

State Machines as Mixins

Ashley McNeile and Nicholas Simons, Metamaxim Ltd., London, U.K.

Abstract

Mainstream object modelling techniques use Statechart Diagrams as a means of modelling object behaviour. Research into how statecharts can be used in the context of class generalization hierarchies has focused on applying the Liskov Substitution Principle (LSP) to statecharts. This approach is problematic, and we describe three reservations.

We propose an alternative approach based on mixin-style composition of state transition diagrams. This avoids the problems we note in the LSP based approach; and is also a basis for separating descriptions of behaviour inherent to the modelled domain from behaviour that is not inherent, but a requirement of the system.

1 INTRODUCTION

This paper is about the formalisms used to model object behaviour at the domain¹ and analysis modelling stage of systems development, and is motivated by the need to create and manage re-usable behavioural abstractions when building domain and analysis models. It is informed by investigation into the possibilities of creating executable behavioural models that can be used to explore and validate system requirements at a very early stage in the development lifecycle.

The central point of the paper is that, for domain and analysis modelling, an approach based on a mixin-like style of composition and re-use of state machines (represented as state transition diagrams) is superior to the more conventional approach based on statecharts and generalization hierarchies, in two respects:

- It is simpler, because it avoids the need for model authors to adhere to rules governing how a sub-type should conform to its super-types.
- It is more expressive, because it supports the separation of descriptions concerning domain behaviour from descriptions concerning systems requirements.

¹ Domain modelling is sometimes referred to as “Conceptual” or “Essential” modelling.

2 STRUCTURE OF THIS PAPER

This paper is structured as four main parts:

- Sections 3 – 5 provide some background on statecharts and generalization hierarchies, why they are important, and the approach that has generally been adopted to synthesizing them into a coherent overall modelling framework based on the Liskov Substitution Principle.
- Section 6 describes our reservations with this Liskov based approach to synthesis.
- Sections 7 – 12 describe in outline our proposal for an alternative approach based on mixin-style composition of state transition diagrams, and shows how this avoids the difficulties described in Section 6.
- Section 13 shows how our proposed approach allows the separation of domain behaviour descriptions from required behaviour descriptions.

3 GENERALIZATION HIERARCHIES AND STATE MACHINES

Two paradigms, with different origins and histories, have become parts of the current modelling vernacular: Generalization Hierarchies and State Machines.

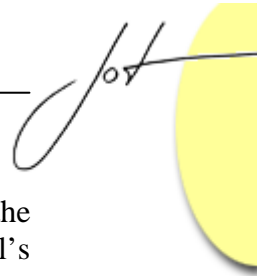
Generalization Hierarchies

The concept of a Generalization Hierarchy derives most significantly from the inheritance concepts of Object Oriented Programming (OOP), first introduced in the SIMULA-67 programming language. Successive generations of OOPs have successfully exploited the idea that software can be built by defining general components, with general capabilities, that can be sub-typed to make more specialized components. By supporting the inheritance of capabilities (attributes and methods) in a type hierarchy, these languages allow re-use and (properly used) promote economy of expression and ease of maintenance.

In all mainstream OO approaches, the Generalization Hierarchy has been adopted as a central construct for both OOA (Analysis) and OOD (Design) modelling. The idea is that, when analysing a domain to create an OOA model, a hierarchical taxonomy is made of domain concepts or objects, and that this is then mapped directly into a software class inheritance hierarchy in the OOD model.

State Machines

A State Transition Diagram (STD) is a notation that describes **states** (normally represented by ovals) and **transitions** (normally represented by arrows). The diagram represents the behaviour of a conceptual machine (a Finite State Machine) that engages in events that fire transitions and cause the machine to move from one state to another.



STDs have a long and respectable service record as a means of describing the behaviour of event driven software. An extended form of STD notation based on Harel's "Statechart"² notation [Harel 87] is incorporated in the UML. The statechart features adopted by the UML from Harel's notation include:

- Guards: Boolean conditions on transitions specifying when the transition may occur.
- Composite States: Allowing hierarchical nesting of states.
- Concurrent Regions: Allowing a single statechart to include separate, independently pursued, threads of states and transitions.

In domain and analysis modelling, the UML proposes statecharts as a tool for modelling the life cycles of application objects in terms of the domain events that affect their state and data (sometimes called "Protocol State Machines" [OMG 03 page 2-165]).

4 THE IMPORTANCE OF STATECHARTS

Although part of the UML armoury for behavioural specification, Statechart Diagrams have not been widely used.³ Instead, it has been more usual to use Interaction Diagrams (Sequence and Collaboration Diagrams) as the means of specifying the behaviour of object based software.

