International Journal of Modelling and Simulation, Vol. 22, No. 1, 2002

VALIDATION LED DEVELOPMENT OF SOFTWARE SPECIFICATIONS

C.A. Lakos* and V.M. Malhotra**

Abstract

A software development methodology is defined that integrates software specification and validation efforts. The integration helps in achieving the twin goals of correct software with well-defined specifications that document it. The major focus of the paper is on dynamic life-cycle models. We indicate how language analysis of the problem description can be extended to derive not only a static class model for the system but also an initial dynamic life-cycle model. Consistency and completeness checks are provided that further drive the requirements elicitation. A case study is presented that clearly demonstrates the methodology.

Key Words

Object-oriented, life-cycle, validation and verification, requirements analysis, software specification

1. Introduction

The waterfall model [1] and its subsequent evolution into the spiral model [2] encapsulate the traditional wisdom in software development. The underlying principle in the two models is to divide the software development process into stages, each of which produces a *deliverable*. A deliverable not only defines the conclusion of the stage but also provides the natural focus for ensuring the correctness, so that the project continues to be on course to deliver the end product that the user wants. This assurance effort is called validation and verification. The verification efforts aim at ensuring that the end result of the stage is as per the input specifications. Validation ensures that the result is what the user needs. The correctness of the initial requirements definitions in setting the stage for an end product that satisfies the user needs cannot be overemphasized.

The gulf between application users and computer professionals, caused by differences in their expertise, emphasis, terminologies, and notations, makes the process error prone and time consuming. Many tomes have been written describing how computer professionals can aid communication between themselves and users. In addition, software development methodologies are emerging that shift the effort towards application users.

Object-oriented analysis and design has received a lot of attention as a methodology that bridges the gap between the world of application domain users and that of computer professionals. Application domain experts gain an understanding of their problems by forming abstractions in terms of objects and their interactions. Object-oriented methodologies focus on these objects to construct information models of the applications. The direct modelling of a problem domain into a computerized application eliminates errors that result from translation between models, and also enhances maintainability, as a small change to the problem will only result in a small change to the model [3].

Given the background in entity-relationship diagrams, the (static) information models encompassing classes, associations, and attributes are well defined and understood. However, in spite of the success and popularity of the object-oriented methodologies they continue to prove difficult for both users and computer experts untrained in these methodologies. The static model captured by a typical object oriented analysis is abstract. Users prefer to describe how the objects interact with one another rather than model them as forming relationships whose dynamism is to be implemented later by the member functions. Object-oriented methodologies recognize the importance of dynamic models and suggest the use of various finite-state machine models [4–6].

However, most methodologies fall short in describing how these models are derived and how they combine with the static model to complete the picture. Nor do they describe how these lifecycles are converted into methods during the design phase. There are some notable exceptions, such as the Shlaer-Mellor methodology [6]. This methodology relies on a specialized software development tool that is not always available to execute the model. We believe that for a methodology to have wider acceptability and usage it must lead to designs that can be implemented in widely available languages, such as C++.

For object-oriented languages to support the lifecycle models of the objects, they must provide mechanisms through which an object can traverse its lifecycle concurrently with the other objects in the system. Multithreaded execution is thus essential. A naive approach is to associate a thread of execution with each object, but this raises nontrivial problems relating to the synchronization

* Computer Science Department, University of Adelaide, SA 5005, Australia; e-mail: Charles.Lakos@adelaide.edu.au
** School of Computing, University of Tasmania, GPO Box 252-100, Hobart, Tasmania, 7001, Australia; e-mail: Vishv.Malhotra@utas.edu.au

Recommended by Dr. S. Rubin
(paper no. 205-3008)

and communication required by the concurrent progression over the lifecycles of different but cooperating objects. This article proposes a solution to this problem.

We describe a methodology that retains and implements lifecycles as it moves from analysis to design and finally to the programming stage. The methodology uses an event-based scheduling paradigm to enable concurrent execution of objects in a safe way in C++, a sequential language. The main contributions of this work can be summarized as follows:

1. It provides a methodology for extracting a life-cycle model together with a class model of the system from the textual description.
2. It provides a methodology for analyzing and developing the system specifications that leads to a complete and consistent specification. The methodology guides the developer towards those aspects that need to be addressed in achieving completeness and consistency.
3. It provides a method for translating specifications into programs in standard object-oriented languages, such as C++.
4. The complete methodology is illustrated with the help of a nontrivial case study.

The article is organized as follows. A nontrivial case study is described in Section 2. The fundamentals of the methodology are first raised in Section 3, and the methodology is applied to the case study in Section 4. This includes a technique for constructing object lifecycles from the problem description. Implementation-specific issues are discussed in Section 5. Having described our methodology, we compare it in more detail with the other established object-oriented methodologies in Section 6.

2. Case Study Problem

The purpose of this section is to describe a nontrivial problem that illustrates the methodology introduced in this work. The case study involves a system of lifts in a building. The versions of this problem have a long history of use in software engineering and formal specification studies for the reason that Schach [7] describes: "the problem is by no means as simple as it looks."

A building is serviced by several identical lifts. These lifts travel between the floors of the building and are controlled by a common controller. Each floor, with the exception of the ground and the top floor, has two call buttons, one for each direction of travel. For obvious reasons, the ground and the top floor have only one button.

Passengers arriving at a floor press the call button appropriate to their direction of travel. They wait in a first-in-first-out (fifo) queue for their turn to enter a lift. When a lift arrives and the doors open, the current passengers destined for this floor get out, and then as many waiting passengers as possible get in. These embarking passengers press appropriate buttons for their destinations. When the lift arrives at their destination, they get out.

Each call for a lift is registered by the controller. For each call of a lift, the controller determines if an idle lift is better placed than the currently active lifts to service the call. If this is the case, the idle lift is dispatched towards the calling floor. If several idle lifts are equally placed to service the call, one is chosen at random. Otherwise, an active lift will eventually arrive at the calling floor and will be able to service the call.

An idle lift continues to be inactive and stationary at a floor, with the door closed, until it is activated by the controller in response to a call from a newly arrived passenger. An idle lift, when activated, begins to travel towards the calling floor and serves the other passengers like other active lifts. An active lift continues to move upwards and downwards serving the waiting passengers until it finds no further remaining work. At this stage it becomes idle. A lift going upwards halts at various floors to drop passengers and to pick up new passengers. It does not change its direction of travel until it reaches a floor where it is no longer carrying a passenger, and where there are no waiting passengers on any floor above. The lift may change direction in order to service the passengers waiting to travel in the other direction. If this also yields no further work for the lift, the lift enters its idle state. An analogous behaviour is shown by a lift travelling downwards.

We assume that timing is significant for the problem solution. Thus, each lift travels at a fixed rate and needs some fixed delay to open and close its door. Likewise, the passengers take a fixed delay to enter and exit a lift. Each lift has as many destination buttons as there are floors in the building.

Our aim in modelling the system will be to model the problem as closely as possible to retain its association with the physical system.

3. The Fundamentals of the Methodology

An effective analysis and design methodology is distinguished by its ability to partition the problem into independent components. A measure of this independence is the high cohesion within the components and a weak coupling between them. In the absence of a methodology, the level of detail even in the above case study could be daunting. An object-oriented methodology partitions the problem by identifying the object classes in the problem domain. These classes exhibit a level of independence that places the object-oriented methodologies at the forefront of the system-structuring methods.

In the following presentation, we consider the fundamental issues concerning an object-oriented methodology. The identification of classes, associations, and attributes is assumed to follow one of the major methodologies. The dynamic issues, including the nature of lifecycles, is examined in more detail. The components of these lifecycles — states, events, and transitions — are considered in order, together with related issues such as state folding and the processing of events by functions. The methodology as a whole is considered in the following section.

3.1 Object Classes

The first step in the methodology is the identification of object classes. Objects that the problem domain experts identify or perceive define the classes of interest. A class specifies the attributes, which are represented by a set of data members (in C++ terminology). In each object, the appropriate data member holds the value of the associated attribute.

Each object has a distinct identity but shares a common behaviour with other objects of its class. The behaviour is captured by a set of member functions (in C++ terminology) defined by the class. Driven by the twin goals of reuse and encapsulation, many methodologies have been reported for object-oriented analysis, design, and programming (see, e.g., [4, 7, 8]). However, these methodologies fail to view objects as active entities executing their own lifecycles.

