Formal Methods in the IS Domain: Introducing a Notation for Presenting Object-Z Specifications

Danielle C. Fowler
Paul A. Swatman
Evalyn Wafula

Formal Methods Research Group
Department of Information Systems
Swinburne University of Technology
Email: paul@swin.edu.au

March 20, 1995

Abstract

The evaluation of an information systems development method (Swatman & Swatman, 1992) synthesised from research into:

- the object oriented approach
- mathematically formal specification languages
- socio-organisational contextual analysis.

has led us to analyse the process by which models of the problem context (loosely, requirements specifications) developed under this approach are validated. Our research suggests that user *acceptors* find it useful to analyse the dynamics of interaction of objects within a system. In this paper, we describe a diagrammatic notation which we call 'event chains' by which such analysis may be facilitated. Following our development approach (Swatman & Swatman, 1992), we focus on and demonstrate the application of the event chain notation in conjunction with the object oriented formal specification language Object-Z. We argue, however, that the event chain notation and its underlying philosophy are valuable independently of the mathematical formalism.
We suggest a three-part approach to the presentation of specifications:

- a managerial overview
- a behavioural perspective (event chains formalised by means of Object-Z schema fragments—exemplified within this paper)
- a design perspective (Object-Z schemata embedded in explanatory text and supported by MOSES (Henderson-Sellers & Edwards, 1994) diagrams).

and argue that, although there is redundancy in the presentation of the formal specification across parts two and three, such redundancy (and, hence, the potential for conflict within the specification) could be controlled mechanically.

1 Introduction

Swatman & Swatman (1992) describe an outline information systems development methodology, widely applicable within the conventional information systems domain, which draws upon research into:

the object oriented approach in which situations are modelled as systems of interacting, encapsulated objects, each object belonging to some class

mathematically formal specification languages in particular, the object oriented specification language Object-Z (Duke *et al.*, 1991a) by means of which the abstract characteristics of classes may be described precisely and unambiguously

socio-organisational contextual analysis following the work of Checkland (1981, 1989) and Checkland & Scholes (1990) which, in the general case, denies the existence of an objective requirements specification waiting to be discovered by the systems analyst.

The methodology (Formal Object Oriented Method—FOOM) addresses the processes of information analysis, formal modelling, and debate and validation. Each of these activities forms part of a cycle which results in the iterative elicitation of the system requirements. In an action research (Susman & Evered, 1978) study undertaken at the Western Australian Government Department of State Services, we applied, evaluated and enhanced the methodology. In this paper, we

- discuss difficulties within the process by which models of the problem context (loosely, requirements specifications) developed under this approach were *validated*
- highlight the need, in the context of specification validation, for a mechanism to enable the illustration of dynamic aspects of system behaviour, then, as the major focus of this paper
- introduce and exemplify a diagrammatic notation designed to illustrate the interaction of component objects in implementing system behaviour.

One of the most significant difficulties associated with using formal specification techniques in the requirements analysis phase is that of encouraging participation and ensuring focus, clarity and precision in the debate and validation process. In order for users to participate in this process, they require a certain amount of training. In the case of users who are simply participating in the requirements determination process, where the flow of information is largely one-way (from the user to the specialist specifier), a brief introductory course of perhaps one day's duration would be sufficient. User *acceptors*[1], however (those users responsible for the acceptance of the requirements specification), have a much greater involvement in the process and, correspondingly, require more training (Swatman, 1993). Clearly the greater the understanding of the specification by the user acceptors, the greater the confidence which can placed in the validation. It is critical to this process, therefore, to present the specification in such a way as to facilitate the best possible understanding. Unfortunately, the "traditional" techniques used for the presentation of formal specifications are poorly suited to this task. Formal specifications are generally presented as schemata embedded within explanatory text—the structure of the explanation is determined by the structure of the specification (which is dictated by the

[1] A term coined by Olle *et al.* (1991) to describe the representative of client during the requirements specification validation process (*inter alia*)

syntax and suggested by class-based encapsulation) rather than by the behaviour of the system as understood by the users of the system.