Recently, as part of its MDA (Model Driven Architecture) initiative, the Object Management Group has adopted executable semantics for the UML [OMG 03], which is intended to provide the basis for the creation of executable software directly from models. In particular, by automating the transformation from PIM (a Platform Independent Analysis Model) to PSM (a Platform Specific Design Model), model based code generation tools aim to reduce or even eliminate the dependencies between how a software application is defined and the platform on which it runs.

The UML execution semantics focus on the statechart as the basis for defining behaviour [OMG 03, Mell 02]. This is because the other forms of behavioural specification included in the UML, Interaction Diagrams, are essentially case-based descriptions of behaviour: a means of describing graphically the message trace of a single instance of execution, or at most a set of related executions. As such, they are unsuitable as a basis for model execution, either by code generation or metadata interpretation.

In so far as model execution and model based code generation are objectives of the modelling process, and we believe they should be, the statechart must be the central medium for behavioural specification.

² Throughout this paper we use the term *State Transition Diagram* (STD) to mean a simple form with states and transitions (no composite states, no concurrent regions and no guard conditions) and *Statechart* to mean the extended form including these features.

³ Statecharts are more frequently used in real-time/embedded software development. Our interest, however, is in information systems.

5 THE PROBLEM OF SYNTHESIS

The synthesis of statecharts and generalization hierarchies is problematic because the normal mechanisms of inheritance, as applied to attributes and operations (methods), do not apply in any obvious way. Attributes and operations take the form of a list or collection of individual items, and these lists can be merged and extended and items selectively refined or overridden. But a statechart is a single entity, and there is no obvious simple way of merging multiple statecharts, or of extending or refining all or part of a single statechart.

Researchers have therefore resorted to first principles to inform thinking on how statecharts should be handled in generalization hierarchies. A key idea is that a child, although a more specific version of its parent, is “substitutable” for the parent – in the sense that any context where the parent can operate, the child could too. Substitutability is commonly taken as a principle of sub-typing and is known as the Liskov Substitution Principle (the LSP) after Barbara Liskov [Lisk 88]. The UML has adopted substitutability as a key characteristic of generalization hierarchies [Booc 97 page 141].

Mainstream researchers into the synthesis of statecharts and generalization hierarchies have taken this as their basis [Hare 99, Schr 00, Eber 94, Cook 94, Simo 02]. As noted by all of these sources, the implication of applying the LSP to behaviour (as specified by a statechart) is that any sequence (trace) of event driven transitions that a parent handles, its child must handle too – otherwise the child could not successfully substitute for the parent. In other words, the set of transition traces of the child must be a superset of that of the parent.

6 THREE RESERVATIONS

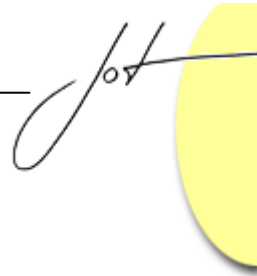
We have three reservations about applying the LSP to behaviour in this way, on the grounds of usefulness, practicability and necessity.

We should emphasize that this is not a general criticism of the LSP, only of its application to state transition behaviour in domain and analysis modelling.

Reservation on Grounds of Usefulness

Our first reservation is on the grounds of usefulness. If the child’s transition trace set is as wide or wider than that of its parent, as the LSP requires, then the child’s behaviour is more general than that of the parent. This is the inverse of the normal, intuitive, idea of a generalization hierarchy, where you expect the parent to be more general than the child.

Although cases can be constructed where the behaviour of a specialized object is more general than that of its parent, our contention is that common case is the other way round, and that trying to use the LSP requires artificial decisions. This cannot be proved in any formal way, but the following example illustrates the point.



A generic bank Account has a basic behaviour based on the events Open, Deposit, Withdraw, Close. The generic account supports both positive (credit) and negative (overdrawn) balances, and a variety of access mechanisms, including teller and ATM.

A Savings Account has the above behaviour, but because it is a savings vehicle, does not support going overdrawn. Also, it only supports withdrawal by transfer to another account, and not cash via teller or ATM.

It seems natural to model the Savings Account as a sub-type of Account: it is, after all, a “kind of” Account. However, the behaviour of a Savings Account is more restricted than that of its parent type, so could not substitute successfully for it – in violation of the LSP.

To conform to the LSP, the definition of the parent type can include only those behaviours that are also common to all its children. So including behaviour “x” in Account means that every type of sub-typed account must also include “x”. This is possible in principle, but will not result in a stable model. How do we know what new type of account the product development department of the bank might want to introduce next year, and what kinds of behaviour that product might include or not include? What if it doesn’t include “x”? The answer might be to remove “x” from the parent – with possible repercussions on other sub-types that may rely on it.