We shall adopt the convention of declaring all data members of a class to be private (or protected) and only accessing them through specially defined function members. This means that *all* of an object's behaviour is captured by its function members, and that the implementation of the classes is separated from their external interface. For our purposes, the class hierarchy of Fig. 1 captures the classes of interest to us in the lift system described in the case study. It uses the OMT notation of Rumbaugh et al. [8].

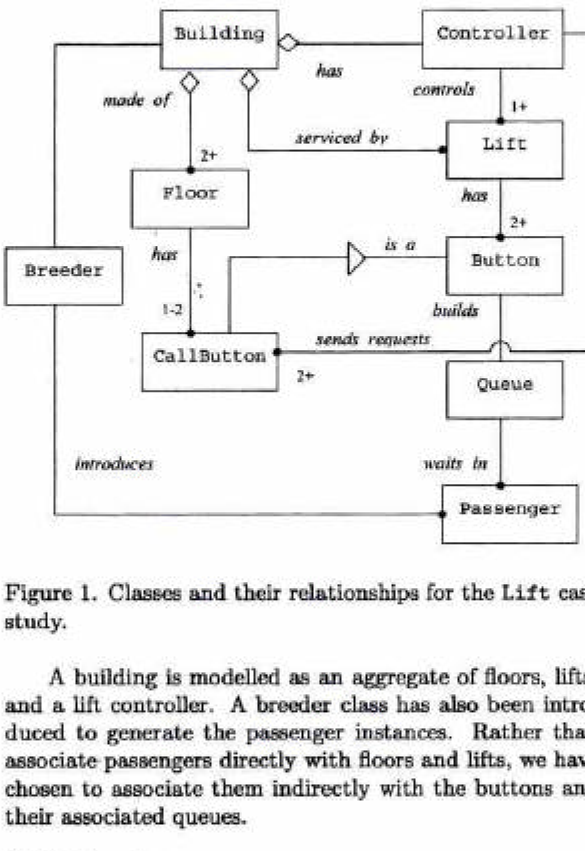

Figure 1. Classes and their relationships for the `Lift` case study.

A building is modelled as an aggregate of floors, lifts, and a lift controller. A breeder class has also been introduced to generate the passenger instances. Rather than associate passengers directly with floors and lifts, we have chosen to associate them indirectly with the buttons and their associated queues.

3.2 Lifecycles

As already noted, the attributes of classes capture only one facet of the problem domain objects, another important one being their behaviour. Most available methodologies rely on the function members of a class to provide this behaviour, which is exhibited by an object in its interaction with other objects in the system. The functions modelling the behaviour need to capture or encode this interaction. Without a disciplined methodology, this leads to an escalation of concerns and reduces the independence among the object classes. For example, in the case study, an object of class `Lift` needs to interface with an object of class `Passenger`. The two classes should have functions that implement the interactions in a consistent way. Further, each lift should be able to interface with any object of class `Passenger`. Likewise, each object of class `Passenger` should be able to interact with any one of the lifts in the system. Yet this interaction must be state dependent — a real lift interacts with a passenger only when it has stopped at a floor and its door is open. A simulated lift does so when its member function is called. The difference must be resolved by the designer.

The main problem with the above approach (which is encouraged by the C++ style function members) is that it focuses on atomic interactions rather than on the lifecycle of the object. Instead, we advocate that an object-oriented methodology start with the lifecycles and then derive the appropriate function members. Fortunately, it is possible to capture the lifecycle model of each class of objects with relative ease. Some guidelines on this are given in Section 4. We therefore exploit this ease to produce a lucid implementation of object behaviour.

The lifecycle of an object consists of two kinds of components: states and transitions, which we now consider in turn.

3.3 States

Objects tend to differ in the way they behave during their lifetimes. Some (simple) objects exhibit uniform behaviour and others have behaviour that changes markedly.

For objects of the former type, their lifecycles are uninteresting or trivial. In this case, we need only define a suitable set of function members that implement the object's interface to other objects. These objects are typically passive and serve as the repository of some information that is accessed and changed by other objects, through calls to appropriate function members. The model created for these objects can be verified by testing an isolated object. The exact sequence in which the functions are called is unimportant.

For objects of the latter type, their behaviour is determined by the state the object is in. The sequence in which the calls are made is usually important — the sequence of calls may have some bearing on the current state of the object. The models for such objects can only be verified by testing each object against plausible sequences of calls.

The external behaviour of an object is not the only determinant of the state of an object. Two objects of the same class may show identical behaviour in their current states but may have the potential of exhibiting different behaviour in the states that can be reached from their present states. For example, two lifts carrying passengers may behave identically. However, a damaged lift has potential of behaving in an unsafe way in a state where it is loaded to its full capacity. Shlaer et al. [6] suggest that the identification of states must be based on the internal details of the real-world objects. States based solely on the current behaviour of the object are usually not adequate.

A state can be represented in many ways:

- Explicitly using a data member of a suitable enumeration type. This representation is especially useful if the state also abstracts the behavioural history of the object.
- Implicitly, through a well-established protocol of interactions among objects. In other words, the state of this (or another) object is known by virtue of the preceding sequence of messages that have been exchanged.
- Implicitly as a set of values of its data members. This representation uses the current values of the object's data members for determining the state. Typically, the particular state is determined by an expression that depends on some combination of the attributes of the object.

These approaches can be combined with one another, as noted in the following subsection.

3.3.1 State Folding

If every possible combination of values of an object's data members were treated as a separate state, then the number of states would be unmanageable, as would the lifecycle of the object. Harel suggests the use of statecharts [9] to better organize the states of an object. However, a lifecycle description of an object can be made concise by retaining only the significant states of the object. This serves to partition the possible combinations of data member values. Alternatively, it can be considered to be achieved by *folding* a number of states onto one state. Consequently, there will be some data members that contribute to the differentiation of states, and other data members that will not.

For example, a lift may be in an idle state at any of the floors of a building. The floor at which the lift is idle may be of only secondary interest (as far as the lifecycle is concerned). The fact that it is in an idle state is of primary importance in modelling the lifts, and hence all idle states can be folded into the one idle state that appears as a single node in the lifecycle graph. Thus, the complexity and extent of the lifecycle graph can be reduced by coalescing all idle states. The current floor is still accessible through a data member.

Where some value of an object's data member only results in a different response in some situations, it is possible for the states to be further folded and for them to be differentiated in the appropriate context using a conditional expression. The above approach to state folding is founded on the folding that is fundamental to the Coloured Petri Net formalism for dynamic systems [10].

3.4 Events

An object progresses through its lifecycle in response to the reception of events. An event is a message received by the object, either from another object or originating within the current object. For example, when an embarking passenger presses a button for the destination floor, this will constitute an externally generated event for the lift. On the other hand, the event to signal arrival of the lift at a floor will have been originated by the lift (for itself) when leaving the previous floor.

Events therefore provide the mechanism by which objects are made aware of the asynchronous activities of other objects in the system. Events are identified by the object to which they are directed, an indication of the kind of event, and additional data, which may be considered to serve as parameters.

The response of an object to the reception of an event may be conditional on the current state of the object. In this way, an object can be guarded against unexpected events. An event may be handled immediately, or may be deferred for later response. We adopt the convention that only the object itself can defer or schedule an event for later response. This convention encourages better encapsulation of the classes.

To enable the objects to schedule and cancel event delivery we assume that the object-oriented language provides a suitable mechanism for this purpose. We will discuss an implementation in Section 5.

3.4.1 Functions and Event Handlers

From the above discussion, it will become apparent that an object will have two kinds of function members. One kind will respond to requests independent of the current state. These function members are used to access the data members of the object. We will refer to these function members simply as *functions*. Note that where a (simple) object has no state-dependent behaviour, all of its function members are functions (in this sense).

Another kind of function member will respond to events in a state-dependent way. We will refer to these function members as *event handlers*. Event handlers are distinguished (though not exclusively) by the fact that they do not return a result (or, in C++ terminology, return a result of type `void`).

Thus, an event delivered to an object causes a specified event handler to be called. There is one event handler for each type of event (rather than having one event handler for each type of event in each state). The event handler will thus have activity that is conditional on the state of the object when the event handler was activated. This poses little difficulty, as the state of the object before the delivery of the event can easily be found.

