiple objects is called a *Shared Event*.

Every event must be addressed (using reference attributes) to all the objects in whose Event Vocabulary it appears. Table 2 shows the objects addressed by the events in Figure 5.

Table 2

EVENT	OBJECT(S) ADDRESSED	REFERENCE ATTRIBUTE	ATTRIBUTE TYPE
Register	The Customer being registered.	Customer	Customer
Open	The Account being opened. The Customer who owns the Account.	Account Owner	Account Customer
Change Address	The Customer whose address has changed.	Customer	Customer
Cash Deposit	The Account into which funds are being deposited.	Target	Account
Cash Withdraw	The Account from which funds are being withdrawn.	Source	Account
Close	The Account being closed.	Account	Account

The first two events (Register and Open) in Table 2 are creation events (for Customer and Account respectively). Creation events are addressed to the objects they create – **even though these objects do not yet exist.**

The set of events that have Customer in the last column of Table 2 corresponds exactly to the Event Vocabulary of Customer (row 1 of Table 1), and the set of events that have Account in the last column corresponds exactly to the Event Vocabulary of Account (row 2 of Table 1). Models must always be constructed to have this correspondence.

Suppose, for instance, that the Open event had been defined as shown in Figure 6.

```
EVENT Open
      ATTRIBUTES Account: Account, Account Number: String
```

Figure 6

This does not correspond to the state transition diagram for Customer. Customer has a transition for Open, but there is no attribute in Figure 6 to cause the Open event to be sent to a Customer object.

When it loads a model, ModelScope checks that this correspondence is observed and gives an error if it is not.

4.1.6 Event Processing Cycle

ModelScope follows a cycle in processing an event, as described in Table 3.

Table 3

PROCESSING STAGE	DESCRIPTION	AUTOMATIC PROCESSING
Pre-presentation	After the user has chosen an object and an event at the user interface, but before the event attribute entry windows are displayed.	ModelScope populates the event attributes from the selected object by **name co-incidence**.
Presentation	On completion of the Pre-presentation Stage, the event attributes are presented to the user at the user interface. The user can enter values, altering or over-writing those loaded automatically at the pre-presentation stage.	N/A
Post-presentation	After the user has entered values for event attributes and clicked the button to submit the event.	ModelScope checks that the content of each attribute conforms to the validation rules given in Section 7. If any check fails, a message is presented at the user interface, and no further processing of the event takes place.
Update	After all the checks in the Post-presentation Stage have been performed, and only if they all pass.	ModelScope presents the event to each object addressed by the event. For each object it: Populates the object attributes from the event attributes by **name co-incidence**. Updates the state according to the transition definition (i.e. the object state is changed from the starting to the ending state of the transition).

The Event Processing Cycle shown in Table 3 has been simplified. A full version of the event processing cycle is given in Section 8.

4.1.7 Name Co-incidence Data Transfer

As highlighted in Table 3, ModelScope performs automatic transfer of attribute values by name-coincidence:

- At Pre-presentation stage, the event is populated with object attributes prior to display at the user interface. The user may then alter or over-write these values before submitting the event.
- At Update stage, the event attributes are transferred to the object attributes.

Figure 7 shows how this works for a Change Address event for Customer.

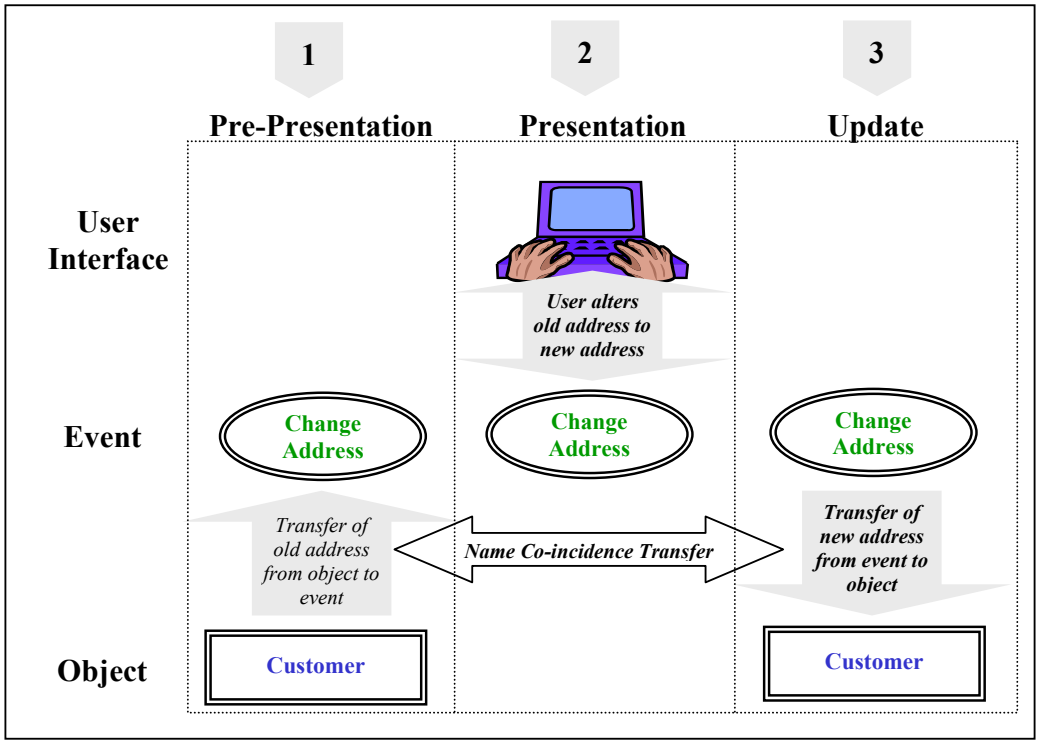

Figure 7

Automatic transfer takes place only if both the Attribute name and the Attribute type match.

Reference attributes are transferred as well value attributes. This means that, for instance, when a new Account is opened the owner attribute from the open event, which references the Customer who owns the Account, is transferred to the owner attribute in the Account object. This creates the Foreign Key pointer from an Account to its owning Customer.

Name co-incidence transfer does not work if a calculation is needed in the update. For instance, the update of the balance attribute in Account for a deposit or withdraw event requires that the amount from the event is added to or subtracted from the balance. In this case a callback is needed, as described in the following section.

4.1.8 Event Processing Callbacks

If updating requires more than simple value transfer name co-incidence updating is not sufficient. In this case a callback is needed.

The metadata for Account is shown in Figure 8.

```
OBJECT Account
      NAME Account Number
      ATTRIBUTES Account Number: String, Owner: Customer, Balance: Currency
      STATES open, closed
      TRANSITIONS @new*Open=active,
            active*Close=closed,
            active*!Cash Deposit=active,
            active*!Cash Withdraw=active
```

Figure 8

The "!" in front on the Cash Deposit and Cash Withdraw events in the Transitions indicate that there are

callbacks associated with these transitions. The callbacks are shown in Figure 9.

```
package Bank1;

import com.metamaxim.modelscope.callbacks.*;

public class Account extends Behaviour {

        public void processCashDeposit(Event event, String subscript) {
                int newBalance = this.getCurrency("Balance") +
                        event.getCurrency("Amount");
                this.setCurrency("Balance", newBalance);
        }

        public void processCashWithdraw(Event event, String subscript) {
                int newBalance = this.getCurrency("Balance") -
                        event.getCurrency("Amount");
                this.setCurrency("Balance", newBalance);
        }

}
```

Figure 9

Note the following:

- The class containing the callback functions is named after the object, in this case: Account.
- The callbacks are coded as two member functions, named "`process`" followed by the name of the event (spaces are removed from the event name).
- "`this`" is the object (i.e. the Account) calling the callback. The "`event`" parameter is the event (Cash Deposit or Cash Withdraw) being processed.
- Special "`getCurrency`" and "`setCurrency`" statements are used to access the values from the Event and Behaviour attributes. The callback language is described in detail in Section 10.5.

ModelScope supports a variety of different kinds of callback. The ones described here are called *Event Processing Callbacks* (see 10.4.4).

Arguably, there should also be an Event Processing Callback for the Open event to initialise the Balance to zero. However, as ModelScope initialises numeric attributes to zero when instantiating an object (See Section 7), this can be omitted.

4.1.9 Actors

Typically, the users of a system fall into different categories, with different responsibilities, interests and needs. *Actors* allow the User Interface to be tailored to match these different categories. For instance, in the simple banking system, two actors might be:

Customer Management	When a new customer wishes to open an Account, a Customer Manager is responsible for working through the procedures required to register the customer with the bank and open the account. This person deals with subsequent significant events in the relationship with the customer, such as changing address, opening further Accounts, closing Accounts and finally leaving the bank. The Customer Manager therefore needs access to Customer and Account objects and the Register, Change Address, Open, Close and Leave events.
Teller	The Teller handles deposits and withdraws. The teller needs access to the Account object and the Cash Deposit and Cash Withdraw events.

The Metadata for these actors is shown in Figure 10.

```
ACTOR Customer Manager
      BEHAVIOURS Customer, Account
      EVENTS Register, Change Address, Open, Close

ACTOR Teller
      BEHAVIOURS Account
      EVENTS Cash Deposit, Cash Withdraw
```

Figure 10

Actor specifications are independent of the rest of the model, the only constraint is that the objects and events used to define the subset for each actor exist. Actors are merely windows on the model and cannot alter or corrupt its behaviour.

Note that the subsets defined for different actors can overlap. Account is in both actor definitions and it is possible to have events that appear in more than one actor definition too, although that does not occur in this example.

4.2 Bank2

Please refer to the example model Bank2 for illustration of all the ideas and syntax introduced in this section.

4.2.1 Events in Context

The phrase *In Context* is used to describe whether or not an event can happen, based on the state of an object. For each state of an object, the events that are in context for that state are those for which there is an arrow (transition) leaving the state.

The events in context are determined from the state transition diagrams. Table 4 shows, for Customer and Account, the events in context for each state as specified in the diagrams in Figure 1.

Table 4

STATE	EVENTS IN CONTEXT
CUSTOMER	
@new	Register
registered	Open, Change Address
ACCOUNT	
@new	Open
active	Cash Deposit, Cash Withdraw, Close
closed	

This simple definition of In Context is for objects that have a single state transition diagram. As we shall see in the next section, a slightly more complex definition is needed for objects that have multiple state transition diagrams.

4.2.2 Objects and Behaviours

Simple objects, such as the Customer and Account objects defined above, only need a single state transition diagram to define their behaviour. More complex object behaviour may require the use of multiple state transition diagrams, and for these ModelScope uses *Behaviours*. Each Behaviour has its own state transition diagram, and Behaviours are linked together (or composed) to form the full definition of an object.

Every object in a model has an *Owning Behaviour* (defined using the keyword OBJECT in the metadata) and may have any number of other *Subsidiary Behaviours* (each defined using the keyword BEHAVIOUR in the metadata). **For the rest of this document, the word "Behaviour" will be used in the generic sense, to**

mean of either an Owning Behaviour or a Subsidiary Behaviour.

When more than one Behaviour is used to define an object, the lists of attributes of the Behaviours are added together to give the attributes of the object, and the state transition diagrams of the Behaviours are composed in parallel.

Composing the state transition diagrams works as follows:

- Each event in the Event Vocabulary of the object must appear in at least one of the state transitions diagrams of the object, but can appear in more than one.
- Where an event appears in more than one state transition diagram, it is in context for the object as a whole if it is in context in each of the diagrams where it appears.

Figure 11 shows two state transition diagrams, both describing an Account.

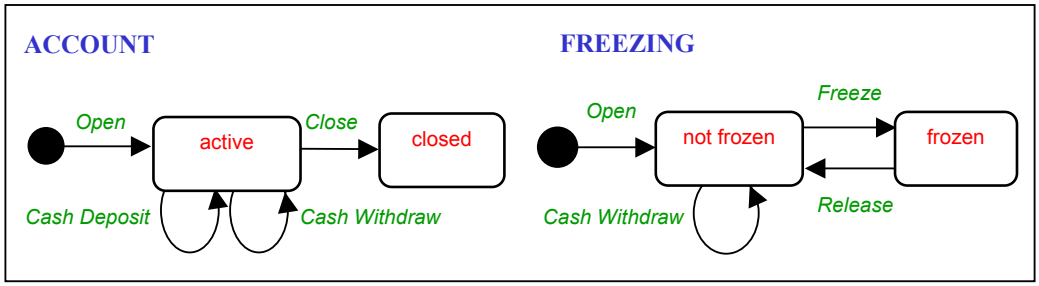

Figure 11

These two diagrams both belong to the Account object. The first diagram is the same as the Account in Figure 2.

The second diagram, Freezing, makes the following additional statements:

- Once an Account has been opened, it can be frozen and released any number of times.
- A cash withdraw can only be made when the Account is in the not frozen state.

Note the following:

- A cash withdraw can only be made when the Account is both active and not frozen, as this event appears in both of the diagrams.
- A cash deposit can be made whether the Account is frozen or not. The cash deposit event does not appear in the second diagram, so it is not constrained by it.
- Similarly, a close can take place whether the Account is frozen or not.