In situations like this, the LSP is forcing decisions at modelling time that cannot be properly made and, if made arbitrarily, may be unstable.

Reservation on Grounds of Practicability

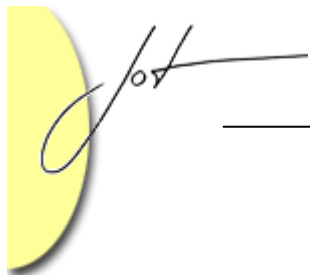
Our second reservation is on the grounds of the practicability. This is concerned with the following question: Suppose that we wish to sub-type STDs in a way that conforms to the LSP, what are the rules for ensuring conformance and are these rules simple and well understood enough to be used in practice? For this we look at some of the work that has been done to elucidate these rules [OMG 03 page 2-168, Schr 00, Eber 94, Cook 94 page 206, Simo 02].

Generally, all these sources agree on a basic set of techniques and rules that can be used to create an LSP consistent child statechart from parent statecharts (of which there may be more than one because of multiple inheritance). These rules and techniques, summarized in informal language, are:

- a) Keep all transitions and states of the parent statechart(s)⁴.
- b) Extend the child statechart by adding new transitions and states, not in any parent statechart⁵.
- c) Refine inherited states by decomposing into sub-statecharts.
- d) Refine inherited transitions by adjusting their source and/or target states to use substates of the inherited parent states, and possibly refine guards, subject to

⁴ In the case of multiple inheritance, the statechart of the child is the orthogonal composition of the statecharts of its parents. See UML Semantics [OMG 03 page 2-169].

⁵ Generally, this is done by orthogonal composition of a new concurrent state containing the new behaviour.



ensuring pre-conditions (transition source states and guards⁶) are not strengthened.

In their recent paper [Simo 02], Simons et al. state that rule (b) (extending) is unsafe and that extending should not be allowed.

It is not our purpose to critique the different formulation of these rules. Rather, we want to make the following observation. There are no other areas of OO modelling notation where the rules governing how to construct a well-formed model have anything like this complexity. As modelling (particularly during the early, domain modelling and analysis phases) is a highly iterative and dynamic activity, and a model is typically subject to frequent (and sometimes radical) re-factoring, we do not believe that observance of these rules is a practical proposition.

The difficulty of achieving substitutability is apparent to Simons et al. who open the concluding section of their paper [Simo 02]:

“The basic premise of component substitution is ‘no surprises’, yet (our) examples show how difficult it can be to avoid unexpected behaviour or even failure”.

In fact, the situation is actually more complex than these authors have allowed. All the sources we have examined assume that either:

- a) The child inherits from a single parent; or
- b) When there are multiple parents, the parents are disjoint: they have no states or transitions in common.

We do not think that this is a reasonable assumption when domain and analysis modelling. Consider the example of a car rental company. The object *Rental Car* might be sub-typed from both *Financial Asset* (concerned with the asset value, depreciation, etc.) and *Managed Asset* (concerned with the organizational responsibility for the asset, e.g. to arrange servicing and repairs). The event *Change of Ownership* (e.g. the vehicle is sold) is likely to be present as a transition in the statecharts of both super-types, as it has both financial and management implications. The rules give no information on how such an event should be handled.

We conclude, as do the authors of the UML Semantics Specification [OMG 03 page 2-166] about the whole area of statechart refinement:

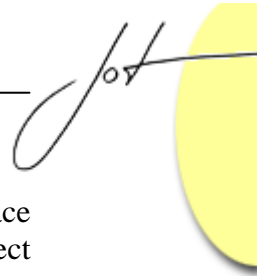
“...readers are reminded that this topic is still the subject of research, and that it is likely that other approaches may be defined either now or in the future”.

Reservation on Grounds of Necessity

Our final reservation is concerned with the premise that behavioural substitutability, of the form described by these authors, is necessary.

The basic assumption behind behavioural substitutability is that the statechart, as a specification of the event sequences that an object is able to handle, is public and relied

⁶ Not all the authors consider guards.



on by other objects. Consequently, if an object's statechart changes, so an event trace previously allowed is no longer allowed, or if the object is substituted by another object that does not allow the trace, the model will fail in some way.

However, it is not necessary for an object to make its statechart public. Another possibility is that:

- Statecharts are private to (hidden by) an object.
- Every object supports a run-time method that returns the set of events that it is currently able to handle⁷ based on its current state (essentially, a list of all the outgoing arrows from the statechart box corresponding to its current state).
- All other objects use this method to determine how to interact with the object, and make no assumption about the event sequencing that it may or may not support.

This scheme does not require behavioural conformance in order to achieve substitutability.