3.5 Transitions

A transition is the response of an object in a particular state to a given event. Every event delivered to an object causes it to perform a state transition. The performance of a transition causes the object to progress from one state to the next. The next state after firing a transition may coincide with the starting state of the transition. A state represents a period of time in the life of an object over which it interacts with the other objects in the system exclusively through function calls. On the other hand, a transition represents a point in time at which the object changes its state and interacts with other objects in a significant way as far as its lifecycle graph is concerned. A transition may:

- alter the values assigned to the current object's data members
- call functions of other objects in the system (to exchange data with them by accessing their data members)
- schedule or cancel the delivery of future events
- cause synchronous and/or asynchronous transitions in other objects (by passing events to them)

A transition is specified by a 5-tuple <`current_state, event, guard, actions, next_state`>. The operational semantics for the transition is described as follows: the transition is activated on receiving notification of the specified event `event`, provided that the object was in state `current_state` and the expression `guard` holds. The `guard` may be an expression based on the state of the object executing as well as the other objects in the system. The stipulated actions in `actions` are executed, and finally the object enters the new state `next_state`.

For example, a passenger object in the case study may undergo the transition of entering a lift if it is initially in the state of waiting for a lift, and if it receives the event (or invitation) from the lift to enter. The next state is that of entering the lift, and an action would be to schedule the completed entry event for some future time.

As must be clear, transitions represent threads of execution. The objects in the system can progress through their lifecycles concurrently by a suitable discipline of thread scheduling.

Note that a single event may be associated with a number of transitions in the lifecycle graph of the object. This requires that the actions associated with these transitions be implemented by the event-handling function.

4. The Methodology at Work

The software development process that we aim for is depicted in Fig. 2. The process begins with analysis. The aim of this phase is to collect data about the problem domain. The design phase organizes the data in a form that fits the needs of the software system to be developed. As a design emerges, completeness and consistency checks can be performed to identify the specific information that is lacking in the analysis document. These two phases repeat a number of times until the design can be completed, whereupon the implementation phase can begin. A utopian view is that a good design ensures that the earlier phases are not repeated after the implementation begins.

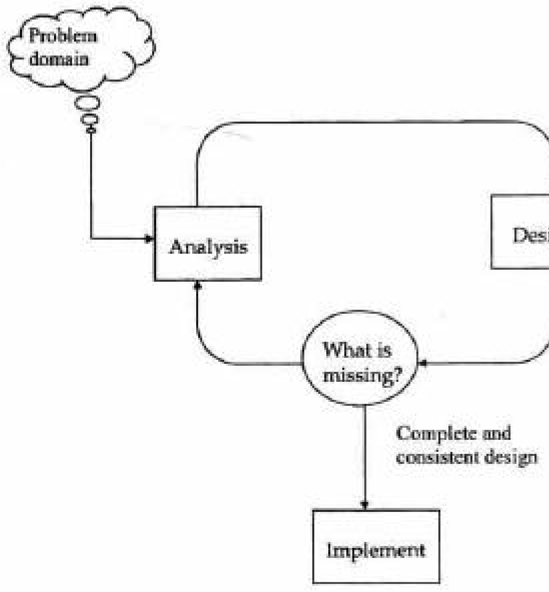

Figure 2. A software development process.

In what follows we give some guidelines to initiate the analysis-design cycle from the informal prose description of the problem.

4.1 Guidelines

A variety of techniques have been proposed for selecting the components of an information model — the classes, associations, attributes, and so on. Some have suggested that the identification of classes is straightforward [11]; some have suggested the use of certain categories such as tangible things, roles, and events interactions [6]; and others have suggested analysis of the problem description [8]. A significant advantage of the latter approach is that it forces the developer to work in the vocabulary of the problem space. It is also claimed that the approach has significant disadvantages — it is not a rigorous approach (given the imprecision of natural language), and it does not scale up well to larger systems. However, in the context of teaching students how to get started with object-oriented analysis, we have found the language analysis approach ideal. In this section, we extend the proposals of Rumbaugh et al. [8] to encompass the derivation of the lifecycle model as well.

At a fairly simple level, sentences are made up of nouns (which represent things), verbs (which represent actions), and qualifications of the above. In Rumbaugh's proposal, the nouns of the problem description are the prime candidates for classes, objects, and attributes. The analyst will need to exercise discretion in deciding whether a given noun falls into one of these categories or should be ignored. Rumbaugh uses the nouns as an initial list of classes and then provides a checklist that helps to identify those that are redundant, irrelevant or vague, and those that are attributes, operations, implementation constructs, and the like. Qualifications of nouns, such as adjectives, phrases, and clauses, may help to identify attributes or attribute values. Again, Rumbaugh provides a checklist in order to identify inappropriate classifications.

Although the above guidelines seem to be reasonably complete, and students seem to pick them up easily, the same is not true of the identification of associations and operations. We believe this is due to the inadequate attention given to the identification of operations in the context of lifecycles — their states, events, and transitions. Rumbaugh advocates the identification of stative verbs or verb phrases in the problem description as a guide to the associations. Again, a helpful checklist is provided to narrow down the list and exclude irrelevant or implementation associations, actions, derived associations, and so on. However, this approach is not continued for actions, let alone for lifecycles.

We recommend that the language analysis approach be extended to encompass some (at least) of the dynamics of a system. Verbs can indicate a continuous or an instantaneous action. Language textbooks variously refer to this distinction as linear versus punctiliar, or stative versus instantaneous. As with Rumbaugh, we recognize that stative verbs may indicate an association between classes. However, it is more precise to say that they indicate an association over a period of time. If that period of time is the lifetime of the system, or at least the lifetimes of the participating objects, then we simply have an association. If, however, the period of time is less than the lifetimes of the participating objects, then we have identified three states — the state before the association is established, the state while the association is in force, and the state after the association is dismantled. The analyst should then identify the events that cause the association to be established and dismantled, where they originate, and so on.

Where a verb is instantaneous, it suggests the presence of two states: before the action occurs, and after. It is quite likely that the action will be associated with an event, in which case the analyst should identify the kind of event, together with the sending and receiving objects.

4.2 Application to the Case Study

The choice of classes and associations is well known, so we do not pursue this aspect of the analysis in detail. An information model for the lift problem has already been given in Fig. 1.

We now turn to the dynamics of the system and the lifecycle model. We first consider the Passenger lifecycle at a level of detail that is possible given its simplicity. Later, we consider the more involved `Lift` lifecycle in less detail. We use the verbs to identify the associations, states, events, and actions. The states are numbered as in Fig. 3 and Table 1, and the events are related to the transitions in the same figures.

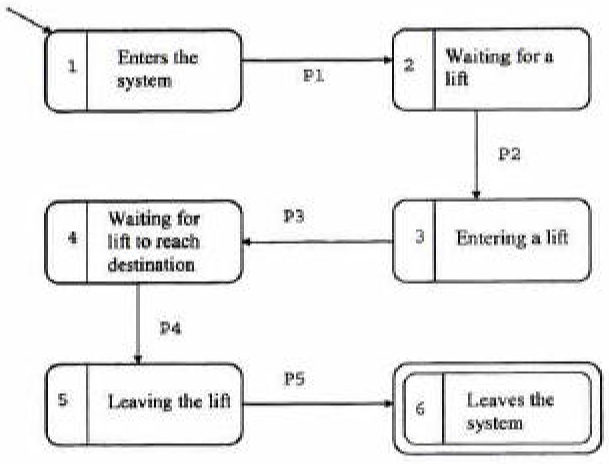

Figure 3. Lifecycle for Passenger objects. The transition details area available in Table 1.

The description states that a passenger *arrives* at a floor and *presses* the call button. If arrival is considered an instantaneous action, then it separates two states — that prior to arrival (when the passenger has been created) {state 1}, and that following arrival (when the passenger is waiting for a lift) {state 2}. The event that causes this change of state is the arrival (according to some predetermined probability distribution) {transition P1}. An event scheduler, described in Section 5, is responsible for the orderly delivery of such events. On receiving this event, the passenger will initiate the action of pressing the call button. Note that following this event, the passenger will be in an association with the floor, namely waiting at the floor for a lift.

The problem description goes on to say that the passenger can *get into* the lift once the lift has arrived, the doors have opened, and the passengers wishing to alight here have done so. The action of getting into the lift requires some time duration, and thus implies the need for three states — prior to boarding {state 2}, in the process of boarding {state 3}, and having boarded {state 4} — and two events — start boarding {transition P2} and stop boarding {transition P3}. The event to trigger start boarding will be sent to the passenger by the lift, and the event to stop boarding will be scheduled by the passenger for the appropriate time. The passenger will also initiate the pressing of the destination floor button in synchrony with the completion of boarding. It is interesting to note that while boarding, the passenger will have an association both with the arrival floor and with the lift — the start boarding event introduces the association with the lift, and the stop boarding event terminates the association with the floor.