The Event Vocabulary of the Account object is now extended, as shown in Table 5.

Table 5

OBJECT	EVENT VOCABULARY
Customer	Register, Open, Change Address
Account	Open, Cash Deposit, Cash Withdraw, Close, **Freeze, Release**

The Account is now described as shown in Figure 12. The Behaviour called Freezing defines the second state transition diagram and it is linked to the main Account Behaviour using the INCLUDES statement in the Account object.

```
OBJECT Account
      NAME Account Number
      INCLUDES Freezing
      ATTRIBUTES …
      STATES …
      TRANSITIONS …

BEHAVIOUR Freezing
      STATES not frozen, frozen
      TRANSITIONS @new*Open=not frozen,
            not frozen*Freeze=frozen,
            frozen*Release=not frozen,
            not frozen*Cash Withdraw=not frozen
```

Figure 12

When an Account is instantiated, an instance of Freezing is automatically instantiated with it as part of the same object. These two Behaviours work together to determine how the Account behaves.

Note that the metadata for Freezing in Figure 11 has no NAME entry. Only Owning Behaviours (defined using the OBJECT keyword) require a NAME entry.

The Behaviours defined so far can be depicted diagrammatically as shown in Figure 13.

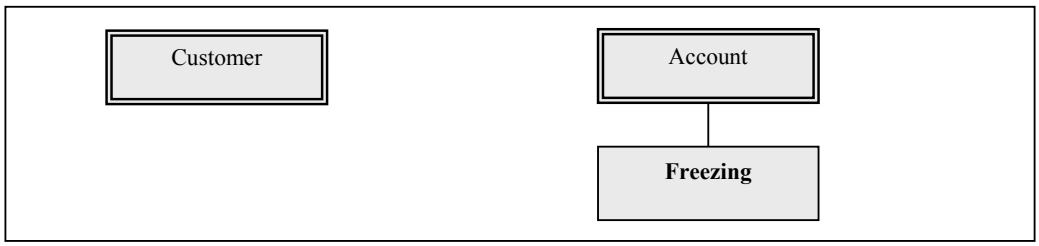

Figure 13

There are now two new events, for which the metadata is shown in Figure 14.

```
EVENT Freeze
      ATTRIBUTES Subject: Freezing

EVENT Release
      ATTRIBUTES Subject: Freezing
```

Figure 14

Note that Freezing, rather than Account, is used as the type of the reference attributes in the Freeze and Release events. Account would work equally well in this version of the model. However, as will be seen later in Section 4.4.1, using Freezing as the type will facilitate re-use.

In addition the Actor specifications need to be altered, as shown in Figure 15. Only the Customer Manager has access to the Freeze and Release events.

```
ACTOR Customer Manager
      BEHAVIOURS Customer, Account
      EVENTS Register, Change Address, Open, Close, Freeze, Release

ACTOR Teller
      BEHAVIOURS Account
      EVENTS Cash Deposit, Cash Withdraw
```

Figure 15

4.2.3 Derived Attributes

Derived attributes are calculated rather than stored. The fact that an attribute is derived is signalled by a "!" before the attribute name in the ATTRIBUTES metadata entry. A callback is needed to return the value.

ModelScope invokes the callback whenever the value of the attribute is needed, either to display at the user interface or because it is used in another callback.

Figures 16 and 17 show an example. Figure 16 shows the metadata.

```
OBJECT Customer
      NAME Full Name
      ATTRIBUTES Full Name: String, Address: String, !Total Balance: Currency
      STATES registered, left
      TRANSITIONS @new*Register=registered,
                  registered*Open=registered,
                  registered*Leave=left
```

Figure 16

Figure 17 shows the callback code that returns the value of the attribute. This is done by iterating through the Accounts owned by the Customer and adding the balances together.

```
package Bank2;

import com.metamaxim.modelscope.callbacks.*;

public class Customer extends Behaviour {

        public int getTotalBalance() {
                int totalBalance = 0;
                Instance[] accounts = this.selectByRef("Account", "Owner");
                for (int i = 0; i < accounts.length; i++)
                        totalBalance += accounts[i].getCurrency("Balance");
                return totalBalance;
        }

}
```

Figure 17

Note the following:

- The class containing the callback functions is named after the Behaviour, in this case: Customer.
- The member function is named "get" followed by the name of the Attribute (leaving out any spaces from the attribute name).
- `this.selectByRef("Account", "Owner")` returns an array of all the Account objects whose reference attribute "Owner" points to this Customer. In other words, it returns all the Accounts for this Customer.
- The function returns an integer because this is the way ModelScope represents currency amounts (see Section 7).

4.2.4 Derived States

So far, the states of a Behaviour have been driven by events as shown in the state transition diagrams. However, ModelScope also allows states to be derived, in a similar manner to derived attributes.

Suppose that an Account can only be closed if it is in credit. This is handled by having another Behaviour, called "Close Control", associated with an Account. The state transition diagram for Close Control is shown in Figure 18.

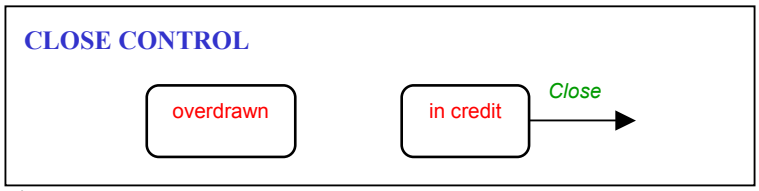

Figure 18

The diagram in Figure 18 makes the following statements:

- A close can occur when the Account is in credit.
- A close cannot occur when the Account is overdrawn.

No other events are mentioned, so no other events are constrained by this diagram.

ModelScope uses this constraint to determine when the Close event is in context for an Account. If the Account is "in credit" the close event is in context and appears as a possible event at the user interface. If the Account is overdrawn Close is not shown as a possible event.

This state transition diagram is different from previous ones in the following respects:

- There is no start state (black dot).
- The diagram is not "connected" – not all the state boxes are joined by arrows.

Both of these differences are because the diagram has states that are calculated rather than driven by events. The use of a calculated state is appropriate here because whether an Account is in credit or overdrawn depends on the cumulative effect of deposits and withdrawals, not just on the last event.

The metadata is shown in Figure 19. The fact that the state is calculated is signalled by a "!" in front of the Behaviour name in the BEHAVIOUR entry.

```
OBJECT Account
      NAME Account Number
      INCLUDES Freezing, Close Control
      ATTRIBUTES …
      STATES …
      TRANSITIONS …

BEHAVIOUR Freezing
      STATES …
      TRANSITIONS …

BEHAVIOUR !Close Control
      STATES in credit, overdrawn
      TRANSITIONS in credit*Close=@any
```

Figure 19

The INCLUDES statement for Account now specifies both Freezing and Close Control.

The transition in Close Control uses `@any` as the post-state. This means that any post-state is acceptable for this transition, as shown in the diagram by the fact that the Close arrow does not end at a state. The `@any` does not need to be included in the STATES entry.

Note that both "in credit" and "overdrawn" must be listed in the STATES entry for Close Control, even though "overdrawn" is not used by any transition. All states, other than those that start with "`@`", must be listed.

Figure 20 shows the callback code that calculates the state.

```
package Bank2;

import com.metamaxim.modelscope.callbacks.*;

public class CloseControl extends Behaviour {

        public String getState() {
                if (this.getCurrency("Balance") < 0) return "overdrawn";
                else return "in credit";
        }

}
```

Figure 20

The Behaviour structure is now as shown in Figure 21.

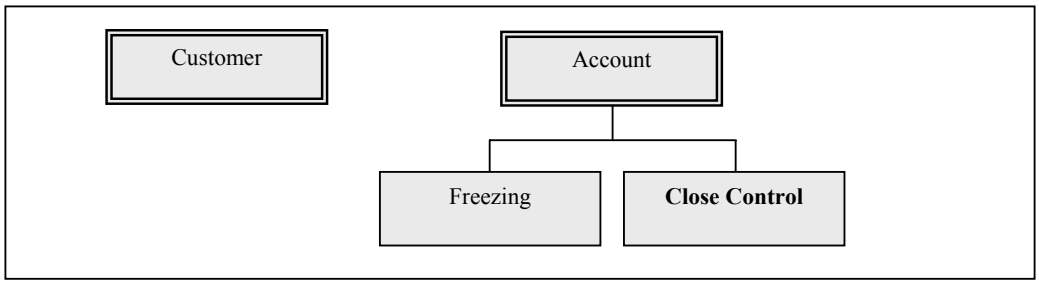

Figure 21

4.3 Bank3

Please refer to the example model Bank3 for illustration of all the ideas and syntax introduced in this section.

4.3.1 Behaviour Re-use

Behaviours can be re-used in different objects, as shown in Figure 22.

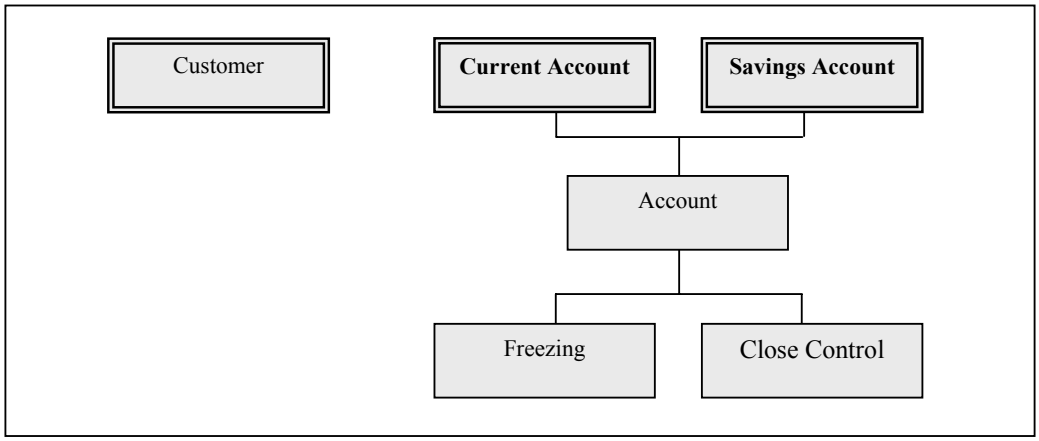

Figure 22

This new structure introduces two new objects: Current Account and Savings Account. Both of these use Account. Because they use Account, they also use the Freezing and Close Control Behaviours.

Outline metadata for Current Account, Savings Account and Account is shown in Figure 23.

```
OBJECT Current Account
      NAME Account Number
      INCLUDES Account
      STATES …
      TRANSITIONS …

OBJECT Savings Account
      NAME Account Number
      INCLUDES Account
      STATES …
      TRANSITIONS …

BEHAVIOUR Account
      INCLUDES Freezing, Close Control
      ATTRIBUTES Account Number: String, Owner: Customer, Balance: Currency
      STATES …
      TRANSITIONS …
```

Figure 23

Account is no longer an Owning Behaviour, so is declared using the BEHAVIOUR keyword and now has no NAME entry. Otherwise, the metadata for Account is unchanged from Figure 8.

Current Account and Savings Account are Owning Behaviours so have NAME entries. Note that the attribute used for the NAME entry does not have to be local that behaviour. In this case the attribute used to name Current and Savings Accounts (Account Number) belongs to Account.

When ModelScope instantiates an object, it instantiates the Owning Behaviour and all Subsidiary Behaviours below it in the INCLUDEs structure. Thus the Behaviours included when each object type is instantiated is shown in Table 6, with the Owning Behaviours shown in bold.

Table 6

OBJECT	BEHAVIOURS INSTANTIATED
Customer	**Customer**
Current Account	**Current Account**, Account, Freezing, Close Control
Savings Account	**Savings Account**, Account, Freezing, Close Control

4.3.2 *Post-State Constraints*

The two objects Current Account and Savings Account are so far identical. This is not realistic as Current and Savings Accounts normally have different purposes and different rules.

For example, a Current Account has a limit, set when the Account is opened, and a rule that the Account cannot be overdrawn beyond this limit. (There would also be behaviour rules specific to a Savings Account but these are not considered here).

The constraint on the balance for a Current Account is specified using a derived state Behaviour as shown in Figure 24.

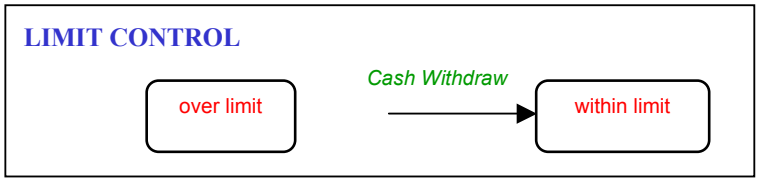

Figure 24

This diagram is similar to that shown in Figure 18, except that the constraint is expressed in terms of the terminating state ("within limit") rather than the initial state. This is because the constraint concerns the effect (resulting state) of the event. Figure 24 says that a withdraw on an Account must **result** in the state "within limit"; in other words, a withdraw that does not end in this state cannot happen. This kind of constraint is called a *Post-State Constraint*.

ModelScope can only check that post-state constraints are met after the effects of the event have been determined in the final stage of the event processing cycle. If an event violates a post-state constraint a message is issued to the user and the event has no effect on the states or attributes of the model. It is "Rolled Back", in database jargon.