In our experience, there are good reasons, quite apart from the simplification of the model formation rules, for taking this approach. The exposure of the event sequence behaviour, meaning that one object knows about and may rely upon the sequencing of events permitted by another, is a source of coupling, making models hard to change. Minimizing coupling is particularly important if you are interested, as we are, in executing models as a means of validation during their development, when they are subject to frequent change.

7 BACK TO THE DRAWING BOARD

Developers exploiting the executable UML semantics to produce model execution tools have already pondered these difficulties. For instance, Mellor and Balcer [Mell 02 page 227] give the following advice on combining statecharts and type hierarchies (specifically: defining statecharts at both super and sub-type levels): *Don't do it.*

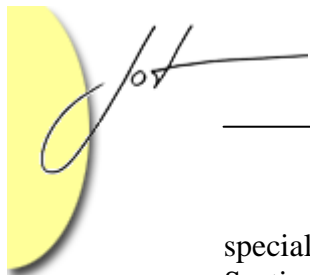
We are not content with this. Specializing and re-using behaviour is a central requirement of a complete and useful object-modelling paradigm. We believe that new thinking is required.

As a basis, we refer to Ebert and Engels [Eber 94], who make a distinction between *Invocational Consistency* (essentially, LSP style substitutability) and *Observational Consistency*. Observational Consistency requires that (in informal language):

If the trace of a child is censored so that only events present in the parent are visible, the trace is a valid trace of the parent.

This does not guarantee substitutability, as the possible (censored) traces of the child need only be a subset of the possible traces of the parent – not a superset as substitutability demands. But it does capture the notion that the child is a variant or

⁷ This requires “reflection” capabilities – i.e. that an object has run-time access to its own metadata.



specialization of the parent. In particular it avoids the instability problem described in Section 6 under Usefulness as the sub-types of Account do not need to support all the behaviours of the parent Account.

Our proposal is that Observational Consistency is used, and we now describe a mechanism for achieving it based on parallel composition of simple STDs.

8 COMPOSITION OF STDs

For simplicity, let us start with simple, flat, STDs:

- No composite states.
- No concurrent regions (alternatively called concurrent substates).
- No guard conditions.

(Readers unfamiliar with these features of statecharts may want to refer to [Booc 97 Chapter 21].)

We allow the behaviour of an object to be defined as the parallel composition of one or more STDs. The semantics of the composition is:

- For a single STD, an event is allowed provided that there is an outgoing transition for the event for the current state (as usual).
- For the composite object (specified using two or more composed STDs) an event is allowed by the object provided that it is allowed by each diagram in which it appears.

This follows the semantics of parallel composition⁸ proposed by Hoare in CSP [Hoar 85]. As an example, consider Figure 1.

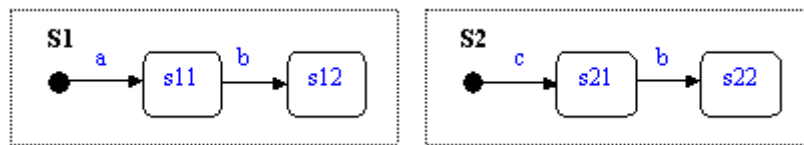


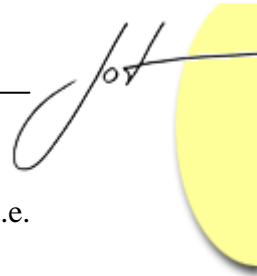
Figure 1: Example of STD Composition

An object whose behaviour is defined by the parallel composition of these two STDs ($S1||S2$) only allows the event **b** when the left hand STD is in state **s11** and the right hand is in state **s21**. So the valid (complete) traces for $S1||S2$ are:

- **a, c, b**
- **c, a, b**

The semantics of the composition is based on consideration of events alone. The states are private to each STD, so each STD has its own *state space*. Renaming states (e.g.

⁸ The parallel composition operator ($P||Q$), not the interleaving operator ($P|||Q$).



changing s_{11} to t_{11}) has no effect on the behaviour specified by the composition (i.e. the event traces it allows).

Note that the $S1||S2$ is Observationally Consistent with both $S1$ (valid trace: a, b) and $S2$ (valid trace: c, b). This style of composition guarantees the Observational Consistency of a composite whole with any single component STD, and with any parallel composition of component STDs. This can be shown as follows:

Suppose C is a composition of STDs and that C' is a composition of a subset of the STD components of C . If C is not Observationally Consistent with C' then there is some valid trace, T , of C which has the property that:

$T|_{C'}$ is not a valid trace of C'

Where $T|_{C'}$ means T censored to the event alphabet of C' .

In other words, if C' is presented with the event sequence $T|_{C'}$ it will, at some point, refuse an event because some component of C' does not allow it.