Finally, the description states that the passenger will get out when the lift reaches the destination and its doors have opened. Again, the act of alighting requires some time duration and hence will identify three states — prior to alighting {state 4}, in the process of alighting {state 5}, and having alighted {state 6} — and two events — start

Table 1
Transition Descriptions for the Lifecycle Passenger Shown in Fig. 3

Transition	Start State	Event	Guard	Actions	Next State
P1	Enters the system	Passenger	True	Presses the call button appropriate for the intended direction of travel	Waiting for a lift
P2	Waiting for a lift	EnterLift	True	Schedule EnteredLift event to be delivered after necessary delay	Entering a lift
P3	Entering a lift	EnteredLift	True	Press the destination button of the lift and send PassengerIsIn event to the lift	Waiting for the lift to reach destination
P4	Waiting for the lift to reach destination	Destination _Reached	True	Schedule LeftLift event to be delivered after necessary delay	Leaving the lift
P5	Leaving the lift	LeftLift	True	Send PassengerGone event to the lift and Passenger	Leaves the system

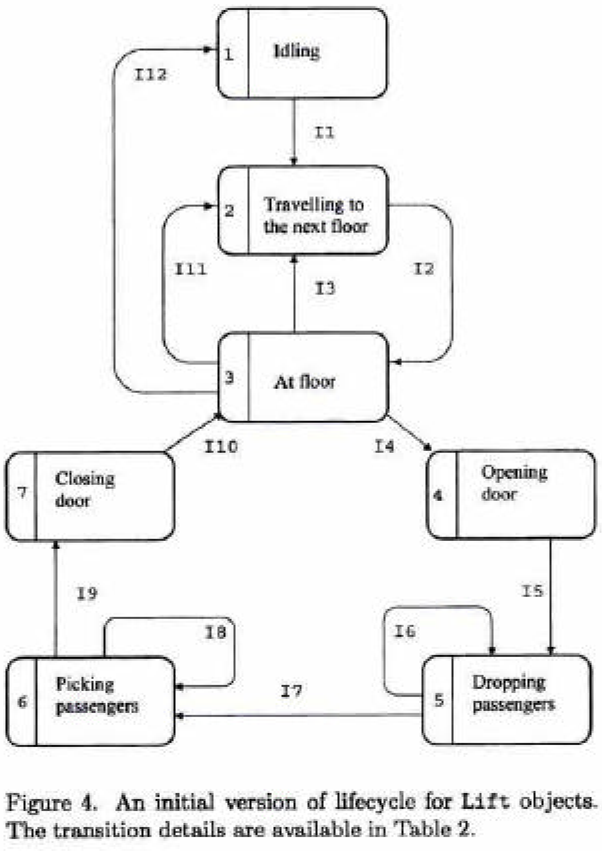

Figure 4. An initial version of lifecycle for Lift objects. The transition details are available in Table 2.

alighting {transition P4} and stop alighting {transition P5}. Again, the start alighting event will be sent to the passenger by the lift, and the stop alighting event will be scheduled by the passenger. The first event introduces the association of the passenger with the destination floor, and the second event terminates the association with the lift.

The above lifecycle was simple and linear, with the result that next to no effort was required to decide whether each identified state was a new one or a repetition of an earlier one. The lifecycle for Lift, which we now consider, is not that simple. For the sake of brevity, we simply annotate the original problem description, with extended comments to indicate contentious issues. The labelling of states and transitions corresponds with those of Fig. 4 and Table 2.

A building is serviced by {association} *several identical lifts. These lifts travel* {explains previous association} *between the floors of the building and are controlled by* {association} *a common controller. Each floor, with the exception of the ground and the top floor, has* {association} *two call buttons — one for each direction of the travel. For obvious reasons, the ground and the top floor have only one button.*

Passengers arriving at a floor press the call button appropriate to their direction of travel. They wait in a first-in-first-out (fifo) queue for their turn to enter a lift. When a lift arrives {transition I2 and state 3} *and the doors open* {state 4 and transition I5}, *the current passengers destined for this floor get out* {state

Table 2
Transition Descriptions for the Lifecycle Lift (Initial Version) Shown in Fig. 4

Transition	Start State	Event	Guard	Actions	Next State
I1	Idling	CallReceived	Call from a floor other than where the lift is idling	Determine the direction of the travel and schedule AtNextFloor event to be delivered after relevant delay	Travelling to the next floor
I2	Travelling to the next floor	AtNextFloor	True	Schedule ServiceFloor immediately	At floor
I3	At floor	ServiceFloor	No passenger to put down or pick up at this floor, or further to go	Schedule AtNextFloor event after relevant delay	Travelling to the next floor
I4	At floor	ServiceFloor	True	Schedule DoorIsOpen after relevant delay	Opening door
I5	Opening door	DoorIsOpen	True	Send the passenger a DestinationReached event	Dropping passengers
I6	Dropping passengers	PassengerGone	There is another passenger for this destination	Send the passenger a DestinationReached event	Dropping passengers
I7	Dropping passengers	PassengerGone	There are no more passengers for this destination and there is a passenger to get in at this floor	Send the passenger an EnterLift event	Picking passengers
I8	Picking passengers	PassengerIsIn	There is another passenger to get in and lift is not full	Send the passenger an EnterLift event	Picking passengers
I9	Picking passengers	PassengerIsIn	There are no more passengers to get in	Schedule DoorIsClosed event after appropriate delay	Closing door
I10	Closing door	DoorIsClosed	True	Schedule ServiceFloor immediately	At floor
I11	At floor	ServiceFloor	No more work in current direction, but there is work in the other direction	Determine the direction of the travel and schedule AtNextFloor event to be delivered after relevant delay	Travelling to next floor
I12	At floor	ServiceFloor	No more work in either direction	Empty	Idling

5 and transitions I6 and I7}, *and then as many waiting passengers as possible get in* {state 6 and transitions I8 and I9}. *These embarking passengers press* {the same transition I8 as we have already assumed that pressing the destination button is in synchrony with completion of boarding} *appropriate buttons for their destinations. When the lift arrives* {transition I2 as above} *at their destination, they get out* {state 5 and transitions I6 and I7 as above}.

Each call for a lift {event} *is registered by* {action} *the controller. For each call, the controller determines* {calculation action} *if an idle lift is better placed to service the call than the currently active lifts. If this is the case, the idle lift is dispatched* {transition I1 from state 1 to state 2} *towards the calling floor. If several idle lifts are equally placed to service the call, one is chosen* {calculation action} *at random. Otherwise, an active lift will eventually arrive* {transition I2 from state 2 to state 3} *at the calling floor and will be able to service the call.*

An idle lift continues to be {state 1} *inactive and stationary at a floor, with the door closed* {state 1} *until it is activated* {transition I1 and states 1 and 2 as above} *by the controller in response to a call from a newly arrived passenger. An idle lift when activated begins to travel* {state 2 as above} *towards the calling floor and serves* {multiple states and events} *the other passengers like other active lifts.* {The servicing of other passengers is decidedly vague and would require the analyst to investigate further. We assume that it implies that the lift visits one floor at a time on the way to the destination, and on arriving at each floor it makes a decision whether or not to stop. This yields transition I2 for arriving at a floor, transition I3 for moving immediately to the next floor because there is no-one to serve, and transition I4 to start opening the doors to service a passenger.} *An active lift continues to move* {state 2 as above} *upwards and downwards serving* {states and transitions as above} *the waiting passengers until it finds* {suggests an event} *no further remaining work.* {The previous sentence is again vague. We interpret it to mean that the lift determines it has no further work on arriving at a floor, not while it is in transit. This suggests state 3 for being at a floor and transitions I3 and I12 in response to the decision on whether it has or has not any further work.} *At this stage it becomes idle* {transition I12}. *A lift going upwards* {fold states for travelling up and travelling down into the one state 2} *halts* {transition I2 and I4} *at various floors to drop* {state 5 and transitions I6, I7} *passengers and to pick up* {state 6 and transitions I8, I9} *new passengers. It does not change* {transition I3, also foreshadows transition I11} *its direction of travel* {attribute not reflected in distinct states} *until it reaches* {transition I2} *a foor where it is* {suggests a state but it is folded into state 3} *no longer carrying a passenger, and where there are no waiting passengers on any floor above. The lift may change direction* {transition I11} *in order to service the passengers waiting in the other direction. If this also yields no further work for the lift, the lift enters* {transition I12} *its idle state. Analogous behaviour is shown by a lift travelling downwards* {suggests a folding of the state space}.