Figure 25 shows the new Behaviour structure.

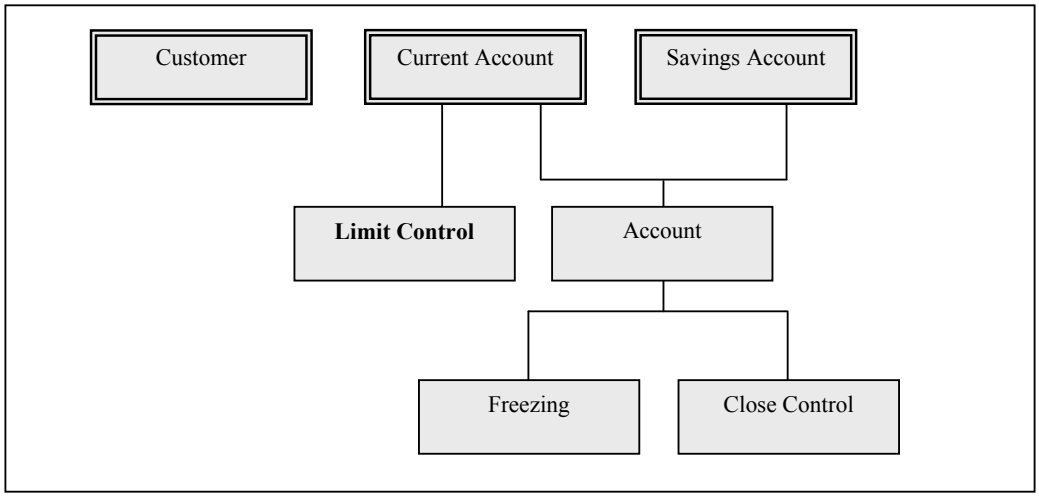

Figure 25

Figure 26 shows the metadata for Limit Control.

```
BEHAVIOUR !Limit Control
      STATES within limit, over limit
      TRANSITIONS @any*Cash Withdraw=within limit
```

Figure 26

Here the special state `@any` is used to indicate that we do not care what the starting state is.

Figure 27 shows the callback for the derived state of Limit Control.

```
package Bank3;

import com.metamaxim.modelscope.callbacks.*;

public class LimitControl extends Behaviour {

        public String getState() {
                int minBalance = 0 - this.getCurrency("Limit");
                if (this.getCurrency("Balance") < minBalance)
                        return "over limit";
                else    return "within limit";
        }

}
```

Figure 27

The Limit attribute is maintained by Current Account. How this is done is described below.

4.3.3 Generic Events

To support the withdrawal constraint described in the previous section a Current Account needs an attribute, Limit, that is set when the Account is opened and may subsequently be changed. This attribute cannot be put in the Account Behaviour because it does not apply to Savings Accounts. It is therefore maintained by the Current Account Behaviour. Figure 28 shows the modified state transition diagram for this behaviour including a new event to change the limit.

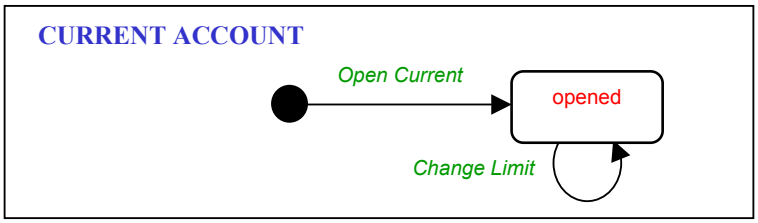

Figure 28

This diagram says that a Change Limit event can happen any time once the Current Account is opened.

We now need two forms of the Open event: Open Current (with a Limit attribute) and Open Savings (without a Limit attribute). Figure 29 shows the metadata for the Current Account and Savings Account Behaviours and the events Open Current, Open Savings and Change Limit.

```
OBJECT Current Account
      NAME Account Number
      INCLUDES Account, Limit Control
      ATTRIBUTES Limit: Currency
      STATES opened
      TRANSITIONS @new*Open Current=opened,
            opened*Change Limit=opened

OBJECT Savings Account
      NAME Account Number
      INCLUDES Account
      STATES opened
      TRANSITIONS @new*Open Savings=opened


EVENT Open Current
      ATTRIBUTES Account: Current Account, Account Number: String,
            Owner: Customer, Limit: Currency

EVENT Open Savings
      ATTRIBUTES Account: Savings Account, Account Number: String,
            Owner: Customer

EVENT Change Limit
      ATTRIBUTES Account: Current Account, Limit: Currency
```

Figure 29

Note the following:

- The states of Current Account and Savings Account are event driven so there is no "!" in front of the behaviour names, and no callbacks are necessary to calculate the state.
- Name co-incidence is used to update the Limit attribute therefore no callback is necessary for update.
- The type of the Account attribute in the events Open Current and Change Limit is "Current Account" and in the event Open Savings is "Savings Account". This is because these events are specific to the different Account types and must address them accordingly.

The reference attributes with type Account in the other events used by Accounts (Cash Deposit, Cash Withdraw, Close, Freeze, Release) are left as they are as these events are not specific to Current or Savings.

There is now an apparent problem with the definition of the model. We have replaced the old Open event (as defined in Figure 5) by Open Current in Current Account and Open Savings in Savings Account, but the two Behaviours Account and Freezing still use the old Open event. If the model is left as it stands, there would be two unwanted results:

- Accounts would have two creation events, the new Open Current (or Open Savings) and the old Open.
- The new creation events (Open Current and Open Savings) would not cause a transition to fire in the Account Behaviour (to take it from `@new` into the active state) or the Freezing Behaviour (to take it from `@new` into the not frozen state) because these still use the old Open event.

This is a common situation when re-using Behaviours and arises because the re-used Behaviours need events that have specific forms in the re-using objects. ModelScope provides a facility for defining Generic Events for such situations, as shown in Figure 30.

```
OBJECT Current Account
      NAME Account Number
      INCLUDES Account, Limit Control
      ATTRIBUTES Limit: Currency
      STATES opened
      TRANSITIONS @new*Open Current=opened,
         opened*Change Limit=opened

OBJECT Savings Account
      NAME Account Number
      INCLUDES Account
      STATES opened
      TRANSITIONS @new*Open Savings=opened

BEHAVIOUR Account
      INCLUDES Freezing, Close Control
      ATTRIBUTES Account Number: String, Owner: Customer, Balance: Currency
      STATES open, closed
      TRANSITIONS @new*Open=active,
         active*Close=closed,
         active*!Cash Deposit=active,
         active*!Cash Withdraw=active

BEHAVIOUR Freezing
      STATES not frozen, frozen
      TRANSITIONS @new*Open=not frozen,
         not frozen*Freeze=frozen,
         frozen*Release=not frozen,
         not frozen*Cash Withdraw=not frozen

EVENT Open Current
      ATTRIBUTES Account: Current Account, Account Number: String,
         Owner: Customer, Limit: Currency

EVENT Open Savings
      ATTRIBUTES Account: Savings Account, Account Number: String,
         Owner: Customer

EVENT Change Limit
      ATTRIBUTES Account: Current Account, Limit: Currency

GENERIC Open
      MATCHES Open Current, Open Savings
```

Figure 30

The Generic defines "Open" as a generic name that can match either "Open Current" or "Open Savings". A transition defined in terms of the generic Open will "fire" when either an Open Current or an Open Savings occurs. Semantically, this is equivalent to making two specific versions of Account, and similarly two versions of Freezing, one using Open Current and one using Open Savings for use in the Current Account and Savings Account respectively.

The old EVENT definition of the Open event (as shown in Figure 5) must be removed from the metadata as it is now superseded by the new Generic definition of Open. The BEHAVIOUR metadata for Account and Freezing are unchanged from the versions given in Figures 8 and Figure 12.

Events that are defined with an EVENT entry in the metadata are called *Concrete Events* to distinguish them from Generics. The Event Vocabulary of an Object is determined by concrete event transitions only, and adding a transition based on a Generic does not change the Event Vocabulary. For example, the Event Vocabularies of Current Account and Savings Account are now as in Table 7.

Table 7

OBJECT	EVENT VOCABULARY
Current Account	Open Current , Cash Deposit, Cash Withdraw, Change Limit, Close, Freeze, Release
Savings Account	Open Savings, Cash Deposit, Cash Withdraw, Close, Freeze, Release

Generics are therefore ignored when checking the correspondence between Events and Transitions as described in Section 4.1.5.

The Open event also appears in Customer (see Figure 1). This must be replaced with the two events Open Current and Open Savings as shown in Figure 31.

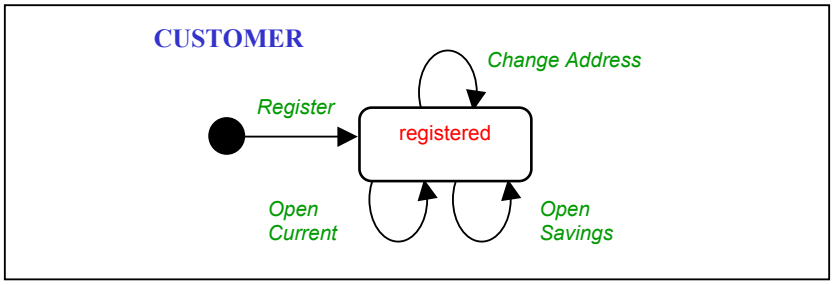

Figure 31

Had the state transition diagram for Customer been left as it was in Figure 1 the Open Current and Open Savings would not be in the Event Vocabulary of Customer, as Open is a Generic and the Event Vocabulary is determined only by concrete events. This would result in an error, as there is a Customer reference attribute in the metadata for both Open Current and Open Savings, so the correspondence check between Events and Transitions would fail.

The revised Customer metadata is shown in Figure 32.

```
OBJECT Customer
      NAME Full Name
      ATTRIBUTES Full Name: String, Address: String
      STATES registered
      TRANSITIONS @new*Register=registered,
         registered*Open Current=registered,
         registered*Open Savings=registered,
         registered*Change Address=registered
```

Figure 32

Also, Actors are always defined in terms of concrete events so the actors shown in Figure 15 are changed as shown in Figure 33.

```
ACTOR Customer Manager
      BEHAVIOURS Customer, Account
      EVENTS Register, Change Address, Open Current, Open Savings,
            Change Limit, Close, Freeze, Release

ACTOR Current Teller
      BEHAVIOURS Current Account
      EVENTS Cash Deposit, Cash Withdraw

ACTOR Savings Teller
      BEHAVIOURS Savings Account
      EVENTS Cash Deposit, Cash Withdraw
```

Figure 33

These actor specifications assume, somewhat unrealistically, that there are separate Tellers for Current and Savings Accounts. This unrealistic definition has been made to illustrate that Actor definitions can use both Owning and Subsidiary Behaviours. Using Current Account and Savings Account means that visibility will be restricted to these types of Account in the user interface. On the other hand, the Customer Manager actor is specified using the Account Behaviour making all Accounts visible.

4.3.4 Event Subscripts

A Transfer event allows funds to be transferred from one Account to another. The metadata for the Transfer event is shown in Figure 34.

```
EVENT Transfer
      ATTRIBUTES Source: Account, Target: Account, Amount: Currency
```

Figure 34

Transfer has two reference attributes with the type "Account", one for the source Account of the Transfer, and one for the target Account. "Account" is used as the type of these attributes, rather than "Current Account" or "Savings Account", because the two Accounts involved in a Transfer can be of either type.

The revised metadata for Account including transitions for Transfer is shown in Figure 35.

```
BEHAVIOUR Account
      INCLUDES Freezing, Close Control
      ATTRIBUTES Account Number: Integer, Owner: Customer, Balance: Currency
      STATES active, closed
      TRANSITIONS @new*Open=active,
            active*Close=closed,
            active*!Cash Deposit=active,
            active*!Cash Withdraw=active,
            active*!Transfer[Source]=active,
            active*!Transfer[Target]=active
```

Figure 35

The Transfer event is a shared event (see section 4.1.5). In the examples of shared events up to this point, sharing has been between two objects (or, more correctly, Behaviours) of different types. For instance, Open Current is shared between the two Behaviours Current Account and Customer. Transfer, however, is shared between two Behaviours of the same type, namely Account. This means that there are two transitions for the Transfer event in Account (shown in bold in Figure 35), one for transferring in and one for transferring out. The two transitions are distinguished using the names of the reference attributes from the Transfer event ("Source" and "Target") enclosed in square brackets as shown in Figure 35. This is called *Event Subscripting.*

Event subscripting is required when a single event addresses two (or more) Behaviour instances of the same

type, to distinguish the different transitions caused by the event. It is possible (though not the case here) that the different transitions caused by the event have different starting and ending states.

For the purposes of checking the correspondence between Events and Transitions, as described in Section 4.1.5, Transfer[Source] and Transfer[Target] are counted as distinct members of the Event Vocabulary of the Current Account and Savings Account objects. So, although there are two attributes (Source and Target) in Transfer that address these objects, there are also two matching members of the Event Vocabulary and the correspondence is maintained.