As the behaviour of C' is completely unaffected by events not in its alphabet, this same refusal would cause T to be an invalid trace of C . This is a contradiction, so T cannot exist, and C must be Observationally Consistent with C' .

This means that the modeller is not required to follow any rules (such as those outlined in Section 6 under Practicability) when authoring a model to achieve the desired degree of behavioural consistency, as the composition semantics guarantee it.

This is the first advantage we claim for the proposed approach.

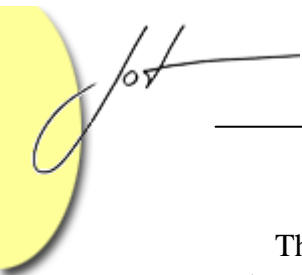
9 STDS AS MIXINS

Consider a Bank Account that has five “aspects” to its overall behaviour, characterized by five state spaces as follows. The Account can be:

- a) Pending open approval, open, or closed.
- b) Available, or frozen.
- c) In credit, or overdrawn.
- d) Balance above the account overdraft limit, or below it.
- e) Enabled for access from abroad, or disabled.

The behaviour of the Account can be modelled with five separate STDs, one for each of the five aspects, composed as described in the previous section. Each diagram captures a state space for one aspect, and the events that move it from state to state within that space.

In a sense, the Account could be said to “inherit” these individual STDs. But what status should the individual component STDs be given? Are they abstract super-types of Account? This seems to be nonsense, as the state diagrams have no meaning outside of their account context, and so do not seem to merit being identified as super-types.



There is another possible approach though, that conforms to current UML but does not require super-types. The statechart formalism used by the UML allows separate *concurrent regions* within a single diagram, having their own independently pursued states and transitions. The five STDs for the Account could be modelled as separate concurrent regions within the overall Account statechart. While this avoids the introduction of meaningless super-types, it precludes selective re-use of the individual components in the definition of other objects, as they are now bundled together. This is undesirable if there are a number of different kinds of account, each using some selection of the behavioural features these components represent, and we do not favour this approach either.

A more radical solution, and the one that we propose, is a pure mixin based solution [Brac 90]. In this approach, an STD is regarded as a stand-alone re-usable building block. Objects are defined using one or more STDs in parallel, CSP style, composition. In this approach there are no generalization hierarchies.

10 ANATOMY OF A MIXIN COMPONENT

Although we have concentrated on behaviour, as this is the basic motivation for the approach, our concept is that a mixin component has all the paraphernalia to be an object in its own right, namely:

- An *STD*.
- *Attributes* (both stored and derived).
- *Actions* that update attributes when events (transitions) take place.

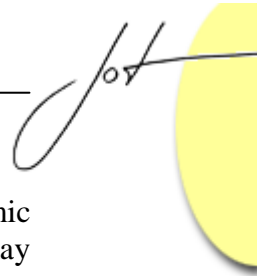
When two mixins are composed, the STDs are composed as described in Section 8, and the attributes and actions are merged. We assume that there are no merging conflicts – which is easily arranged if the attribute sets of different components are disjoint and actions in a component only update attributes belonging to the same component.

This does not necessarily mean that every mixin component can be instantiated by itself – some can, but others cannot and are only used to build larger assemblies.

(There is a lot more that would need to be described to define exactly what such a mixin component looks like. This is only intended to provide a sketch.)

11 ANALOGY WITH BILL OF MATERIAL STRUCTURES

With a mixin based approach, there is a close analogy with a *Bill of Materials* (BoM) used in manufacturing. A BoM describes how a manufactured product is built up from individual parts and sub-assemblies. The BoM structure is normally defined recursively: an end product is made from a number of components, each component being either an atomic part or a sub-assembly. Each sub-assembly is then the subject of its own BoM structure.



The point of identifying sub-assemblies is that they are re-usable, non-atomic components. A company that manufactures both dish and clothes washing machines may manufacture and install the same kind of pump in both types of machine. The pump would be identified as a sub-assembly incorporated in the BoM of both end products, and would have its own BoM describing how it is composed. The company might also sell its pumps, in which case the pump sub-assembly is also an end product.

The analogy between mixin based scheme and a BoM is close⁹, and is as follows:

- A mixin (an STD with associated attributes and actions) is an **atomic part**.
- A re-usable composition of mixins is a **sub-assembly**.
- An instantiatable object is an **end product**.

We have found it useful to arrange mixins in BoM style hierarchical structures for ease of re-use. Note that these are not generalization hierarchies, although they bear a relationship to conventional generalization hierarchies. (The nature of this relationship is beyond the scope of this paper.)