We assume that timing is significant for the problem solution. Thus, each lift travels at a fixed rate and needs some fixed delay to open {state 4} *and close* {state 7} *its door. Likewise, the passengers take a fixed delay to enter* {state 6 and transition I8} *and exit* {state 5 and transition I6} *a lift. Each lift has* {association} *as many destination buttons as there are floors in the building.*

There is some unstated common knowledge that goes with the above statement. This knowledge helps in relating states with one another. The lift needs to open its door for the passengers to enter or exit a lift. The passengers enter the lift one at a time. Likewise, they leave the lift one at a time. A lift must close its door before it can start traveling. A lift has a maximum capacity that determines the number of passengers it can carry.

The above analysis into states and events can be coalesced into lifecycle diagrams for `Passenger` and `Lift` respectively. It requires only a little more effort to identify the transitions in response to events and their associated guard conditions. This yields the complete `Passenger` lifecycle of Fig. 3 and the associated transition descriptions of Table 1.

The lifecycle for the lift, shown in Fig. 4 and Table 2, is still sparse and incomplete. Before proceeding further, we simplify the lifecycle by merging states 2 (`Traveling to the next floor`) and state 3 (`At floor`) into a single state. The simplification is justified by the fact that state 3 is a transitory state that waits for no external event — state 3 is mainly a computational convenience. The merged state will be called *In transit.*

4.3 Analyzing the Lifecycle Models

The tabular arrangement that we have used for specifying transition is motivated by our desire to be able to analyze the lifecycles. The analysis may reveal a gap in information or possible inconsistency. The analyst is thus guided to seek specific information to correct the situation. In the rest of this section we describe three checks that we have found useful in developing the complete lifecycle model for the lift: target identification, completeness analysis, and consistency analysis.

4.3.1 Target Identification

An event is posted to an object. It is therefore imperative that the guards clearly and uniquely identify the target objects for the events posted in the action part of the transition. The ambiguous nature of the textual descriptions and desire to avoid rare cases lead to the initial versions of lifecycles that fail to identify the target objects adequately in their transitions. For example, consider the transition I5 in Table 2. The lift is expected to send a `DestinationReached` event to the passengers destined for the current floor. Clearly, a guard for the transition is needed to ensure that a passenger who will be receiving this event exists. The need for further data gathering is obvious. A likely outcome of this data-gathering exercise will be the creation of more specific transitions to cater for the two cases where a waiting passenger exists and where no such passenger exists.

4.3.2 Completeness Analysis

The completeness check requires that for each triple of state **S**, event **E**, and system-wide predicate **P**, either it should be established that the triple cannot occur or there should be one or more transitions, $\mathbf{T_i} =< \mathbf{S}, \mathbf{E}, \mathbf{G_i}, -, - >$ where $i \in \mathbf{I}$ for some index set **I**, such that **P** logically implies $\mathbf{G} \cup i$. The check ensures that, within the constraints of the system (given by predicate **P**), every event can be processed whenever it is delivered to the object. In Fig. 4, which is an incomplete lifecycle graph for a lift, we notice that it is not possible to identify a transition out of the state 1 (`Idling`) on event `CallReceived` if the call is from the floor at which the lift is idling. The case study clearly is incomplete and the analyst needs to seek the necessary information in the application domain.

Another example concerns transitions I6 and I7 out of state 5 (`Dropping passengers`) that do not cover the case where all passengers for the floor have departed but there is no passenger waiting to enter the lift. On the other hand, it is not possible to process the `openDoor` event in state 2, but this is acceptable by virtue of a system constraint that states that a lift door cannot be opened while the lift is in motion.

It is worth noting that a nontrivial guard may arise from a number of sources. Such a guard may be used to differentiate the states of an object, whether in general or just in a particular lifecycle context. In this case, it would be possible, by unfolding the states, to remove the necessity for a guard (whether or not this style of definition suited the analyst). However, the guard can also be used to specify transitions that are conditional on the states of *other* objects. In this case, the same approach cannot (in general) be used to eliminate the nontrivial guards.

4.3.3 Consistency Analysis

Consistency analysis requires that the transitions be mutually exclusive. That is, at no stage should it be possible to take more than one transition in response to an arriving event. For example, a harmless case of inconsistency exists between transitions I3 and I4. An unambiguous lifecycle description requires that the two guards exclude each other.

A repeated application of these checks interspersed with acquisition of more data from the problem domain enables us to complete the lift lifecycle shown in Fig. 5 and Table 3.

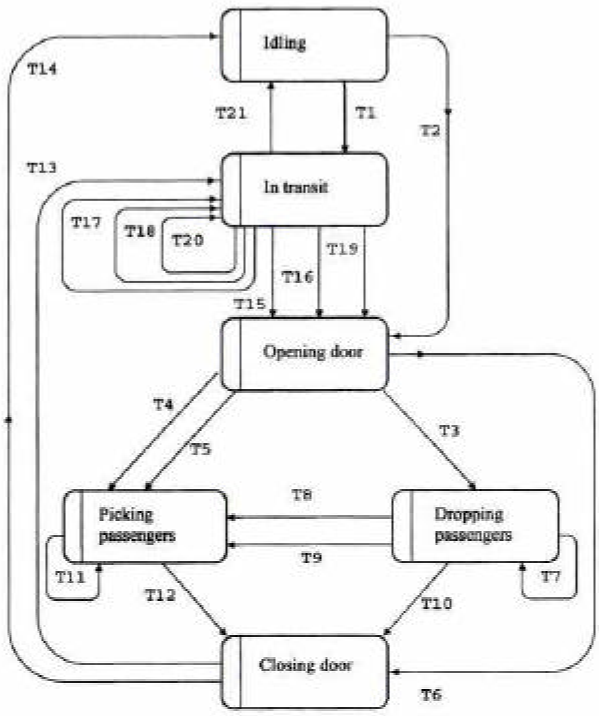

Figure 5. A complete lifecycle for `Lift` objects. The transition details are available in Table 3.

5. An Implementation Framework

The methodology we have proposed relies on an implementation framework that the commonly available programming languages do not directly provide. Fortunately, the flexibility of object-oriented programming (OOP) does provide ways to overcome this limitation, albeit at some cost to the elegance of the solution. The acceptance of the paradigm introduced in this work may provide some impetus for improvement in this direction.

A C++ implementation of the case study may be obtained from the authors, on an as-is basis.

5.1 Function Calls, Message Passing, and Deferred Calls

A function call is the mechanism for initiating execution of computation abstracted as a function. Object-oriented languages provide message passing as the mechanism for calling methods [12]. The object receiving the message determines the function to be executed. However, this mechanism, like that of the function call, is a synchronous operation. The computation is completed before control returns to the caller.

There are operational difficulties with the implementation of threads using function calls — the last-in, first-out (lifo) nature of this mechanism makes it awkward to enable concurrent progression over the threads in a number of objects. An unnecessary burden is placed on the designer and

Table 3
Transition Descriptions of a Complete Lifecycle for Lifts Shown in Fig. 5

Transition	Start State	Event	Guard	Actions	Next State
T1	Idling	CallReceived	Call from a floor other than where the lift is idling	Determine the direction of the travel and schedule AtNextFloor event to be delivered after necessary delay	In transit
T2	Idling	CallReceived	Call from the floor where the lift is idling	Schedule DoorIsOpen event to be delivered after necessary delay	Opening door
—	Any state other than Idling	CallReceived	True	Empty	No change
T3	Opening door	DoorIsOpen	Someone in the lift wants to get out on this floor	Send DestinationReached event to a passenger in the lift for this destination	Dropping passengers
T4	Opening door	DoorIsOpen	No one in the lift wants to go out here and the lift is not full and there is a waiting passenger on this floor to go in the current direction of lift's travel	Send EnterLift event to a waiting passenger going in the current direction of the lift's travel	Picking passengers
T5	Opening door	DoorIsOpen	No one left in the lift and no pending call requires lift to continue to travel in its current direction; there is, however, a passenger on this floor waiting to travel in the other direction	Reverse the direction of travel and send EnterLift event to a waiting passenger going in the new direction of the lift's travel	Picking passengers
T6	Opening door	DoorIsOpen	None of the guards in T3, T4, and T5 is true	Schedule DoorIsClosed event to be delivered after necessary delay	Closing door
T7	Dropping passengers	PassengerGone	Someone in the lift wants to get out on this floor	Send DestinationReached event to a passenger in the lift for this destination	Dropping passengers