Because the Balance of the Account must be updated by a callback there is a "!" in front of the Event name for both Transfer transitions. The update callbacks are shown in Figure 36.

```
package Bank3;

import com.metamaxim.modelscope.callbacks.*;

public class Account extends Behaviour {

   public void processCashDeposit(Event event, String subscript) {
      int newBalance = this.getCurrency("Balance")
         + event.getCurrency("Amount");
      this.setCurrency("Balance", newBalance);
   }

   public void processCashWithdraw(Event event, String subscript) {
      int newBalance = this.getCurrency("Balance")
         - event.getCurrency("Amount");
      this.setCurrency("Balance", newBalance);
   }

   public void processTransfer(Event event, String subscript) {
      if (subscript.equals("Target")) {
         processCashDeposit(event, subscript);
      }
      else
         processCashWithdraw(event, subscript);
      }
   }

}
```

Figure 36

In the new callback, the parameter "subscript" is used to determine whether the callback is being invoked for the transition in the source Account or the target Account as the processing is obviously different for the two.

In addition, the withdraw side of a Transfer event is also subject to the constraints described in Figures 11 (Freezing) and 24 (Limit Control), as shown in Figure 37.

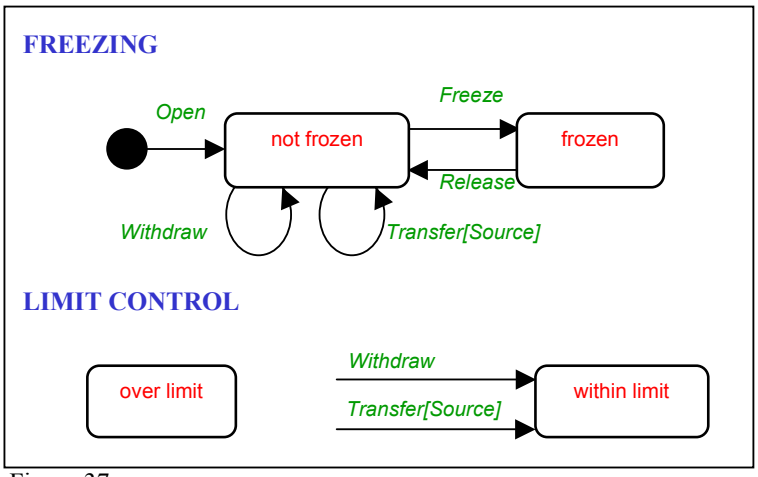

Figure 37

The revised metadata for these two Behaviours is shown in Figure 38.

```
BEHAVIOUR Freezing
      STATES not frozen, frozen
      TRANSITIONS @new*Open=not frozen,
         not frozen*Freeze=frozen,
         frozen*Release=not frozen,
         not frozen*Cash Withdraw=not frozen,
         not frozen*Transfer[Source]=not frozen

BEHAVIOUR !Limit Control
      STATES within limit, over limit
      TRANSITIONS @any*Cash Withdraw=within limit,
         @any*Transfer[Source]=within limit
```

Figure 38

An alternative and better approach is to define a Generic for any kind of withdraw event, as shown in Figure 39.

```
BEHAVIOUR Freezing
      STATES not frozen, frozen
      TRANSITIONS @new*Open=not frozen,
         not frozen*Freeze=frozen,
         frozen*Release=not frozen,
         not frozen*Withdraw=not frozen

BEHAVIOUR !Limit Control
      STATES within limit, over limit
      TRANSITIONS @any*Withdraw=within limit

GENERIC Withdraw
      MATCHES Cash Withdraw, Transfer[Source]
```

Figure 39

This gives exactly the same behaviour as the metadata in Figure 38 but makes maintenance of the model easier. If a new kind of withdraw (e.g. funds transfer to another Bank) is introduced into the model only the Generic definition needs to be changed and the constraints specified by Freezing and Limit Control will automatically apply.

As Figure 39 illustrates, it is possible to use event subscripts when defining a Generic.

4.4 Bank4

Please refer to the example model Bank4 for illustration of all the ideas and syntax introduced in this section.

4.4.1 More on Re-use

Suppose the Freeze and Release events are to be made available for Customer as well as for Accounts. If a Customer is frozen, all of their Accounts are frozen. In this sense placing a freeze on a Customer "cascades" to the Accounts held by the Customer.

The approach is to reuse the Freezing Behaviour across both the Account and Customer objects. The constraint on Withdraw is separated to another Behaviour, Freeze Control, as it requires a derived state based on the freeze status of both the Customer and the Account.

Figure 40 shows the new Behaviour structure.

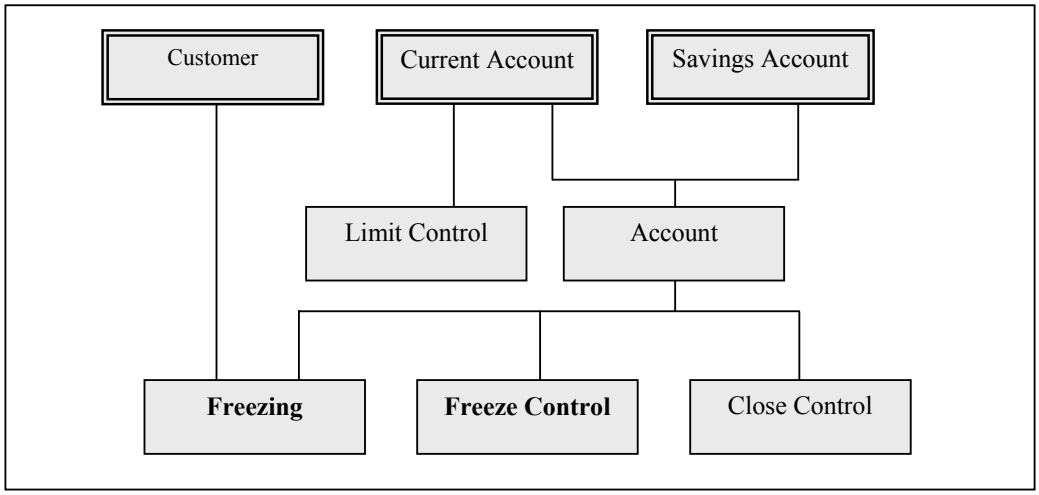

Figure 40

The revised state transition diagram for Freezing and the state transition diagram for Freeze Control are shown in Figure 41.

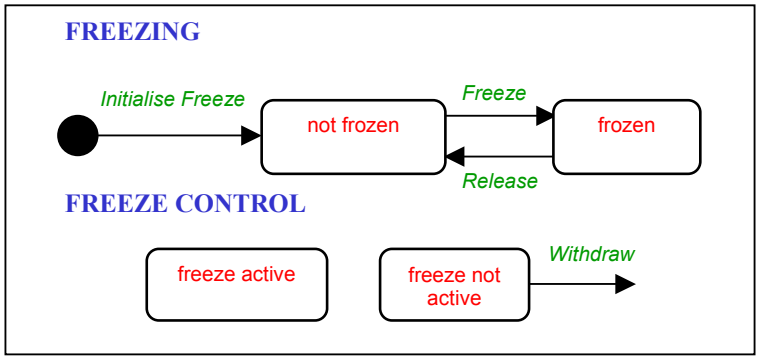

Figure 41

Freeze Control does not need an initialising event because its state is derived.

The metadata required is shown in Figure 42.

```
OBJECT Customer
      NAME Full Name
      INCLUDES Freezing
      ATTRIBUTES Full Name: String, Address: String
      STATES registered
      TRANSITIONS @new*Register=registered,
         registered*Open Current=registered,
         registered*Open Savings=registered,
         registered*Change Address=registered

BEHAVIOUR Account
      INCLUDES Freezing, Close Control, Freeze Control
      ATTRIBUTES Account Number: String, Owner: Customer, Balance: Currency
      STATES active, closed
      TRANSITIONS @new*Open=active,
         active*Close=closed,
         active*!Cash Deposit=active,
         active*!Cash Withdraw=active,
         active*!Transfer[Source]=active,
         active*!Transfer[Target]=active

BEHAVIOUR Freezing
      STATES not frozen, frozen
      TRANSITIONS @new*Initialise Freeze=not frozen,
         not frozen*Freeze=frozen,
         frozen*Release=not frozen

BEHAVIOUR !Freeze Control
      ATTRIBUTES !Customer Freeze Status: String
      STATES freeze active, freeze not active
      TRANSITIONS freeze not active*Withdraw=@any

GENERIC Withdraw
      MATCHES Cash Withdraw, Transfer[Source]

GENERIC Open
      MATCHES Open Current, Open Savings

GENERIC Initialise Freeze
      MATCHES Open[Account], Register
```

Figure 42

Note the following:

- The transition "`not frozen*Withdraw=not frozen`" has been removed from Freezing as the constraint on Withdraws is now handled by the new Freeze Control Behaviour.
- A Generic called Initialise Freeze has been used to allow the Freezing Behaviour to be initiated, by putting it into the not frozen state, when either a Register (for a Customer) or an Open (for an Account) takes place. Initialise Freeze is an example of a Generic defined in terms of another Generic (Open).

The Generic, Initialise Freeze, created to match any event that creates either a Customer or an Account, uses a subscript in its definition. The subscript is inherited when the Generic is expanded into its constituent concrete events. The definition of Initialise Freeze in Figure 42 is therefore equivalent to that shown in Figure 43.

```
GENERIC Initialise Freeze
      MATCHES Open Current[Account], Open Savings[Account], Register
```

Figure 43

The subscript [Account] is required because Open Current and Open Savings also appear as transitions in Customer. However, the transition for Open events in Customer, which reflect the fact that only a registered Customer can open an Account, should not cause the state of Freezing for the Customer to be initialised, only

the initial Register should do that. In Customer however, the Open events have the (implicit) subscript [Owner], as it is this attribute that addresses the event to the Customer object. Using the subscript [Account] means that Initialise Freeze will not match the Open transitions in Customer.

The type of the reference attributes in Freeze and Release, Freezing, (defined in Figure 14) do not have to be changed. Had Account been used as the type for the reference attribute (Subject) in these events, the Event Transition correspondence check described in Section 4.1.5 would fail when applied to Customer. This is because Customer now has the events Freeze and Release in its Event Vocabulary, but the event reference attributes would not address the Customer object.

The callback code for the Freeze Control Behaviour is shown in Figure 44. This prevents a Withdraw if there is a freeze active for either the Customer or the Account.

```
package Bank4;

import com.metamaxim.modelscope.callbacks.*;

public class FreezeControl extends Behaviour {

        public String getCustomerFreezeStatus() {
                return this.getInstance("Owner").getState("Freezing");
        }


        public String getState() {
                if (this.getState("Freezing").equals("frozen") ||
                    this.getString("Customer Freeze Status").equals("frozen"))
                return  "freeze active";
                else return "freeze not active";
        }

}
```

Figure 44

4.4.2 Event Handling Callbacks

Normally ModelScope passes events directly to the objects they affect, as described in the Event Processing Cycle in Table 3. However it is possible to instruct ModelScope to pass an event to a callback, instead of submitting it to the model. This allows more complex events to be defined as callback processing can spawn a number of separate events to the model from a single event entered by the user.

A full version of the Event Processing Cycle, showing where all types of Callback are invoked, is in Section 8.

Suppose that when a new customer registers, he or she normally gets a Current Account and may get a Savings Account. New Accounts are numbered sequentially. A new Set Up event allows all this to be done as a single input by the user. Figure 45 shows the revised state transition diagram.

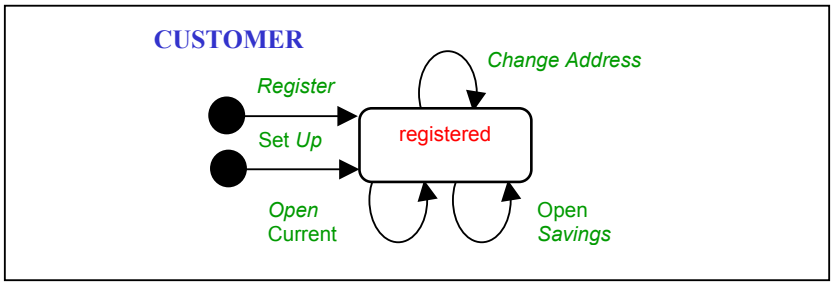

Figure 45

The old Register is left there to accommodate the case of a Customer that needs to be registered without opening any accounts, so there are now two creation events for Customer either of which will create a new Customer.

Figure 46 shows the required metadata.

```
OBJECT Customer
      NAME Full Name
      INCLUDES Freezing
      ATTRIBUTES Full Name: String, Address: String
      STATES registered
      TRANSITIONS @new*Register=registered,
         @new*Set Up=registered,
         registered*Open Current=registered,
         registered*Open Savings=registered,
         registered*Change Address=registered

EVENT !Set Up
      ATTRIBUTES Customer: Customer, Full Name: String, Address: String,
         Limit: Currency, Savings: Boolean
```

Figure 46

The new event has been added as a transition in Customer with a pre-state of `@new`. This means that the event will appear as a possible creation event for a new Customer.