Our initial solution was to supplement the mathematics and text with diagrams in a style drawn from the OO literature—we chose an early version of MOSES (Henderson-Sellers & Edwards, 1994; Henderson-Sellers *et al.*, 1992). MOSES, in common with most existing OO notations, concentrates on encapsulation, information hiding, the static relationships between classes or objects and the dynamic behaviour of "isolated" objects—thus providing a useful structuring mechanism for system designers and implementors (see Section 2.2 for a discussion of the overall structure of a specification and its potential types of audience), indeed, for the management of the design and implementation activities. Unfortunately, this view is not sufficient for the validation process because it fails adequately to reflect the way in which the users perceive the system—the notations focus on attributes and operations (in early notations, what the object *is*); more recently, also related behaviour (what the object *is responsible for*); but not how component objects interact to implement some system level behaviour.

We may explore the difference between these viewpoints by analogy, considering the example of a car. Following a traditional OO approach, we might describe:

- the car as a collection of sub-objects—such as an engine, a set of wheels and so on
- the behaviour of the car[2]. For example, we may say that the car responds to a "go faster" message—the input being pressure on the accelerator, the output being the new reading on the speedometer and the associated state change being a revision in the speed of the car object. It would also be legitimate to describe dynamic behaviour at a more primitive level, focusing, perhaps, on the behaviour of the accelerator pedal or the fuel injection system.

This presentation is clearly suitable for those who will design and build the car. Encapsulation and information hiding have been applied in a manner which corresponds with the cognitive models which the OO world has argued are appropriate for implementors. The presentation is also suitable for a prospective driver of the car who is protected from any need to "look under the bonnet". There is, however, an important interest which is not well served by the cognitive model which underlies the approach to encapsulation and information hiding adopted within the presentation discussed. The interest is that of the person responsible for licensing the car for use on our roads. The licensor is interested in the interplay of components within the car as it implements some "user level" behaviour. From this perspective, we want to show that on pressing the accelerator, fuel will be pumped into the carburettor and then passed into the engine, resulting (assuming the car is in gear) in an increase of speed. In other words, the licensor perceives the car to be *sets of related events* which combine to describe the (system level) behaviour of the car.

The analogue of the car licensor, in the world of information systems development, is the user acceptor. The notation reported in this paper extends MOSES by means of what we call 'event chains'. The approach allows us to take advantage of the benefits of the existing notation as well

[2]By means of a state transition approach, or a derivative technique, we may describe the way a car will change state, accept inputs and produce outputs in response to the occurrence of events. It is most common to organise the presentation of information about such dynamic behaviour around each class—that is, all possible dynamic behaviour of a car will be collected and presented together—although, more recently, Jacobson *et al.* (1992), for example, have suggested structuring the presentation of dynamic behaviour around the event which gives rise to the behaviour.

as providing a means of capturing the dynamic interaction between components which causes the observable behaviour of a system.

One notable difference in style between event chain notation and current MOSES is in the depiction of aggregation relationships. Within event chains, aggregation is shown directly by the encapsulation of the icon representing the component within the icon representing the "assembly". This representation style appears to be more effective in communicating the intended message to clients—especially those who are unfamiliar with the OO paradigm. Practical problems (of exploding complexity) associated with this representational style in models of the static aspects of systems appear to be of much less concern in event chains—indeed, our experience suggests that over-complex event chain diagrams are likely to indicate poor partitioning of the model (either of the event chain into links, or of the system into classes). A detailed discussion of representation options for aggregation relationships within OO specifications is beyond the scope of this paper. The issue is, however, a subject of our current research.

The event chain notation is used for the explanation of requirements (which are expressed in Object-Z thus forming the definitive specification) to the user acceptors during the specification validation process. Its development arose from the absence of an existing notation which could be used to present system behaviour in a manner suitable for validation by a user acceptor.

Existing semi-formal OO notations concentrate, understandably enough, on encapsulation, information hiding and the static relationships between classes or objects—inheritance, aggregation and association. Most of these notations[3] are geared purely toward design, and many support only static representation (eg Wirfs-Brock *et al.* (1990)): that is the structural aspects of objects, their relationships and attributes (Monarchi & Puhr, 1992). Whilst this approach is appropriate for the constructors of a software system, the car analogy given above shows that it provides an inadequate mechanism for the user validation of specifications.