T8	Dropping passengers	PassengerGone	No one in the lift wants to get out here and the lift is not full and there is a waiting passenger on this floor to go in the current direction of lift's travel	Send EnterLift event to a waiting passenger going in the current direction of the lift's travel	Picking passengers
T9	Dropping passengers	PassengerGone	No one left in the lift and no pending call requires lift to continue to travel in its current direction; there is, however, a passenger on this floor waiting to travel in the other direction	Reverse the direction of travel and send EnterLift event to a waiting passenger going in the new direction of the lift's travel	Picking passengers
T10	Dropping passengers	PassengerGone	None of the guards in T7, T8, and T9 is true	Schedule DoorIsClosed event to be delivered after necessary delay	Closing door
T11	Picking passengers	PassengerIsIn	The lift is not full and there is a waiting passenger on this floor to go in the current direction of the lift's travel	Send EnterLift event to a waiting passenger	Picking passengers
T12	Picking passengers	PassengerIsIn	The lift is full or there is no waiting passenger on this floor to go in the current direction of the lift's travel	Schedule DoorIsClosed event to be delivered after necessary delay	Closing door
T13	Closing door	DoorIsClosed	The lift is not empty or there is call from a passenger that the lift should attend	Determine the direction of the travel and schedule AtNextFloor event to be delivered after necessary delay	In transit
T14	Closing door	DoorIsClosed	The lift is empty and there is no call that the lift need attend	Empty	Idling
T15	In transit	AtNextFloor	Someone on this floor is waiting to travel in the direction of lift's travel	Schedule DoorIsOpen event to be delivered after necessary delay	Opening door
T16	In transit	AtNextFloor	Someone in the lift wants to get out on this floor	Schedule DoorIsOpen event to be delivered after necessary delay	Opening door

T17	In transit	AtNextFloor	The lift is not empty and no one in the lift wants to get out here and there is no waiting passenger on this floor to go in the current direction of lift's travel	Schedule AtNextFloor event to be delivered after necessary delay	In transit
T18	In transit	AtNextFloor	No one in the lift wants to get out here and there is no waiting passenger on this floor to go in the current direction of lift's travel and there are waiting passengers on the floors in the current direction of lift's travel	Schedule AtNextFloor event to be delivered after necessary delay	In transit
T19	In transit	AtNextFloor	Guards for T15, T16, T17, and T18 are all false and there is a waiting passenger on this floor to go in the direction opposite to the current direction of lift's travel	Reverse the direction of the travel and schedule DoorIsOpen event to be delivered after necessary delay	Opening door
T20	In transit	AtNextFloor	Guards for T15, T16, T17, T18, and T19 are all false and there are waiting passengers on the floors in the direction opposite to the current direction of lift's travel	Reverse the direction of the travel and schedule AtNextFloor event to be delivered after necessary delay	In transit
T21	In transit	AtNextFloor	Guards for T15, T16, T17, T18, T19, and T20 are all false	Empty	Idling

programmer to track these threads, and it requires specialized data structures as well as code that transfers the control among the threads in an equitable fashion. Concurrent execution of the threads using specialized libraries, such as lightweight processes (LWP) or system calls (see, e.g. [13]), presents its own difficulties.

There are also conceptual difficulties for the programmer, in identifying appropriate assertions, if threads are implemented using function calls. A long chain of calls, threading through virtually every object, would result if we were to use the function calls as a mechanism for progressing computation concurrently among the objects. Functional abstractions of the computation are predicated on certain stipulated preconditions holding before a function call. Method invocation by message-passing also requires the receiving object to satisfy an integrity constraint denoted by its invariant assertion. An object in the midst of executing a function does not guarantee to satisfy its invariant [11]. As a consequence, it is hazardous to send a message to an object when another function has not yet exited. This makes it impractical to use function calls to run concurrent objects.

In this article, we suggest the use of *deferred function calls* as a way to achieve concurrent progression along the multiple threads while at the same time avoiding the hazards that infect the other approaches. A deferred function call may be described as a call that returns immediately, leaving the execution of the function or the method pending. A pending call is selected for execution after the current thread has finished. Clearly, at this stage (prior to sending the deferred call) all functions have exited and, as a consequence, the stipulated invariants hold for every object. In this scheme, a program executes through a sequence of dynamically created threads — one for each deferred call. A thread, typically short, runs like the conventional object-oriented program by calling functions and sending messages. It should be noted that in this approach, the post conditions of certain functions may need to be weakened to indicate that the action has either been performed or been scheduled to be performed.

5.2 Deferred Calls as Events

Most commonly available languages do not provide a mechanism for deferred calls. However, they can be implemented in an object-oriented language by defining a set of classes for scheduling and delivering events. A deferred call is recorded by posting an event. The event is delivered after the completion of the current thread by the scheduler. The delivered event starts the execution of the intended function coded as an event handler.

The mechanism for posting and delivering events can be built in C++ in a number of ways. We describe two implementations.

In the first implementation, we use an abstract class `FSM` (Finite State Machine) with a virtual function `dispatch()` to receive the events and to call the associated event-handler functions for execution. A class containing event handlers uses the class `FSM` as its base class. In addition, the class needs to define an enumeration type `EventKind` listing all the events together with the definition for the function `dispatch()`. A typical `dispatch()` will be a switch selecting cases based on the events enumerated in type `EventKind`. To handle events with parameters, and also to improve encapsulation, the classes may provide specific member functions to post (schedule) events.

A second implementation consists of the base class FSM with a dummy set of virtual functions that the derived classes can populate with suitable function definitions. The scheduler accepts the events together with their parameters. The events are delivered by the scheduler by making a function call to the appropriate event handler function with parameters. We prefer this method, for it opens up an opportunity for inclusion among the language constructs. A library-based implementation is described in greater detail below.

5.2.1 Simulated Time

Simulated time can be handled using standard discrete-event simulation techniques [14]. The delayed calls can easily be extended to provide a simulated clock. Each event is posted with a scheduled time for its delivery. To be meaningful, the scheduled time cannot be earlier than the current time on the simulated clock. Events are delivered on nearest future event basis; on each delivery of an event the simulated clock is set forward to the delivery time of the event.

We have also found it necessary to make provision for cancelling scheduled events. In particular, it is desired to cancel the scheduled future events of an object as it is removed from the system.

5.2.2 Synchronous and Asynchronous Transitions

A transition in an object may be asynchronous — it does not require some other transitions to be happening simultaneously. Such transitions are easily implemented by posting new events. Alternatively, transitions may be synchronous — a transition may require that one or more other transitions occur simultaneously with the current transition. A practical model for these transitions is based on function call or message passing. The procedures implementing the synchronized transitions are executed prior to the procedure implementing the current transition being completed.

5.3 An Event Scheduler

The preceding discussions place an event scheduler at the heart of the object-oriented methodology introduced in this work. A suite of C++ classes has been developed to fill this role. The suite is modelled around the implementation of `softclock()` routine in 4.3 BSD Unix [15]. The balance of this section provides a brief description of the suite.

Class `FSM` is a base class that enables the derived classes to implement the state-specific behaviours. The class defines virtual event handlers that are overridden in the derived classes to implement the class-specific event handlers. All classes derived from the class `FSM` can use an object **EventScheduler** to schedule delivery of events to themselves. The object **EventScheduler** is a single object of class **Scheduler**. The objects in the derived classes of **FSM** post new events to themselves and cancel posted events by sending messages to **EventScheduler**.

An event is scheduled (posted) by calling the method **schedule()** with suitable arguments. The arguments include a pointer to (or address of) the object, the simulated time of the occurrence of the event, a pointer to the event handler function, and a pointer to a structure containing its parameters. The *event descriptor* returned by the call can be used to cancel the event before it has been delivered.