The Set Up event has the same attributes as the Register event, plus two new ones:

- A Limit to be used in the Current Account.
- A Boolean that states whether or not a Savings Account is to be created.

The "!" in front of the event name indicates that, after the event attributes have been entered at the user interface and the event submitted, it is passed to a callback rather than to the model. The callback is shown in Figure 47 and does the following:

- Creates and submits a "Register" event to create the Customer.
- Creates and submits an "Open Current" event to open the Current Account.
- If the Boolean on the Set Up event is true, creates and submits an "Open Savings" event.

Note that the callbacks are coded in a class named after the event (SetUp) and that this class extends (subclasses) "Event".

```
package Bank4;

import com.metamaxim.modelscope.callbacks.*;

public class SetUp extends Event {

   public void handleEvent() {

      Instance myCustomer = this.getInstance("Customer");

      Event register = this.createEvent("Register");
      register.submitToModel();

      Event openCurrent = this.createEvent("Open Current");
      openCurrent.setNewInstance("Account", "Current Account");
      openCurrent.setInstance("Owner", myCustomer);
      openCurrent.setString("Account Number", AcInit.nextAcNo(myCustomer));
      openCurrent.submitToModel();

      if (this.getBoolean("Savings")) {
         Event openSavings = this.createEvent("Open Savings");
         openSavings.setNewInstance("Account", "Savings Account");
         openSavings.setInstance("Owner", myCustomer);
         openSavings.setString("Account Number", AcInit.nextAcNo(myCustomer));
         openSavings.submitToModel();
      }
   }

}
```

Figure 47

Note that, when a new event is created by invoking "`this.createEvent("Event Name")`", attributes from the source event (`this`) are transferred by name co-incidence. So it is not, for instance, necessary to populate the Full Name and Address attributes of Register, or the Limit attribute of Open Current.

Events that are created in callbacks are called *Sub Events*. Events that are entered at the user interface are called *User Events*. Note that Register can be either a Sub Event (if created as a result of a Set Up) or a User Event (if entered at the user interface).

A class called `AcInit` has been written to handle the assignment of the Account Number. This is shown in Figure 48. This code could have been included directly in the SetUp callback, but as it is going to be re-used later it makes more sense to place it in a separate class.

```
package Bank4;

import com.metamaxim.modelscope.callbacks.*;
import java.text.DecimalFormat;

public class AcInit {

   public static String nextAcNo(Instance anyThing) {
      Instance[] accounts = anyThing.selectInState("Account", "@any");
      return(new DecimalFormat("000").format(accounts.length + 1));
   }

}
```

Figure 48

Note that the parameter `anyThing` can be any object. It just provides a instance on which to invoke the select on Accounts. The result of the select is independent of the object used.

4.4.3 Attribute Handling Callbacks

ModelScope provides automatic validation that event attributes conform to their declared type (See Section 7). This validation can be supplemented, e.g. to ensure that a non-null string or non-zero numeric has been entered. Values can also be loaded, overwriting the default initial value and any value that has been loaded by name co-incidence.

Suppose that the Set Up event has to ensure that a Customer Name and Address are entered, and provide a default Limit of £250.00.

Revised metadata for Set Up is shown in Figure 49.

```
EVENT !Set Up
      ATTRIBUTES Customer: Customer, !Full Name: String, !Address: String,
        !Limit: Currency, Savings: Boolean
```

Figure 49

The attributes that have to be validated or given a default value have "!" in front of their names, indicating that Attribute Handling Callbacks are to be invoked. The callback code is shown in Figure 50.

```
package Bank4;

import com.metamaxim.modelscope.callbacks.*;

public class SetUp extends Event {

   public void setLimit(EventValueAttribute attribute, Instance selected,
      String subscript) {
         attribute.setCurrency(AcInit.defaultLimit());
   }

   public void setFullName(EventValueAttribute attribute, Instance selected,
      String subscript) {
         attribute.setRule("WORD_CHAR_RULE", "Full Name must be supplied");
   }

   public void setAddress(EventValueAttribute attribute, Instance selected,
      String subscript) {
         attribute.setRule("WORD_CHAR_RULE", "Address must be supplied");
   }

   public void handleEvent() {

      As Figure 47.

   }

}
```

Figure 50

To test that Full Name and Address have been entered, a ModelScope supplied validation script (`"WORD_CHAR_RULE"`) has been used. The validation scripts available for each value type are given in Section 7.1).

The code to supply the initial value of the limit has been added to `AcInit` as shown in Figure 51.

```
package Bank4;

import com.metamaxim.modelscope.callbacks.*;
import java.text.DecimalFormat;

public class AcInit {

   public static String nextAcNo(Instance anyThing) {
      Instance[] accounts = anyThing.selectInState("Account", "@any");
      return(new DecimalFormat("000").format(accounts.length + 1));
   }

   public static int defaultLimit() {
      return 25000;
   }

}
```

Figure 51

The events Open Current and Open Savings should also set default values for Account Number and Limit. Figure 52 shows the required metadata and callback code. The only change is the addition of "!" on front of Account Number and Limit attributes to indicate the use of Attribute handling Callbacks.

```
EVENT Open Current
   ATTRIBUTES Account: Current Account, !Account Number: String,
      Owner: Customer, !Limit: Currency

EVENT Open Savings
   ATTRIBUTES Account: Savings Account, !Account Number: String,
      Owner: Customer
```

Figure 52

Figure 53 shows the callback class for Open Current. Open Savings has a similar one (not shown here).

```
package Bank4;

import com.metamaxim.modelscope.callbacks.*;

public class OpenCurrent extends Event {

   public void setAccountNumber(EventValueAttribute attribute,
      Instance selected, String subscript) {
         attribute.setString(AcInit.nextAcNo(selected));
   }

   public void setLimit(EventValueAttribute attribute, Instance selected,
      String subscript) {
         attribute.setCurrency(AcInit.defaultLimit());
   }

}
```

Figure 53

4.5 Bank5

Please refer to the example model Bank5 for illustration of all the ideas and syntax introduced in this section.

4.5.1 Domain Rules and Business Rules

In general, the role of a business application is twofold:

- To mirror accurately the state of the business domain.

- To influence the behaviour of the domain to adhere to patterns of events and states that accord with the aims of the business.

In order to reflect these two roles, ModelScope models distinguish between behaviour rules that are inherent in the domain (*Domain Rules*) and behaviour rules that represent business requirements and policies (*Business Rules*).

Domain Rules are about the first role: they ensure that the behaviour of the model accurately matches the behaviour of the modelled domain by guaranteeing that the model cannot enter states that have no counterpart in reality. Business Rules are about the second role: they are about influencing the behaviour of the domain to favour or disfavour particular events for business reasons.

Business Rules, like traffic speed limits, are in general not perfectly enforceable. In order to fulfil the second role, a business application must exert appropriate influence to ensure that business rules are followed, but its obligations under the first role mean that it must be able to accommodate violation. Business Rules are therefore weaker than Domain Rules in the following sense: behaviour specified as Domain Rules cannot be violated, but the observance of Business Rules is at the users' discretion.

4.5.2 Business Rule Modelling

ModelScope distinguishes Domain Rules from Business Rules by giving each Behaviour in a model a TYPE. There are three types of Behaviour: *Essential, Allowed* and *Desired*. The first type is used for Domain Rules and the second two are used for Business Rules, as shown in Table 8.

Table 8

BEHAVIOUR TYPE	DESCRIPTION
FOR MODELLING DOMAIN RULES	
ESSENTIAL	Behaviour inherent to the domain. This is about what CAN happen. Violation is meaningless.
FOR MODELLING BUSINESS RULES	
ALLOWED	There are circumstances under which, because of a business rule or policy, an event is not supposed to take place. This is about what MAY (or is ALLOWED to) happen. Violation is not meaningless, but is discouraged.
DESIRED	There are circumstances under which, because of a business rule or policy, a certain event is required or desired. This is about what SHOULD (or is DESIRED to) happen. Violation is not meaningless, but not encouraged.

If no type is specified, the default is ESSENTIAL. As no types have been specified in the Bank model, so far, all Behaviours have defaulted to this type.

In Behaviours that define Domain Rules, whether an event is in context or not in context determines whether or not the event can or cannot take place at all. In Behaviours that define Business Rules, whether an event is in context or not in context determine whether it is allowed or not allowed and or desired or not desired as shown in Table 9.

Table 9

BEHAVIOUR TYPE	SEMANTICS
ALLOWED	• Events in context are ALLOWED • Events not in context are NOT ALLOWED
DESIRED	• Events in context are DESIRED • Events not in context are NOT DESIRED

ModelScope uses colour to provide feedback at the user interface, using the scheme shown in Figure 54, to show whether an event is in context or not for Business Rule Behaviours.

"Allowed and not desired" is the vanilla case and such events are not coloured. Other cases are coloured red, green or yellow as indicated.

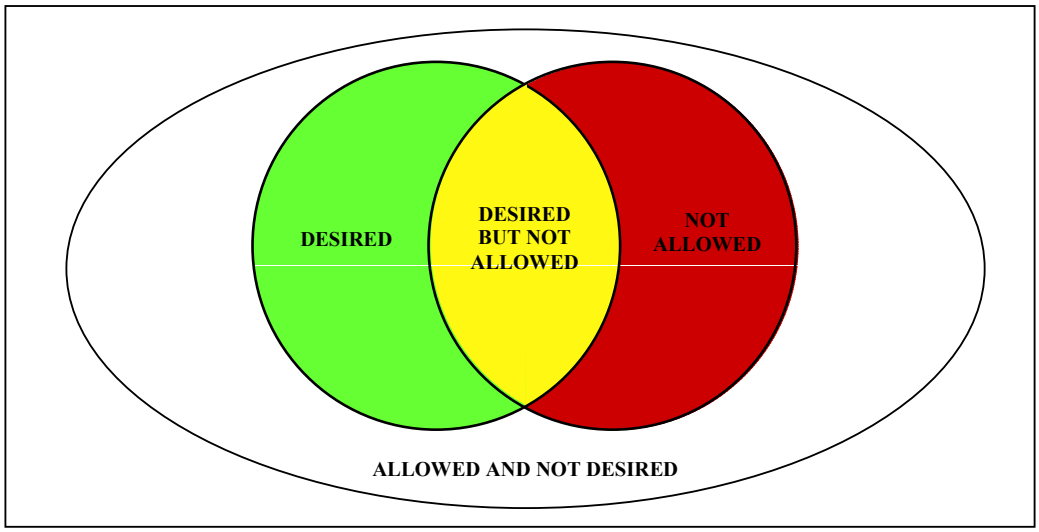

Figure 54

ModelScope does not let Business Rules Behaviours do any updating. ModelScope enforces this discipline by checking that Business Rules Behaviours (i.e. those with type ALLOWED or DESIRED):

- Have a derived state.
- Have only derived attributes.
- Do not have any Event Processing Callbacks on their transitions.

The reason for this discipline is that Business Rules Behaviours are purely advisory. In general, update processing (changing attribute or state values) only takes place when a transition fires. If a Business Rule Behaviour were to perform an update (e.g. by updating an attribute or event driven state), the update would only take place if the transition fires and this requires that the event is in context, i.e. allowed or desired. This would give the Business Rule more than just advisory significance.

Finally, events that appear in the transitions of Business Rule Behaviours do not contribute to the Event Vocabulary of the object. To be included in the Event Vocabulary, an event must appear in at least one Domain Rule (type ESSENTIAL) Behaviour.

4.5.3 Allowed Behaviour

The earlier Close Control (see Figure 19) is now amended as closing an Account that is overdrawn, although against Bank policy, does happen under some circumstances.

The Close Control Behaviour already obeys the rules described above, so the amendment is simply to add the type entry, as shown in Figure 55.

```
BEHAVIOUR !Close Control
      TYPE ALLOWED
      STATES in credit, overdrawn
      TRANSITIONS in credit*Close=@any
```

Figure 55

This change specifies that the Close event is allowed (because it is in context for the Close Control Behaviour) when the Account is in credit. If the Account is overdrawn, the Close is not allowed and will appear in red at the user interface. It will still, however, be possible to perform the event.

4.5.4 Post-State Constraints in Business Rules

It is possible to use post-state constraints in Business Rules. For instance the Limit Control Behaviour could be made a Business Rule as shown in Figure 56.

```
BEHAVIOUR !Limit Control
      TYPE ALLOWED
      STATES within limit, over limit
      TRANSITIONS @any*Withdraw=within limit
```

Figure 56

This Behaviour uses a post-state constraint to specify that a withdraw is only allowed if it leaves the Account within its credit limit. As previously described (Section 4.3.2), ModelScope reports on the constraint violations after it has finished processing the event. Changing this Behaviour from a Domain Rule into a Business Rule means that, instead of a failing an event that violates the constraint, ModelScope presents the user with a choice of either failing the event or letting it succeed.

When a post-state constraint is used in a Business Rule, ModelScope does not know in advance of the submission of the event that it violates the rule, so cannot colour the event in the user interface as described in Figure 54.

4.5.5 Desired Behaviour

Before a Customer leaves the Bank, all of their Accounts must be closed. This means that when a Customer notifies that Bank of their intention to leave, close becomes a desired event for the Accounts they own.