Of the OO approaches which include some form of dynamic modeling, most make no attempt to model the behaviour of the entire system, but rather provide piecemeal modeling using techniques such as state transition diagrams for each object (Fichman & Kemerer, 1992). An example of this is the notation of Booch (1991), which supports dynamic representation through state and timing diagrams[4]. This focus on modeling the behaviour of components of the system, rather than the behaviour of the system as a whole means that system-level behaviour cannot easily be conveyed.

We also considered non-OO approaches which held promise, in particular Petri nets (Peterson, 1977, 1981) and Statecharts (Harel, 1987), both of which have OO extensions: OPNets (Lee & Park, 1993) and ObjectCharts (Coleman *et al.*, 1992). These notations proved to be unsuitable for the same reason many of the notations we examined were unsuitable: they focus on formalising the properties of a system in some detailed manner (ultimately, one which is useful to system constructors). Consequently, when dealing with complex systems they become too complicated and inflexible as explanatory mechanisms. What we required was not a second formalism to represent a system's behaviour, but rather a less formal, flexible notation which could be used as a roadmap to guide a user acceptor through the formal definition of the system contained in the Object-Z specification.

The notations which offered the most promise were those OO notations which structure the presentation of the system from the user's point of view. These include "scenarios" (Rumbaugh *et al.*, 1991) and "use cases" (Jacobson *et al.*, 1992), which are also used by Lorenz (1993),

[3]This comment relates to notations, NOT methods

[4]In later work, Booch (1994) replaced State Transition Diagrams with Statecharts; and Timing Diagrams with Interaction Diagrams and Scripts.

Booch (1994) and Henderson-Sellers & Edwards (1994). Briefly, use cases and scenarios describe a behaviourally related sequence of transactions which a user of the system (either a human or another system) performs in dialogue with the system. Our research supports the assertion made by the various authors of these notations: that describing the behaviour of a system in this manner facilitates a better understanding of the system by its user acceptors, permitting a more successful validation of the system requirements.

Although the use case/scenario approach seemed to be advantageous, these particular implementations were inadequate for our needs in certain areas. In each case the notation was either non-graphical (or minimally so), making understanding on the part of the user acceptors more difficult, or they were too simplistic to enable the successful explanation of complex chains of related behaviour. In many IS applications use cases or scenarios would work well—for example transaction processing systems, where the user executes some action and some well defined change of state occurs—together, perhaps with some response. There are, however, more complex situations where the result of the user transaction is a changed state within which a concatenation of further state changes may occur without further interaction from the user. It is, of course, always possible to encapsulate the chain of events leading to the next logical state of equilibrium, but doing so would (at best) *suggest* that the chain of events was hard coupled. In the case of the specification which illustrates our notation in this paper, we specifically did *not* want to specify the ordering of operations resulting from many users of the system—that is we wanted to avoid unnecessary determinism within the specification.

2 Event Chains

We required a notation which would:

- describe the behaviour of the system from the point of view of a user of the system, whether that user is human or another system. The better the understanding of the system specification by its users, the more successful the validation of requirements will be. A dynamic representation technique was therefore required
- be simple but expressive—this point is related to the one above. A second formal notation is not required—the formality is captured by the Object-Z; the aim is more clearly to convey behaviour of the system to the users. Given this requirement, some kind of graphical representation was therefore desirable
- support and describe an OO specification
- be able to handle specifications of any complexity. Some sort of hierarchical decomposition or abstraction mechanism was therefore required.

The event chain notation which we have devised is useful for systems with distributed or notionally parallel aspects, rather than for traditional transaction processing systems (which is where existing OO notations seem to have originated), though it may also be used successfully in these simpler cases. The notation is made up of three components: event chain overview diagrams, link diagrams and an event map:

- each 'event chain' is made up of a set of events which combine to describe related behaviour of the system as seen by its users. In this respect they are very similar to use cases or scenarios. The event chain overview diagram shows the whole chain of events at its highest level of abstraction

- each chain is broken down into 'links'—each link diagram describes pictorially one event in the event chain. Link diagrams can be thought of as *icons*, which serve a dual purpose. Each link *represents* a state transition, and it also *illustrates* the associations between classes involved with the change of state

- lastly, the 'event map' shows the relationship between the 'links' in an event chain. Depending on the results of the permissible state transitions, different links combine to form the path which makes up the event chain. In effect, the event map shows the possible combinations of links which make up the event chain. As the links in an event chain may be quite decoupled, the event chain map provides a structuring mechanism which clearly defines the end, or ends, of the event chain.