The other classes in the suite are **Time**, **Event**, and **EventArgs**. In addition to the singular object **EventScheduler**, described earlier, the suite has two other singular objects. These are **now**, giving the current simulated time, and **endOfWorld**, the final time on the simulation horizon. The class **Event** is exclusively used by **EventScheduler** to record events for future delivery.

```
//   Scheduler.h : Takes posted events and delivers them
#ifndef SCHEDULERH
#define SCHEDULERH

#include <classlib\dlistimp.h>
#include "Event.h"
#include "FSM.h"
#include "TSimDlg.h"

class Scheduler {
      private:
            TISDoubleListImp<Event> Remember;
      public:
            void EmitEvents(TSimulationDialog* display);
            Event* schedule(FSM*, long, FSM::Notice,
EventArgs* = noArg);
        void cancel(Event*);
};

extern Scheduler EventScheduler;
#endif
```

```
// FSM.h
// Base class for entities that wish to receive for events
#ifndef FSMH
#define FSMH

#define noArg 0

class EventArgs {};
class Scheduler;
class MyEvents;

class FSM {friend Scheduler;
      protected:
            MyEvents *MyEventList;
      public:
            typedef void(FSM::*Notice)(EventArgs *arg=noArg);
      // Virtual event handlers
            virtual void Action1(EventArgs *arg=noArg);
            virtual void Action2(EventArgs *arg=noArg);
            virtual void Action3(EventArgs *arg=noArg);
            virtual void Action4(EventArgs *arg=noArg);
            virtual void Action5(EventArgs *arg=noArg);
      // Constructors and Destructors
            FSM(void);
            virtual ~FSM(void);
      // Function members
            void CancelAllEvents(void);
};
#endif
```

Figure 6. Header files for Classes **Scheduler**, **FSM** and **EventArgs**.

The class **EventArgs** is a memberless class that is included in the suite to serve as the base class for arguments of the event handlers. The derived classes of **FSM** need to accompany a suitable set of the classes derived from the class **EventArgs**. The general prototype of an event handler in a class is: **void event_handler(EventArgs = noArg)**. It is frequently the case that certain events can be easily passed from the sender to the receiver object. These events need not be delivered through the (object) **EventScheduler**. The other may require the events to be posted. We prefer to encapsulate the functions to post events in the destination object's class. The program segments in Fig. 6 depict the header files for classes **Scheduler**, **FSM**, and **EventArgs** classes. The class **Breed that follows** provides an example of their use. The object receives an event **hatch**. In response to the delivered event, the object constructs a new passenger and schedules the next event after an appropriate random delay for constructing the following passenger. Note that the class has two **hatch** functions; one function provides an interface to other objects enabling them to post events, and the other function is the event handler.

```
// Breeder.h: Breed passengers

#ifndef BREEDH
#define BREEDH

#define hatch Action1

#include "FSM.h"

class HatchArgs: private EventArgs {
     public:
           long identifier;
           long averageDelay;
};

class Breed: public FSM {
     public:
     // Event posting functions
           int hatch(long, long);
     // Event handler functions
           void hatch(EventArgs *arg=noArg);
};

extern Breed Breeder;
#endif

// Breed.cpp: Generates passengers

#include "Breed.h"
#include "Passenger.h"
#include "Schedulr.h"

Breed Breeder;

int Breed::hatch(long id, long interArrivalDelay) {
// Posts a hatch event
      HatchArgs *arg = new HatchArgs;
      arg->identifier = id;
      arg->averageDelay = interArrivalDelay;
      EventScheduler.schedule(this, ComputeDelay(arg),
           &FSM::hatch, (EventArgs*) arg);
      return 1; // Posted successfully
}

void Breed::hatch(EventArgs* EventArgsP) {
// Event handler for event hatch
      HatchArgs* arg = (HatchArgs *) EventArgsP;
      new Passenger(arg->identifier++);
      EventScheduler.schedule(this, ComputeDelay(arg),
           &FSM::hatch, EventArgsP);
}
```

Figure 7. Header and code files for class **Breed**, which is used to generate new **Passenger** instances.

6. Discussion

Object-oriented analysis and design methodologies generally give careful attention to static modelling: the identification of classes, associations, and attributes. With few exceptions, they give only passing attention to dynamic modelling, the identification of states, transitions, and lifecycles. In the following discussion, we use the term *dynamic modelling* in the more general sense of capturing the dynamic behaviour of the system, and the term *object lifecycle* in the limited sense of identifying the dynamic behaviour of an object in a class.

Neither Henderson-Sellers nor Wirfs-Brock et al. address the structure or development of dynamic models [16, 17]. Some of the terminology of Wirfs-Brock et al. may suggest otherwise. For example, they talk about "contracts," by which they mean the list of requirements that a client object makes of a server object. The term "collaborations" is used for associations and "protocols" refers to signatures rather than any interaction pattern.

Rumbaugh et al. include a dynamic model in their methodology, based on Harel's statecharts [8, 9]. However, unlike the derivation of the static model, little in the way of guidelines is given for the derivation of the dynamic model. The dynamic model consists of multiple state diagrams, one state diagram for each class with important dynamic behaviour, and shows the pattern of activity for an entire system [8]. Scenarios and event traces are used to highlight the passing of events between objects. States determine the response to events and ignore attributes that do not affect behaviour. A (continuous or repetitive) action can be associated with a state (in line with Harel's approach). State transitions can be conditional, but it is not clear what those conditions may depend on: state components of the current object or other objects as well.

Rumbaugh et al. also provide for a functional model that indicates the transformation of data using dataflow diagrams. Such diagrams are *not* related to the object model and do *not* show the organization of values into objects. The leaf processes should be the operations on objects, and often there is a correspondence between the nesting of objects and processes. The approach of having a static model, a dynamic model, and a functional model does not result in a tightly integrated methodology; instead it provides three views of a system that later need to be integrated. Although some guidelines are given for achieving that integration, we believe that it is simpler and more effective to derive an integrated model from the start.

Booch gives an account of the dynamic model similar to that of Rumbaugh et al., and in fact states that his notation is that of Harel with Rumbaugh's extensions [4]. Again, the attention given to the dynamic model is significantly less than that given to the derivation of the static model.

Jacobson et al. [18] suggest a use case model to capture interaction among the objects. Given a specific scenario, all events and messages passed between the objects are captured in a single diagram. The focus is thus shifted from the objects to their uses in various cases. This is a big-picture view of the system rather than the nitty-gritty details of an object.

Waldén and Nerson take a rather different approach to object-oriented analysis and design [19]. For them, a key issue is the consistent use of object orientation in analysis, design, and implementation. Consequently, their BON method strives for a seamless and reversible approach: only efficiently implementable constructs are included in the notation, and the various models can be easily updated to match changes in the code. Another key aspect of the BON method is its thorough integration of the notion of software contracting (as originally proposed by Meyer [11]) in order to produce quality software products. It also carefully considers the importance of scalability in building large systems and therefore introduces the notion of class clusters and relationships between them.

With its emphasis on the seamlessness of a consistently object-oriented approach, the BON method avoids anything that may compromise this seamlessness and reversibility. Thus, the authors consider that functional modelling with data flow analysis should not be used at all in an object-oriented method, as the impedance mismatch and model confusion it introduces far outweigh any benefits gained. Similarly, the BON method avoids detailed consideration of dynamic models, because here too, they claim there is an impedance mismatch between FSM modelling and its state transition graphs, on the one hand, and the eventual implementation, on the other. They are critical of the use of event trace diagrams as they believe that these do not scale up for use with larger systems. They do allow the use of Harel statecharts to clarify the dynamic behaviour of selected classes, but consider that this should not be done in general for all classes. Such a loose recommendation indicates that the use of such statecharts is not integrated into the method. The BON method captures the dynamic behaviour of a system using *event charts* (which list the possible external events and the objects involved in responding to those events), *scenario charts* (which list possible interaction sequences), and *object creation charts* (which list which classes create instances of other classes). Then, for each scenario, dynamic diagrams specify the sequence of associated messages, and the objects connected by arcs labelled with those messages. In other words, the BON dynamic diagrams are like the object communication model of Shlaer-Mellor [6], except that they concentrate on one scenario at a time and also indicate the order in which the messages are sent.

In contrast to the above, we believe that the Shlaer-Mellor approach is unique in its detailed examination of the development of dynamic models in general and object lifecycles in particular, as evidenced by their producing an entire book on the subject [6], which carefully considers what constitutes a state, an event, and an action. Their work makes clear that states are differentiated by their effect on internal behaviour rather than on externally observable states; in other words, their view of states relates to white-box behaviour rather than black-box behaviour. Unlike Rumbaugh et al. and Booch, Shlaer and Mellor restrict their attention to simple state machine models, and not the hierarchical diagrams of Harel. (It might be argued that the segmentation of a system into classes makes the hierarchy constructs of Harel's statecharts superfluous.) The Shlaer-Mellor state machine diagrams are also simpler in including only unconditional state transitions — each state transition depends only on the current state and the received event, not on any condition that depends on other attribute values of the object or other objects. Given the demands of encapsulation, this effectively means that state transitions *cannot* be conditional on the current attribute values of *other* objects.