Customer now has the state transition diagram shown in Figure 57.

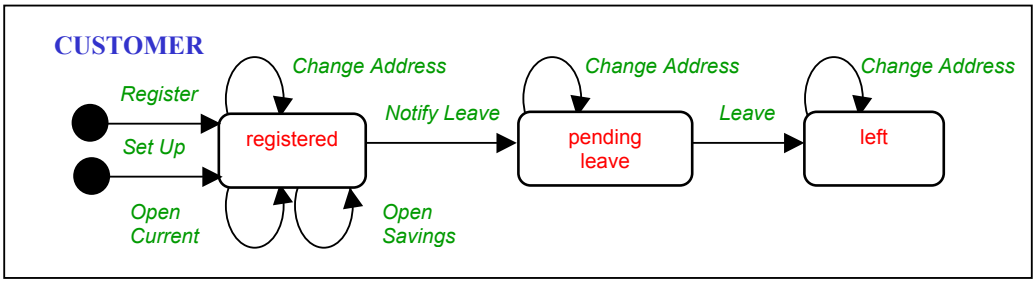

Figure 57

The Customer metadata (see Figure 46) is now amended as shown in Figure 58.

```
OBJECT Customer
      NAME Full Name
      INCLUDES Freezing
      ATTRIBUTES Full Name: String, Address: String, !Total Balance: Currency
      STATES registered, pending leave, left
      TRANSITIONS @new*Register=registered,
         @new*Set Up=registered,
         registered*Open Current=registered,
         registered*Open Savings=registered,
         registered*Notify Leave=pending leave,
         pending leave*Leave=left,
         @any*Change Address=@old
```

Figure 58

Note that the use of `@any` and `@old` to represent the three transitions for Change Address in Figure 57 as a single transition in the metadata. `@old` means that the post-state is the same as the pre-state.

A new Behaviour is needed to ensure that close becomes a desired event on Account when the Customer enters the pending leave state. The state transition diagram is shown in Figure 59.

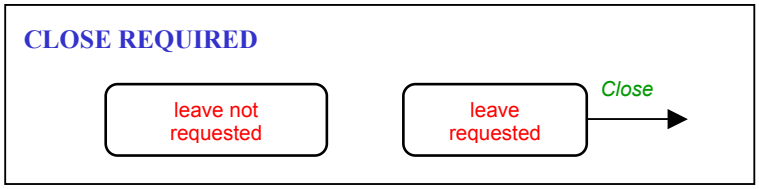

Figure 59

The metadata for this is shown in Figure 60.

```
BEHAVIOUR Account
      INCLUDES Freezing, Close Control, Freeze Control, Close Required
      ATTRIBUTES Account Number: Integer, Owner: Customer, Balance: Currency
      STATES open, closed
      TRANSITIONS @new*Open=active,
         active*Close=closed,
         active*!Cash Deposit=active,
         active*!Cash Withdraw=active,
         active*!Transfer[Source]=active,
         active*!Transfer[Target]=active

BEHAVIOUR !Close Required
      TYPE DESIRED
      STATES leave requested, leave not requested
      TRANSITIONS leave requested*Close=@any
```

Figure 60

The callback code for the state is shown in Figure 61.

```
package Bank5;

import com.metamaxim.modelscope.callbacks.*;

public class CloseRequired extends Behaviour {

      public String getState() {
           String custState = this.getInstance("Owner").getState("Customer");
           if (custState.equals("pending leave"))
                return "leave requested";
           else return "leave not requested";
      }

}
```

Figure 61

Note that if a Customer has notified the Bank of their intention to leave but has an Account that is overdrawn, the Close event is not allowed according to Close Control, but desired according to Close Required. This will cause the event to appear in yellow in the user interface, as described in Figure 47.

Business Rules that specify desired events have a close correspondence with workflow. Typically, in a production application, a desired event will cause a task to be placed on a workflow queue.

5 Metadata Conventions and Concepts

5.1 Metadata Structure

Metadata is specified using *Entries* and *Sub-Entries.*

Entries introduce new metadata elements of the model. Entries start with one of the following keywords shown in Figure 62.

```
MODEL . . .
OBJECT . . .
BEHAVIOUR . . .
EVENT . . .
GENERIC . . .
ACTOR . . .
```

Figure 62

Sub-entries add definition to entries. For instance, OBJECT has the sub-entries shown in Figure 63.

```
OBJECT . . .
        TYPE . . .
        NAME . . .
        INCLUDES . . .
        ATTRIBUTES . . .
        STATES . . .
        TRANSITIONS . . .
```

Figure 63

The sub-entries for each entry are listed in Section 6.

Each entry and sub-entry must start on a new line.

Entries may appear in any order in the Metadata file. Sub-entries always refer to the preceding entry.

Some sub-entries are optional – you only add them if they are needed – see Section 6.

Sub-entries may appear in any order following an entry.

5.2 Metadata Lexical Rules

All names in metadata must be alphanumeric and can have embedded spaces. ModelScope metadata is case-sensitive in handling names.

ModelScope is not case sensitive in handling keywords.

5.3 Conversion of Metadata names to Java names

Where metadata names are also used in Java callback code (Model_Name, Event_Names, Behaviour_Names and Attribute_Names), the spaces have to removed or replaced. The default is to remove spaces.

An entry in the ModelScope parameter file can be used to specify a replacement character (e.g. "_") if this is preferred. The form of the entry is:

SPACE_REPLACEMENT_CHAR=_

For more information about parameters, see the ModelScope "Getting Started" Guide.

5.4 Invisible Behaviour Attributes

Enclosing the metadata for an attribute and its type in parentheses, thus:

```
(Full Name: String)
```

in the definition of the attributes of an Object or Behaviour makes the attribute invisible at the user interface. Apart from being invisible, these attributes are treated exactly like visible attributes. All data transfer, callback invocation etc. is exactly the same.

5.5 State Specifiers

State specifiers are special symbols that are used to specify certain states of a behaviour. The state specifers are shown in Table 10.

Table 10

VALUE	MEANING
@new	The initial state of a behaviour, before any transition has taken place.
@any	Any state of the behaviour, apart from `@new`.
@old	Used as a post-state in a transition to indicate that the post-state is the same as the pre-state.

State specifiers are used in transitions, as described in Section 6.2.7.

State specifiers may also be used in callback code when selecting all instances that are in a certain state, using the selectInState function. This is described in Section 10.5.

5.6 Seed Instances

For each type of object defined in a model, ModelScope creates a single *Seed* pseudo-instance. These appear in the instance list in the user interface as:

```
(new Behaviour_Name)
```

and are used to create new instances of objects. The events that are in context for a seed are the creation events for the object.

A seed has the full set of behaviours appropriate to its type. All non-derived attributes in a seed are given the default value of their data type (see Section 7.1). Behaviours in a seed that have event driven states are in the state `@new`.

Callbacks work within seeds just as they do within normal instances.

6 Metadata Reference

This Section explains the syntax of ModelScope metadata.

6.1 Model

6.1.1 MODEL

Purpose: To name a Model. This name must the same as the Package name for the Callbacks associated with the Metadata.

Type: Entry

Format: `MODEL Model_Name`

Variations: None.

Rules: A Model File must have exactly one MODEL entry and it must be the first entry in the file.

6.2 Object and Behaviour

6.2.1 OBJECT and BEHAVIOUR

Purpose: To declare an Object or Behaviour.

Note: An Object is a special Behaviour. Object is used as the owning (top) Behaviour of an assembly of Behaviours that together model an object.

Type: Entry

Format:
```
OBJECT Behaviour_Name
BEHAVIOUR Behaviour_Name
```

Variations: If *Behaviour_Name* is prefixed by "!", the Behaviour has a Derived State Callback.

Rules: *Behaviour_Name* must be unique in the Behaviour/Event name-space.

6.2.2 NAME

Purpose: To specify which Attribute will be used to identify instances in the User Interface.

Type: Sub-entry

Format: `NAME Attribute_Name`

Variations: If the *Attribute_Name* is not unique within the Object and its directly and indirectly included Behaviours, the form *Behaviour_Name.Attribute_Name* can be used to disambiguate. *Behaviour_Name* is the Object or Behaviour owning the attribute.

Rules: Required as a sub-entry for an OBJECT. Not allowed as a sub-entry for BEHAVIOUR. *Attribute_Name* can be an attribute declared in the ATTRIBUTES of this Object or of any Behaviour included directly or indirectly by this Object. It may be either a stored or a derived attribute.

6.2.3 TYPE

Purpose: To specify whether the Behaviour represents behaviour whose observance is forced by

ModelScope, or behaviour whose observance is at the User's discretion.

Type: Sub-entry

Format: `TYPE Behaviour_Type`

Variations: None

Rules: *Behaviour_Type* must be one of ESSENTIAL, ALLOWED or DESIRED.
If the sub-entry is omitted, ESSENTIAL is assumed.
A Behaviour that is not ESSENTIAL must have a derived state, and cannot have stored (non-derived) attributes or any Event Processing Callbacks.

6.2.4 INCLUDES

Purpose: To define the structure of Behaviour composition.

Type: Sub-entry

Format: `INCLUDES Behaviour_Name , Behaviour_Name , …`

Variations: None

Rules: A Behaviour named in the includes sub-entry must not be an Object.
A Behaviour cannot include itself either directly or indirectly.
A Behaviour may not appear more than once in an includes structure.

6.2.5 ATTRIBUTES

Purpose: To declare the Attributes of the Behaviour.

Type: Sub-entry

Format: `ATTRIBUTES Attribute_Name : Type , Attribute_Name : Type , …`

Variations: If *Attribute_Name* is prefixed by "!" the attribute has a Derived Attribute Callback.

Parentheses used around and attribute and its type
(*Attribute_Name* : *Type*)
indicate that the attribute is invisible – i.e. not displayed at the user interface. Apart from not being displayed, invisible attributes are otherwise treated by ModelScope exactly like visible attributes.

Rules: *Attribute_Name* must be unique within the Object or Behaviour.

Type must be either an built-in type (see Section 7) for a value atribute, or a *Behaviour_Name* for a reference attribute.

6.2.6 STATES

Purpose: To declare the possible states of the Behaviour.

Type: Sub-entry

Format: `STATES State_Name , State_Name , State_Name , …`

Variations: None

Rules: The state specifiers (`@new, @any, @old`) are not declared.

6.2.7 TRANSITIONS

Purpose: To specify the transitions of the Behaviour.

Type: Sub-entry

Format: `TRANSITIONS` *`Transition , Transition , Transition , …`*

The form of a *`Transition`* is:

`Pre_State * Event_Specifier = Post_State`

Pre_State and *Post_State* are either a *State_Name* declared in the STATES sub-entry, or one of the state specifiers `@new, @any, @old`.

Event_Specifier is either an *Event_Name* from an EVENT entry, or a *Generic_Name* from a GENERIC entry.

Variations: The *Event_Specifier* can be subscripted. In this case it takes the form:

`Event_Name[Attribute_Name]` or *`Generic_Name[Attribute_Name]`*

If the *Event_Specifier* is pre-fixed by "!" an Event Processing Callback is invoked when the transition fires. An Event Processing Callback may be used whether the *Event_Specifier* is either a (possibly subscripted) *Event_Name* or *Generic_Name*.

Rules: `@old` can only be used as a *Post_State*

If the Behaviour is event driven (no "!" prefixing the *Behaviour_Name* in the OBJECT or BEHAVIOUR entry):

- There must be at least one Transition with a *Pre_State* of `@new`.
- `@any` can only be used as a *Pre_State*.

If the Behaviour has a derived state (a "!" prefixing the *Behaviour_Name* in the OBJECT or BEHAVIOUR entry):

- `@new` cannot be used.
- `@any` can be used as a *Pre_State* or a *Post_State*.

The set of Transitions in a Behaviour must be deterministic – i.e. no more than one transition can fire for a given event from a given state.

The Events used in Transitions must have type INTERNAL.

6.3 Event

6.3.1 EVENT

Purpose: To declare an Event.

Type: Entry

Format: `EVENT` *`Event_Name`*

Variations: If *Event_Name* is prefixed by !, the Event has an Event Handling Callback.

Rules: *Event_Name* must be unique in the Behaviour/Event name-space.

Only INTERNAL Events (see below) can use an Event Handling Callback.

6.3.2 TYPE

Purpose: To specify whether the Event can be used in Behaviour transitions, or is purely for output.

Type: Sub-entry

Format: `TYPE Event_Type`

Variations: None

Rules: *Event_Type* must be one of INTERNAL or EXTERNAL.
If the sub-entry is omitted, INTERNAL is assumed.
An Event that is EXTERNAL cannot be used in Transitions, or included in an Actor definition.

6.3.3 ATTRIBUTES

Purpose: To declare the Attributes of the Event.

Type: Sub-entry

Format: `ATTRIBUTES Attribute_Name : Type , Attribute_Name : Type , …`

Variations: If *Attribute_Name* is prefixed by "!" the attribute has an Attribute Handling Callback.

Rules: *Attribute_Name* must be unique within the Event.

Type must be either an built-in type (see Section 7) for a value attribute, or a *Behaviour_Name* for a reference attribute.