Nor are they like traditional module calling structures. In a module calling structure, the topology represents invocation relationships. In a mixin structure, the arrangement of the individual mixins within the structure is immaterial to the behaviour of the object being specified, and can be “re-factored” (e.g. to improve the re-use potential) without altering the behaviour it specifies.

12 DERIVED STATES AND GUARDS

When defining objects using mixins, we have found it useful to distinguish between two types of STD: *Event Driven* and *Derived State*:

- With an Event Driven STD, the state behaviour is defined by transitions of the STD, without reference to anything else.
- With a Derived State STD, the states are defined by a function (of the attributes of the object and perhaps other objects) that returns an enumerated type.

An Event Driven STD is the familiar, classical form of STD. An arrow emanating from a state means that that associated event can happen when the machine is in that state. When an event occurs, the associated transition fires and the machine is put into the state at which the arrow terminates. These diagrams are topologically connected, in the sense that every arrow must have both a starting and ending state.

A Derived State STD has a state that is calculated from other information (attributes) available to the STD. Conceptually, a *state function* is invoked to return the state on-the-fly, whenever required. (This is very similar to the familiar notion of a derived attribute.)

An arrow emanating from a state in a Derived State STD has the same meaning as in the Event Driven diagram. An arrow terminating in a particular state means that the state

⁹ A difference is that a BoM can use a given component many times. At least in our current formulation, a mixin can only be included once in the definition of a given object type.

is a necessary result of the transition.¹⁰ Derived State diagrams do not need to be topologically connected. Every arrow must have either a starting or an ending state, but does not need (and normally does not have) both.

The motivation for derived states is to maintain simplicity and transparency in description. This can be demonstrated by considering the familiar example of a Bank Account that can be *in credit* or *overdrawn*. The state function for an STD describing these states is shown in Figure 2.

```

if (self.balance >= 0)
    return `in credit`
else return `overdrawn`
endif

```

Figure 2: State Function for Account

This state function is simple and direct. To model these states in a state transition diagram where the states are event driven is messy. It is only possible if you either:

- Distinguish two types of *Withdraw* event (greater than the balance, or less), and similarly two types of *Deposit*, that then cause different transitions. See for instance [OMG 03 page 2-166], or
- Generate extra events inside the model that are fed back to cause further transitions. See for instance [Shla 92 page 71].

Both of these solutions are in our view unsatisfactory. They make the diagrams complex and they do not allow a declarative definition of the state computation.

Using the mixin approach, the two types (event driven and derived state) of STD can be composed as required to define an object. Figure 3 uses this to describe an Account that cannot be closed when it is overdrawn.

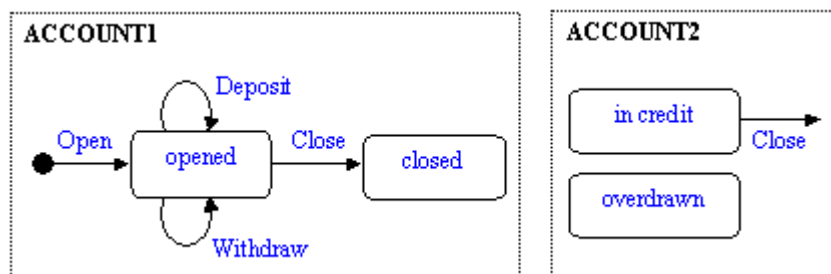
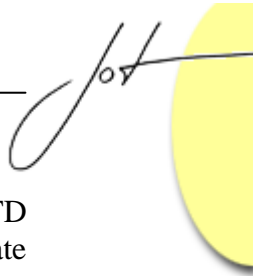


Figure 3: An Account that can only be closed when in credit

¹⁰ WARNING: This is significantly different from the idea of a post-condition, and should not be confused with it.



The first STD (**ACCOUNT1**) has an event driven state. The mixin with this STD maintains the account balance. The second mixin (**ACCOUNT2**) has a derived state (using the state function in Figure 2) and describes the constraint on closing. Note that **self** in Figure 2 refers to the Account as a whole, i.e. the combination of the two mixins.

Readers may observe that the constraint specified by **ACCOUNT2** would, in a UML statechart, normally be handled as a *guard* on the close transition in **ACCOUNT1**, specified by showing `Close[in credit]` against the arrow. While the use of mixins is notationally less succinct than using a guard, the way it allows the guard to be separated from the transition it affects is useful:

- It preserves the simplicity of the STD composition semantics.
- It allows guards to be re-used. **ACCOUNT2** can be composed with any object that has a `Close` event and a (numeric) `balance` attribute.
- It enables the separation of “indicative” and “optative” descriptions (as described below in Section 13).