The Shlaer-Mellor approach continually emphasizes that the lifecycles are class based. Not only is each lifecycle a lifecycle for a particular class, but the object communication model (and the object access model) give the asynchronous (and synchronous) communication between the classes. This approach is also unique in addressing the issue of the dynamics of relationships. It recognizes that relationships may evolve over the lifetime of the system and that this evolution ought to be modelled. This could be considered as the transformation of a relationship into a class, but there are additional issues such as the handling of competitive relationship.

We believe that there are a few rough edges or loose ends in the Shlaer-Mellor approach. For example, the creation and destruction of objects is not handled explicitly; it is assumed that creation and destruction events are somehow handled by the runtime environment. The approach attaches actions as part of the state. The actions associated with a state are executed as the state is entered. This places some irritating restrictions on the number of parameters that a transition can carry into a state. Similarly, the issues of inheritance between lifecycles and the migration between the lifecycles of different subtypes have been hotly debated in the Shlaer-Mellor user group. These issues are still not resolvèd in the Specification of the Unified Modelling Language [20, 21]. However, they are relatively minor matters compared to the detailed consideration that is given to the whole matter of deriving dynamic models as part of the analysis process, and the integration of this with the static models.

Our approach to modelling the dynamic behaviour of an object system and the derivation of object lifecycles has been significantly influenced by the Shlaer-Mellor approach [6]. We have extended it in three significant directions. First, we have given guidelines (in line with the Rumbaugh approach) for using the language of the problem description to derive a first cut at the states, events, and actions of the dynamic model. Second, we have allowed conditional state transitions (which may be conditional on the attributes of both the object and other objects), but have given guidelines on how to decide whether these transitions are complete and consistent. Third, we have carefully considered how these lifecycle models can be mapped into existing object-oriented languages and how those languages should be modified to expedite the process even more.

7. Conclusion

In this article, we have developed a methodology that integrates software specification and validation. We have shown how the text of the problem description can be analyzed to produce a first cut at a dynamic model of the system. We have also presented guidelines that help to guide the developer to producing complete and consistent models.

We have argued that the function members of classes do not naturally capture all aspects of an object's behaviour. Instead, an object's lifecycle is often the more appropriate model, and this is the model derived by our methodology. Such object lifecycles can be implemented through the use of delayed function calls. This mechanism is in addition to the function call and message-passing mechanisms provided by current object-oriented languages. Although we believe that only a native implementation of the mechanism will provide a clean and elegant programming environment, we have suggested and implemented an event-based mechanism to emulate such an environment.

A case study of a lift simulator has been used to illustrate the complete methodology. We showed how the natural language textual descriptions could be analyzed to construct initial descriptions of the object classes. These classes were then analyzed to identify the nature of the missing and conflicting information. The resulting class definitions consist of data members, function members, and lifecycle graphs. The class definitions could then be coded into programs in a natural fashion.

Future work planned for this project includes the development of analysis tools for the completeness and consistency checks used in the problem analysis. Clearly, this requires a language for the formal statement of guards and actions in the transition descriptions. Another area of interest focuses on a preprocessor to isolate the programmer from the implementation details of delayed function calls pending their incorporation as standard object-oriented language extensions.

Acknowledgements

We thank Cristina Cifuentes for her suggestions and comments on this work and its presentation. We also thank the 1996 batch of students in KCA260: Software Engineering 2 and KCA315: Software Engineering 3 for what we have learned from their hesitations, concerns, successes, and failures. Thanks are also due to the anonymous referees who helped to improve the focus of this paper.

References

[1] W.W Royce, Managing the development of large software systems: Concepts and techniques, *Proc. of Wescon*, Los Angeles, CA, 1970, A/1-1–A/1-9.

[2] B.W. Boehm, A spiral model of software development and enhancement, *IEEE Computer, 21*(5), 1988, 61–72.

[3] M. Jackson, *System development* (Englewood Cliffs, NJ: Prentice-Hall, 1983).

[4] G. Booch (ed.), *Object-oriented analysis and design with applications*, 2nd ed. (Redwood City, CA: Benjamin/Cummings, 1994).

[5] M.P.E. Heimdahl & N.G. Levenson, Completeness and consistency in hierarchical state-based requirements, *IEEE Trans. on Software Engineering, 22*(6), 1996, 363–377.

[6] S. Shlaer & S.J. Mellor, *Object lifecycles: Modeling the world in states* (Englewood Cliffs, NJ: Yourdon Press, 1992).

[7] S.R. Schach, *Classical and object-oriented software engineering*, 3rd ed. (Chicago: Irwin, 1996).

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen *Object-oriented modeling and design* (Englewood Cliffs, NJ: Prentice-Hall, 1991).

[9] D. Harel, Statecharts: A visual formalism for complex systems, *Science of Computer Programming, 8*, 1987, 231–274.

[10] K. Jensen, *Coloured Petri nets: Basic concepts, analysis methods and practical use, Vol. 1: Basic concepts* (Berlin: Springer-Verlag, 1992).

[11] B. Meyer, *Object-oriented software construction* (Englewood Cliffs, NJ: Prentice Hall, 1988).

[12] S. Hirshfield & R.K. Ege, Object-oriented programming, *ACM Computing Surveys, 28*(1), 1996, 253–255.

[13] A. Silberschatz & P.B. Galvin, *Operating systems concepts*, 5th ed. (New York: John Wiley & Sons, 1997).

[14] R.M. Fujimoto, Parallel discrete event simulation, *Comm. ACM, 33*(10), 1990, 30–53.

[15] S.J. Leffler, M.K. McKusick, M.J. Karels, & J.S. Quarterman, *The design and implementation of the 4.3BSD UNIX operating system* (Reading, MA: Addison-Wesley, 1989).

[16] B. Henderson-Sellers, *A book of object-oriented knowledge: Object-oriented analysis, design, and implementation, a new approach to software engineering* (Englewood Cliffs, NJ: Prentice-Hall, 1991).

[17] R. Wirfs-Brock, B. Wilkerson, & L. Wiener, *Designing object-oriented software* (Englewood Cliffs, NJ: Prentice-Hall, 1990).

[18] I. Jacobson, M. Christerson, P. Jonsson, & G. Övergaard, *Object-oriented software engineering: A use case driven approach* (Reading, MA: Addison-Wesley, 1992).

[19] K. Waldén & J. Nerson, *Seamless object-oriented software architecture: Analysis and design of reliable systems* (Englewood Cliffs, NJ: Prentice-Hall, 1995).

[20] C.A. Lakos & G.A. Lewis, Behaviour inheritance for object lifecycles, *Proc. of European Conf. on the Technology of Object-Oriented Languages and Systems 33* (IEEE Computer Society, 2000), Los Alamitos, CA, 262–275.

[21] OMG, *OMG Unified modeling language specification*, Version 1.3 (Object Management Group, 1999), Needham, MA, USA.

Biography

Charles Lakos graduated from the University of Sydney with a B.Sc. (Hons) in 1972 and a Ph.D. in 1979, both in computer science. He has been interested in concurrency research since the late 1980s. His work has focused primarily on the augmentation of Petri net formalism with object-oriented extensions, thus providing powerful structuring capabilities. Object orientation has thus been an interest in both teaching and research for over 10 years. Recent work has considered the appropriate use of refinement in Petri net formalism, and this has had implications for the use of inheritance in UML statecharts. Dr. Lakos was employed at the University of Tasmania from 1976 to 1998, and is now on the staff of the Computer Science Department of Adelaide University.

Vishv Malhotra graduated from Birla Institute of Technology and Science, Pilani with a B.E.(Hons) in electronics. He received his Master's and Ph.D. degrees from the Indian Institute of Technology, Kanpur. He worked in the software industry for about a year before starting his teaching career with the Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, in 1981. Since then he has taught at a number of other institutions, including the Asian Institute of Technology, Bangkok; and Flinders University, Bedford Park, Adelaide. He has been with the University of Tasmania, Hobart, since 1991. His interests have spanned a number of different areas in computer science, including algorithms, compilers, programming languages and, more recently, in software engineering.