Only INTERNAL Events can use Attribute Handling Callbacks.

6.4 Generic

6.4.1 GENERIC

Purpose: To declare a Generic.

Type: Entry

Format: `GENERIC Generic_Name`

Variations: None

Rules: *Generic_Name* must be unique in the Behaviour/Event name-space.

6.4.2 MATCHES

Purpose: To specify the membership of a Generic.

Type: Sub-entry

Format: `MATCHES Event_Specifier , Event_Specifier , Event_Specifier , …`

Event_Specifier is either an *Event_Name* from an EVENT entry, or a *Generic_Name* from a GENERIC entry.

Variations: The *Event_Specifier* can be subscripted. In this case it takes the form

`Event_Name[Attribute_Name]` or `Generic_Name[Attribute_Name]`

Rules: A Generic cannot have itself as a member, directly or indirectly.

Subscripted and unsubscripted use of the same Event cannot be mixed in a Generic, either directly or indirectly.

An Event that has type EXTERNAL cannot be in the membership of a Generic.

6.5 Actor

6.5.1 ACTOR

Purpose: To declare an Actor.

Type: Entry

Format: `ACTOR Actor_Name`

Variations: The special *Actor_Name* "All" (ACTOR All) creates an Actor in which all Objects and Events are visible. This Actor has no BEHAVIOURS or EVENTS sub-entries.

If no Actor is specified, ModelScope creates an Actor All.

Rules: None

6.5.2 BEHAVIOURS

Purpose: To specify the Behaviours in a Actor.

Type: Sub-entry

Format: `BEHAVIOURS Behaviour_Name , Behaviour_Name , Behaviour_Name , …`

Variations: None

Rules: None

6.5.3 EVENTS

Purpose: To specify the Events in a Actor.

Type: Sub-entry

Format: `EVENTS Event_Name , Event_Name , Event_Name , …`

Variations: None

Rules: The events must all be type EXTERNAL.
Generics cannot be used.

7 Built in Types

This section describes ModelScope's built-in attribute types. For each, the following information is given:

Representation: How values are held internally by ModelScope. This is the Java type to use if you wish to manipulate values in callback code.

Initialisation: What initial value is given by ModelScope to Event and Behaviour attributes before any value is loaded, either by name co-incidence transfer or callback code.

Validation: The default validation of Event attributes performed by Java at the Post-presentation Stage of the Event Processing Cycle.

Validation Scripts: JavaScript validation scripts that can be applied to an attribute in Attribute Handling Callbacks to supplement the standard validation.

7.1 Value Types

7.1.1 Boolean

Representation: boolean

Initialisation: false

Validation: true or false

Validation Scripts: None

7.1.2 Currency

Representation: int (thus 123.45 is held as 12345).

Initialisation: 0 (representing 0.00)

Validation: integer (zero allowed)

Validation Scripts: POSITIVE_CURRENCY_RULE Value is a positive currency.
NON_NEGATIVE_CURRENCY_RULE Value is zero or a positive currency.

7.1.3 Date

Representation: Date

Initialisation: today

Validation: valid date

Validation Scripts: None

7.1.4 Integer

Representation: int

Initialisation: 0

Validation: integer (zero allowed)

Validation Scripts: POSITIVE_INTEGER_RULE Value is a positive integer.
NON_NEGATIVE_INTEGER_RULE Value is zero or a positive integer.

7.1.5 String

Representation: String

Initialisation: null string

Validation: not all spaces (null string allowed)

Validation Scripts: WORD_CHAR_RULE Value contains at least one alphanumeric character.
NOT_BLANK_RULE Value isn't empty or solely white space characters.

7.2 Reference Type

Representation: object pointer (representation internal to ModelScope)

Initialisation: null pointer (representation internal to ModelScope)

Validation: valid pointer, not the null pointer.

Validation Scripts: N/A

8 Event Processing Cycle

8.1 User Events

Table 11 describes the Event Processing Cycle for events entered by the user at the user interface.

Table 11

STAGE	DESCRIPTION	AUTOMATIC PROCESSING
Pre-presentation	After the user has chosen an object and an event at the user interface, but before the event attribute entry windows are displayed.	ModelScope populates the event attributes from the selected object by name co-incidence. ModelScope calls Attribute Handling Callbacks for each Attribute that has an "!" in front of its name in the ATTRIBUTES metadata for the Event. See 10.3.2.
Presentation	On completion of the Pre-presentation Stage, the event attributes are presented to the user at the user interface. The user can enter values, altering or over-writing those loaded automatically at the Pre-presentation Stage.	N/A
Post-presentation	After the user has entered values for event attributes and clicked the button to submit the event.	ModelScope checks that the content of each attribute conforms to the validation rules given in Section 7. ModelScope checks that attributes pass any supplemental attribute validation rules set by Attribute Handling Callbacks. If any check fails, a message is presented at the user interface, and no further processing of the event takes place.
Update	After all the checks in the Post-presentation Stage have been performed, and only if they all pass.	*For Events with no "!" in front of the Event_Name in the EVENT metadata.* ModelScope presents the event to each object addressed by the event. For each Behaviour of each object where the event fires a transition, ModelScope: • Populates the Behaviour attributes from the event attributes by name co-incidence. • If the Event_Specifier has as "!" in front it in the TRANSITIONS metadata, calls the Event Processing Callback. See 10.4.4. • If the Behaviour has an event driven state, updates the state to the transition post-state. After all Behaviours affected by the event have been updated, ModelScope checks whether any post-state constraints have been violated. Only transitions that fire are checked. Reporting of violations is done at the Pre-Return to UI Stage.
		For Events with "!" in front of the Event_Name in the EVENT metadata ModelScope passes the event to the Event Handling Callback. See 10.3.3.

Continued on next page.

Pre-Return to UI	After the Updating Stage has been completed, including processing of any events created and submitted to the model by callback code, and control is ready to pass back to the user interface.	ModelScope reports on any Post-State Constraints that have been violated. If none have been violated, the event succeeds. If one or more constraints in Behaviours with type ESSENTIAL have been violated, the current User Event is rolled-back. If the only violated constraints are in Behaviours with types ALLOWED and DESIRED, ModelScope presents a dialog at the user interface allowing the user to chose whether the event is to succeed or fail. If fail is chosen, the current User Event is rolled-back.

8.2 Sub Events

Table 12 describes the event processing cycle for Sub Events, i.e. events created and submitted to the model from Event Handling callbacks.

Table 12

STAGE	DESCRIPTION	AUTOMATIC PROCESSING
Pre-Update	After the callback code has populated the event attributes and submitted the event to the model.	ModelScope checks that the content of each attribute conforms to the validation rules given in Section 7. ModelScope checks that the event is in context for all the objects to which it is addressed by its reference attributes. If any check fails a run-time error is raised and the current User Event is rolled-back.
Update	After all the checks in the Pre-update Stage have been performed, and only if they all pass.	*Same as Update Processing in Table 11 for Events with no "!" in front of the Event_Name in the EVENT metadata.*

9 Callback Policy Rules

ModelScope enforces some rules about what may be done in each type of callback. These rules are called *Callback Policy Rules* and are required for two reasons:

1. To ensure that models adhere to the discipline of *encapsulation*. This makes it much easier to understand what a model is doing and to keep its complexity under control
2. To guard against callbacks that attempt to perform actions whose effect is *unspecified* and which would therefore cause indeterminate results or cause ModelScope to fail.

In summary, these rules are:

1. Rules ensuring encapsulation:
   - A Behaviour Attribute may only be updated by the Behaviour to which it belongs.

2. Rules guarding against unspecified effects:
   - Derived Attribute and Derived State callbacks cannot have any "side-effects". That is to say, they may (and must) return a value, but may not perform any updates.
   - Once an Event is being processed, i.e. being used to update objects, it cannot be altered.
   - Only one Event may be updating objects at a time. In other words, all objects that are affected by an event must be completely updated before any updating is done for another event.

The last of these requires a little more explanation.

ModelScope can be thought of as consisting of three layers, as shown in Figure 64.

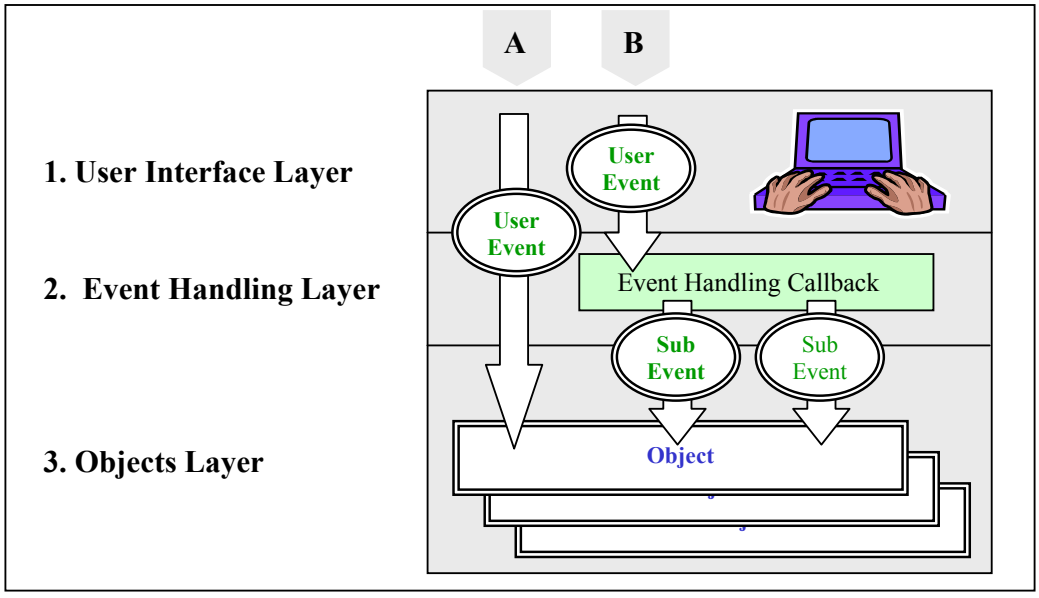

Figure 64

As shown in Figure 64, when a User Event is entered at the user interface, it is either:
A. Delivered directly to the Objects Layer or,
B. Delivered to the Event Handling Layer.

In the second case, the Event Handling Layer may, in its turn, deliver Sub Events to the Objects Layer.

ModelScope ensures that it is not possible for callbacks within the Objects Layer (i.e. Derived Attribute, Derived State and Event Processing callbacks) to create and submit further events. – only the User Interface

and Event Handling Layers can submit an event to the model. Because the Objects Layer must complete and relinquish control before another Event is delivered to it, and it will only relinquish control when it is finished with an event and quiescent, only one event is ever in process at a time.

Details of the Callback Policy Rules can be found in Section 10.5, in the last column of each table.

10 Callback Reference

10.1 Callback Signalling

ModelScope callbacks are fragments of Java code that supplement the metadata. The use of a callback is signalled in the metadata by using a "!".

Callbacks are of two types:

- *Event Class Callbacks* are signalled in the EVENT metadata and the Java code is placed in a Class that has the same name as the Event.
- *Behaviour Class Callbacks* are signalled in the OBJECT or BEHAVIOUR metadata and the Java code is placed in a Class that has the same name as the Behaviour.

Table 13 lists the different kinds of callback, when each is invoked, and how each is signalled in the metadata.

Table 13

TYPE OF CALLBACK	WHEN INVOKED (See Section 8)	HOW SIGNALLED WITH "!" IN THE METADATA	REFERENCE
EVENT CLASS CALLBACKS			
Attribute Handling	Pre-presentation	Prefix to the Attribute_Name in the EVENT definition.	10.3.2
Event Handling	Update	Prefix to the Event_Name in the EVENT definition.	10.3.3
BEHAVIOUR CLASS CALLBACKS			
Derived Attribute	On-the-fly	Prefix to the Attribute name in the OBJECT or BEHAVIOUR definition.	10.4.2
Derived State	On-the-fly	Prefix to the Behaviour name in the OBJECT or BEHAVIOUR definition.	10.4.3
Event Processing	Update	Prefix to the Event_Name in a TRANSITION entry of the OBJECT or BEHAVIOUR definition .	10.4.4

10.2 ModelScope Callback Types

The ModelScope callback language is an extension of Java provided by four Java types and methods on these types. The Java types are listed in Table 14.

Table 14

JAVA TYPE	DESCRIPTION
`Event`	An instance of an event, either created by ModelScope when the user submits an event from the user interface, or created in callback code using createNewEvent.
`Instance`	An instance of an object or a seed pseudo-instance. Conceptually, an Instance comprises all the behaviour instances belonging to the object.
`EventValueAttribute`	A value attribute of an event instance. (Used to access the attribute in an Attribute Handling callback).
`EventReferenceAttribute`	A reference attribute of an event instance. (Used to access the attribute in an Attribute Handling callback).

10.3 Event Class Callbacks

10.3.1 Class Structure

Event Class Callbacks are coded as functions in a class named after the event.

The class name is constructed from the *Event_Name* in the metadata by removing or replacing spaces.