Guard conditions can be functions of event parameters as well as object attributes, which derived states cannot. This might appear to make the derived state approach less powerful than guards. However, we allow a transition in a derived state STD to be constrained by its ending state as well as, or instead of, its starting state. The ending state is computed from object attributes after they have been updated by the event parameters, so can take into account their values. Figure 4 shows an account whose balance cannot be taken below a predefined limit, in addition to the constraint on closing.

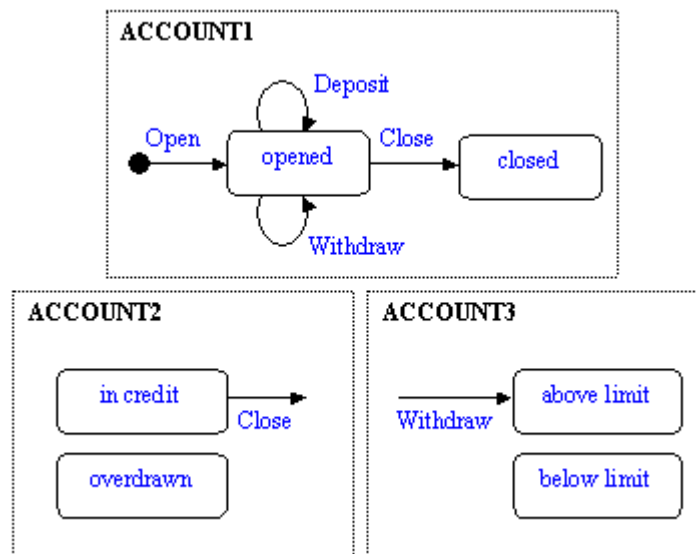


Figure 4: An Account with a post-state constraint on Withdrawals

It could be argued here that an extra mixin is not required: the state function for **ACCOUNT2** could have been redefined to differentiate three state values: `in credit`, `overdrawn but above limit`, or `below limit`. However, using a separate mixin

transparently allows the limit to be positive, representing a minimum balance requirement, as well as negative, representing an overdraft limit.

13 SEPARATION OF INDICATIVE AND OPTATIVE DESCRIPTIONS

When modelling, it is possible to distinguish between two types of description: those that refer to the application domain independently of the existence of the system, and those that pertain to the role of the system in its interaction with the domain. The motivation for this distinction has been made in [JaZa 95, Parn 95] and we will follow Jackson and Zave's terminology, using the word *indicative* to refer to descriptions of the domain, and *optative* to refer to descriptions pertaining to the role of the system.

In general, both kinds of description are necessary when developing a system. The reason for making indicative descriptions is that a system tracks the states of an external reality¹¹, in the sense that a library system tracks the books and members, a stock control system tracks stock levels, and an air traffic control system tracks aircraft. The system is then able to provide its users with information about the reality:

- Who has borrowed the book "Pride and Prejudice"?
- How many widgets do we have?
- Where is flight XX123?

When designing a system it is necessary to understand what states are possible in the domain because the system, in order to track the reality, must be able to mirror these states. Indicative models describe these states.

But a system will also enforce, or help to enforce, user defined rules or policies:

- A book cannot be on loan for more than a month.
- The number of widgets must not fall below the safety stock level.
- Two aircraft must not approach within a minimum distance of each other.

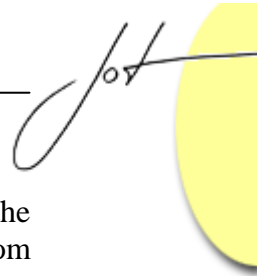
These reflect requirements of the system, as they describe what we want to be true when the domain and the system interact, and are the subject matter of optative descriptions.

STDs can be used to make both indicative and optative descriptions. Consider the following examples about a Lending Library, both of which could be expressed on a STD:

1. A book that is already out on loan to a member cannot be borrowed.
2. A book that has been classified as a reference book cannot be borrowed.

The first of these is indicative. It is a statement about the nature of books and lending, reflecting the fact that a book cannot be lent to a new borrower until it has been returned to the library by the previous one. This is true whether or not the library has a system for recording what books are on loan and to whom. The second is optative, and reflects the library's policy that reference books should be used on the premises and not taken out.

¹¹ Not all systems do this. This is not true for instance of purely transformational systems such as graphics processing, or authoring systems such as word processing or CAD.



However, it is both meaningful and physically possible to take a reference book out of the library, although it might be a requirement of a system to prevent or discourage this from happening.