The structure of the class is shown in Figure 65.

```
// mmmmm is the Model_Name with the spaces removed or replaced.
package mmmmm;

import com.metamaxim.modelscope.callbacks.*;

// eeeee is the Event_Name with spaces removed or replaced.
public class eeeee extends Event {

   // Attribute Handling for Value Attributes.
   // You can have a number of these for different value attributes
   // in the event. aaaaa is the Attribute_Name with spaces removed or
   // replaced.
   public void setaaaaa (EventValueAttribute attribute,
         Instance selected, String subscript) {
               // Function code.
   }

   // Attribute Handling for Reference Attributes.
   // You can have a number of these for different reference attributes
   // in the event. aaaaa is the Attribute_Name with spaces remove or replaced.
   public void setaaaaa (EventReferenceAttribute candidates,
         Instance selected, String subscript) {
               // Function code.
   }

   // Event Handling.
   // You can have at most one of these.
   public void handleEvent() {
               // Function code.
   }

`
```

Figure 65

10.3.2 Attribute Handling (Value and Reference)

These callbacks are invoked after the user has selected an instance and an event, but prior to display of data capture windows for the event.

For value attributes, the callback can set initial values (which override any value that might be loaded from the selected instance by name-co-incidence) and/or associate a JavaScipt rule with the attribute to be used to validate the value entered.

For reference attributes, the callback can refine the list of candidates to be presented at the user interface, and/or arrange that a particular value appears at the top of the presented list as the default choice.
The parameters are shown in Table 15.

Table 15

PARAMETER	DESCRIPTION
`attribute (for value attributes)`	The event attribute.
`candidates (for reference attributes)`	The set of instances for which the selected event is in context.
`selected`	The instance selected by the user.
`subscript`	The event reference attribute that corresponds to the instance selected by the user.

Note that there is always a single attribute of the event that corresponds to the instance selected by the user. If there is an Attribute Handling callback associated with this attribute it will not be called, as this attribute already has a value.

10.3.3 Event Handling

This callback is invoked in two ways:
- For a User Event, immediately after the user has entered data into the event and submitted it.
- By using submitToCallback() on a Sub Event.

The callback can:
- Perform further validation on the event – for instance, cross-field checking.
- Change the event attribute values entered at the user interface.
- Create internal events (Sub Events), populate their attributes, and submit them to further callbacks or to the model.
- Create external events and write them to the Log.
- Cancel the current User Event using rollback().

10.4 Behaviour Class Callbacks

10.4.1 Class Structure

Behaviour Class Callbacks are coded as functions in a class named after the object or behaviour.

The class name is constructed from the *Behaviour_Name* in the metadata by removing spaces.

The structure of the class is shown in Figure 66.

```
// mmmmm is the Model_Name with the spaces removed or replaced.
package mmmmm;

import com.metamaxim.modelscope.callbacks.*;

// bbbbb is the Behaviour_Name with spaces removed or replaced.
public class bbbbb extends Behaviour {

   // Derived Value Attribute.
   // You can have a number of these for different attributes in the behaviour.
   // aaaaa is the Attribute_Name with spaces removed or replaced.
   // ttttt is the type of the Java representation of the attribute (boolean,
   // date, int, String). See Section 7.1.
   public ttttt getaaaaa () {
               // Function code.
   }

   // Derived Reference Attribute.
   // You can have a number of these for different attributes in the behaviour.
   // aaaaa is the Attribute_Name with spaces removed or replaced.
      public Instance getaaaaa () {
               // Function code.
   }

   // Derived State.
   // You can have at most one of these.
   public String getState () {
               // Function code.
   }

   // Event Processing.
   // You can have a number of these for different transitions in the
   // behaviour.
   // eeee is the Event_Name or Generic_Name with spaces removed or replaced.
   public void processeeeee (Event event, String subscript) {
      // Function code.
   }
}
```

Figure 66

10.4.2 Derived Attribute (Value and Reference)

These callbacks are invoked on-the-fly to calculate and return the value of a derived attribute.
The type of the returned value must match the type of the attribute.

10.4.3 Derived State

This callback is invoked on-the-fly to calculate and return the value of a derived state.
Only the string values defined in the STATES sub-entry for the Object or Behaviour are allowed as return values.

10.4.4 Event Processing

These callbacks are called when a transition fires in a behaviour.

The callback can:

- Retrieve attribute values from the event being processed.
- Update attribute values of this behaviour.
- Create external events and write them to the Log or submit them to a callback.
- Cancel the current User Event using rollback().

The parameters are listed in Table 16.

Table 16

PARAMETER	DESCRIPTION
`event`	The event currently being processed.
`subscript`	The event reference attribute that addressed the event to this behaviour.

10.5 Language Reference

This Section provides a reference for the Callback Language.

Some of the methods in the language have type-specific variants. For instance, for retrieving the value of the attribute of an event, there are methods to be used according to the type of the attribute whose value is being obtained:

- `getBoolean("Attribute_Name")`
- `getCurrency("Attribute_Name")`
- `getDate("Attribute_Name")`
- `getInteger("Attribute_Name")`
- `getString("Attribute_Name")`

Rather than spell out each variant, the short-hand `get*` has been used.

Some methods have an optional parameter. This is indicated by showing the optional parameter in parentheses. Where a method has an optional parameter, ModelScope has two versions of the method with different signatures, one with the optional parameter and one without.

Note that not all methods are usable in all contexts. The last column of the tables specify policy rules about the valid use of each method, see Section 9. If these rules are not followed, ModelScope generates a run-time error.

10.5.1 Methods of Event

METHOD	PARAMETERS	DESCRIPTION	NOTES	ALLOWED USAGE (CALLBACK POLICY RULES)
`createEvent`	`"Event_Name"`	Returns a new event instance of the type specified by *Event_Name*. The attributes of the created event are populated from this event by name and type co-incidence.		Event Handling Event Processing
`get*`	`"Attribute_Name"`	Returns the content of the specified value attribute in this event.	1	Event Handling Event Processing
`getInstance`	`"Attribute_Name"`	Returns the content (an Instance) of the specified reference attribute in this event.		Event Handling Event Processing
`getEventType`		Returns the type (a String) of the event.	2	*Unrestricted*
`log`		Writes this event out to the Log		Event Handling Event Processing
`rollback`	`"Reason"`	Causes the current user event to be rolled-back. The *Reason* is shown in the Log.	3	Event Handling Event Processing
`set*`	`"Attribute_Name"` `Value`	Sets the content of the specified value attribute of this event to *Value*	4	Event Handling Event Processing (not allowed for the Event currently being processed)
`setInstance`	`"Attribute_Name"` `Instance`	Sets the content of the specified reference attribute of this event to *Instance.*		Event Handling Event Processing (not allowed for the Event currently being processed)
`setNewInstance`	`"Attribute_Name"` `"Behaviour_Name"`	Sets the content of the specified attribute of this event to a new instance of the object specified by *Behaviour_Name*.		Event Handling Event Processing (not allowed for the Event currently being processed)
`submitToCallback`		Invokes the Callback function handleEvent on this event.		Event Handling Event Processing
`submitToModel`		Submits this event to the model – exactly as if the event had been submitted directly from the User Interface.		Event Handling

Notes

1. The type of the value returned is the Java representation of the attribute. See Section 7.1.
2. Useful when an Event Processing callback is associated with a Generic, to determine the event that has matched the Generic.
3. It is always the user event that is rolled back, even if invoked on a sub event.
4. The Value parameter must conform to the Java representation of the attribute. See Section 7.1.

10.5.2 Methods of Instance

METHOD	PARAMETERS	DESCRIPTION	NOTES	ALLOWED USAGE (CALLBACK POLICY RULES)
`get*`	`("Behaviour_Name") "Attribute_Name"`	Returns the content of the specified value attribute in this instance.	1 3	*Unrestricted*
`getInstance`	`("Behaviour_Name") "Attribute_Name"`	Returns the content (an Instance) of the specified reference attribute in this instance.	1	*Unrestricted*
`getNullInstance`		Returns a null instance.		*Unrestricted*
`getObjectType`		Returns the name (a String) of the object.		*Unrestricted*
`getState`	`"Behaviour_Name"`	Returns the state (a String) of the specified behaviour within this instance		*Unrestricted*
`isEqual`	`Instance`	Returns true if this is the same instance as *Instance*.	4	*Unrestricted*
`isNull`		Returns true if this is a null instance.		*Unrestricted*
`isSeed`		Returns true if this is a seed pseudo-instance.	5	*Unrestricted*
`selectByRef`	`"Behaviour_Name" "Attribute_Name"`	Returns an array if instances, all of which include the specified behaviour and have the specified attribute referencing this.	6	*Unrestricted*
`selectInContext`	`"Behaviour_Name" "Event_Name" ("Subscript")`	Returns an array if instances, all of which include the specified behaviour and have the specified event with the specified subscript in context. The last parameter is optional. If omitted, the event can be in context for any subscript value. See Notes.	7 8	*Unrestricted*
`selectInState`	`"Behaviour_Name" "State"`	Returns an array if instances, all of which include the specified behaviour in the specified state.	7 9	*Unrestricted*
`set*`	`("Behaviour_Name") "Attribute_Name" Value`	Sets the content of the specified value attribute of this instance to *Value.*	1 2 10	Event Processing (allowed for Attributes of current Behaviour only)
`setInstance`	`("Behaviour_Name") "Attribute_Name" Instance`	Sets the content of the specified reference attribute of this instance to *Instance.*	1 2	Event Processing (allowed for Attributes of current Behaviour only)
`setNull`	`("Behaviour_Name") "Attribute_Name"`	Sets the content of the specified reference attribute of this instance to a null instance.	1 2	Event Processing (allowed for Attributes of current Behaviour only)

Notes

1. The first parameter is optional, and only needed if this instance has more than one attribute named *Attibute_Name.*

2. It is only possible to set values of stored (as opposed to derived) attributes
3. The type of the value returned is the Java representation of the attribute. See Section 7.1.
4. Using Java == (double equals) for instance equality will not produce the right result.
5. Useful to determine if an instance returned by a Select is a seed.
6. *Attribute_Name* must be a local attribute of *Behaviour_Name* – i.e. defined in its ATTRIBUTES entry.
7. The results do not depend on the instance on which the method is invoked.
8. Example: this.selectInContext ("Account" , "Transfer" , "Source") selects all Accounts for which Transfer[Source] is in context, whereas this.selectInContext ("Account" , "Transfer") selects all Accounts for which either Transfer[Source] or Transfer[Target] is in context.
9. The state specifiers `"@new"` and `"@any"` may be used for the State parameter.
10. The type of the *Value* parameter must conform to the Java representation of the attribute . See Section 7.1.

10.5.3 Methods of EventValueAttribute

METHOD	PARAMETERS	DESCRIPTION	NOTES	ALLOWED USAGE (CALLBACK POLICY RULES)
`set*`	`Value`	Sets this attribute to *Value*.	1	Attribute Handling
`setRule`	`"Rule_Name"`	Specifies the JavaScript rule to be used to validate user entered values for this attribute. See Section 7.1.		Attribute Handling

Notes

1. The type of the *Value* parameter must conform to the Java representation of the attribute . See Section 7.1.

10.5.4 Methods of EventReferenceAttribute

METHOD	PARAMETERS	DESCRIPTION	NOTES	ALLOWED USAGE (CALLBACK POLICY RULES)
`remove`	`Instance`	Removes *Instance* from the candidates list.		Attribute Handling
`removeAll`	`HashSet`	Removes from the candidates list all the instances in *HashSet* that are also in the candidates list.	1	Attribute Handling
`tag`	`Instance`	Causes *Instance* to be shown as the default by appearing first in the select list of candidates.		Attribute Handling
`untag`		Causes the list of candidates to appear in standard order – no specified first entry.		Attribute Handling

Notes

1. The parameter must be set up as an instance of the class java.util.HashSet whose members are Instances.

11 Instance File

ModelScope uses a text file (the Instances File) to provide persistency. This file is updated at the completion of processing a user event.

Normally, it is not necessary to look at this file. However, it is sometimes a useful aid to checking that a model is working correctly. In particular, the file shows the states of all behaviours, including derived states.

The file can also be edited. However, care must be taken not to destroy referential integrity if ModelScope internal identifiers are changed or deleted.

Figure 67 shows an example of an Instances File. The circled items are ModelScope internal instance identifiers.

```
# Metamaxim ModelScope Instances File written on Sun Aug 04 10:33:24 BST 2003

INSTANCE : 001 = 2
   BEHAVIOUR : Account = active
      Account Number : String = 001
      Owner : Customer = 1
      Balance : Currency = -10.00

INSTANCE : 002 = 4
   BEHAVIOUR : Account = active
      Account Number : String = 002
      Owner : Customer = 3
      Balance : Currency = 50.00

INSTANCE : 003 = 5
   BEHAVIOUR : Account = active
      Account Number : String = 003
      Owner : Customer = 3
      Balance : Currency = 0.00

INSTANCE : Roger Rabbit = 1
   BEHAVIOUR : Customer = registered
      Full Name : String = Roger Rabbit
      Address : String = The Warren

INSTANCE : Minnie Mouse = 3
   BEHAVIOUR : Customer = registered
      Full Name : String = Minnie Mouse
      Address : String = Behind the Fridge
```

Figure 67