Jackson [Jack 95 page 127] proposes a **Principle of Uniform Mood**: “*Never mix indicative and optative moods in the same description*”. We believe that this discipline should be followed, in particular for the following reason: It is not safe to assume, when constructing one part of a model, that optative constraints specified elsewhere in the model will always be obeyed. It may be, for instance, that reference books are taken from the library – albeit only occasionally and only at the discretion of the head librarian, who has the authority to waive the normal rule. So the mechanisms for recording the fact of “being on loan”, issuing reminders when overdue, etc., should be part of the model of a reference book just as they are for other books.

Distinguishing constraints that are indicative (and may be relied on), from those that are optative (and may be broken), is therefore important – because the distinction determines what behaviour possibilities must be allowed for, even if some of these possibilities are only exercised in exceptional circumstances. Maintaining this distinction clearly in the model is difficult if the two types of constraint are conflated in a single description.

So far, we have made the tacit assumption that the STDs in our mixins describe indicative constraints. Now suppose that the bank manager has the authority to allow a customer to withdraw beyond his/her limit. **ACCOUNT3** in Figure 3 needs to be reclassified as optative rather than indicative (**ACCOUNT1** and **ACCOUNT2** are still indicative). We are compliant with the Principle of Uniform Mood, because each mixin is either wholly optative or wholly indicative. Had the limit constraint been specified as a guard on the withdraw event in **ACCOUNT1** this separation could not not have been made.

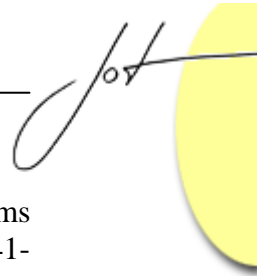
Ability to make this separation is the second advantage we claim for the mixin-based approach.

14 FURTHER WORK

As we say in the introduction to this paper our interest is in the use of executable behavioural models for requirements validation, and our current work is in this area. Further information about this can be found at <http://www.metamaxim.com>.

REFERENCES

- [Hare 87] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, 8, pages 231-274, 1987.
- [OMG 03] OMG, *Unified Modeling Language Specification version 1.5, March 2003*, <http://www.omg.org>. Document ref.: formal/03-03-01
- [Lisk 88] B. Liskov, "Data Abstraction and Hierarchy", *SIGPLAN Notices vol. 23, issue 5*, May 1988.
- [Booc 97] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1997.
- [Hare 99] D. Harel and O. Kupferman, "On the Inheritance of Statebased Object Behaviour", *Technical Report MCS99-12*, The Weizmann Institute of Science, Israel, 1999.
- [Schr 00] M. Schreft and M. Stumptner, "On the Design of Behavior Consistent Specializations of Object Life Cycles in OBD and UML", published in *Advances in Object-Oriented Data Modeling*, The MIT Press, 2000.
- [Eber 94] J. Ebert and G. Engels, "Dynamic Models and Behavioural Views", *International Symposium on Object-Oriented Methodologies and Systems*, LNCS 858, Springer-Verlag, 1994.
- [Cook 94] S. Cook and J. Daniels, *Designing Object Systems – Object-Oriented Modelling with Syntropy*, Prentice Hall International, 1994.
- [Simo 02] A. Simons, M. Stannett, K. Bogdanov and W. Holcombe, "Plug And Play Safely: Rules For Behavioural Compatibility", *Proc. 6th IASTED Int. Conf. Software Engineering and Applications*, (Cambridge MA: IASTED, 2002).
- [Mell 02] S. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison Wesley, 2002.
- [Hoar 85] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.
- [Brac 90] G. Bracha and W. Cook, "Mixin-based Inheritance", *Proc. of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1990.
- [Shla 92] S. Shlaer and S. Mellor, *Object Life Cycles - Modeling the World in States*, Yourdon Press/Prentice Hall, 1992.
- [JaZa 95] M. Jackson and P. Zave, "Deriving Specifications from Requirements: An Example", *ICSE17*, vol. 1995, pages 15-24, 1995.



-
- [Parn 95] D. Parnas and J. Madey, “Functional Documentation for Computer Systems Engineering”, *Science of Computer Programming* (Elsevier) 25(1), pages 41-61, Oct. 1995
- [Jack 95] M. Jackson, *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*, AddisonWesley, 1995

About the authors



Ashley McNeile is a practitioner with 25 years of experience in systems development and IT related management consultancy. His focus is research into requirements analysis techniques and model execution, and in 2001 he founded Metamaxim Ltd. to pioneer new techniques in this area. He has published and presented on object oriented development methodology and systems architecture. He can be reached at ashley.mcneile@metamaxim.com.



Nicholas Simons has been working with formal methods of system specification since their introduction, and has over 20 years experience in building tools for system design, code generation and reverse engineering. In addition, he lectures on systems analysis and design, Web programming and project planning. He is a co-founder and director of Metamaxim Ltd., and can be reached at nick.simons@metamaxim.com.