A number of well-known diagramming techniques may be used to present event maps—we use state transition diagrams which may easily be used to show the relationship between the various links (each link is one of the lines connecting states). Other notations, such as Jackson Structure diagrams (Jackson, 1983) can also be used to construct event maps if they appear more suitable in a given situation—for instance, if they are already familiar to the users of the system.

2.1 Notation Components

Event chain overview diagrams and link diagrams are constructed from a set of components (summarised in Figure 1):

Class boxes Classes within a specification are represented by dome topped rectangular boxes or "tablets" with the class name contained within the dome. Class boxes within other class boxes indicate aggregation, ie the surrounding class "contains" the interior one.

Operation names Operation names appear within the class box of the class to which they belong. Each operation name is marked with a preceding dot.

Arrows Arrows indicate the transfer of information from one object/class to another. The output may itself be any abstract date type (ADT) used within the specification, most typically an object or the attribute(s) of an object. The arrows extend from the operation producing the output (if applicable) to the operation receiving it (if applicable). Arrows are always labelled. The name of the information being transferred is listed along the line. If the output is an attribute (or attributes), rather than an object or other ADT, then the name (names) appear in parentheses

Solid lines Solid lines are drawn from a triggering operation name to a triggered operation name within an encapsulated object. Solid lines are not labelled.

Dashed lines Dotted lines are drawn from a (high-level) operation name to two or more operations within the same or, more usually, encapsulated class boxes. The concept being illustrated here, is that the high-level operation represents the combined effect of the subsidiary operations occurring cooperatively and in parallel.

Grey lines, class boxes and arrows Occasionally when several operations are tightly coupled there will be consequences of these operations (outputs generated, for example) not relevant to the link diagram in question— Figures 7 and 8 in the following example illustrate this idea. Rather than leave out these parts of the diagram, which would obscure understanding of the way in which the operations involved interact, irrelevant sections are distinguished by grey, less pronounced components (ie class box outlines, arrows and

lines). Any lines involved are presented in a paler type than the rest of the diagram, as are any operation names involved.

Event chain overview diagrams are simply high level link diagrams which show the transfer of information between classes at the highest level. Consequently, the only notation components involved in these diagrams will be class boxes and arrows showing the transfer of information. Link diagrams may contain any or all of the notation components.

There are two conventions associated with the composition of link diagrams:

- It is a common feature of object oriented models that an operation on a complex object (one with many layers of nested components) is a simple consequence of an operation occurring on a component object or, more generally, on a component of a component of a component ...to any nested depth. In line with Object-Z, we call this concept "promoted operations". Intermediate layers of promoted operations are not shown in link diagrams (though the nesting of components is illustrated). The class level at which the operation performs its function is the one where the operation name appears. Any arrows or lines (solid or dotted) involved with the operation point directly to the operation name within the class box, or extend directly from it. Figure 5 in the following example illustrates this convention.

- sets of objects (of the same class) are indicated by a class box with a shadow (to indicate depth). In situations where it is necessary to distinguish between the members of the set, the class box is shown as many times as is required to illustrate the effect of the link. Lines of hollow dots link the various instances to indicate the potential existence of other objects of the class. Figure 7 in the following example illustrates this concept

An example of the notation applied to a sample specification, which more clearly illustrates how the notation may be used in practice, appears in the following section.

2.2 Specification Structure

In terms of the overall structure of a specification's documentation, the notation is situated within the section aimed at those users of the system by whom the requirements will be validated. Our research suggests that the following structure is appropriate for the documentation of a specification:

- the specification should begin with a high level overview of the functionality of the system, supported by free form diagrams. This section is suitable for presenting to high level management, and possibly some user participants, with an interest in the system but not in the detail of the specification. This section may be considered an "executive summary"

- the second section, for which the notation we are presenting was required, presents the system in such a way as to enable the validation of requirements elicited via the user acceptors. Appropriate parts of the Object-Z specification (often, relevant fragments of class definitions drawn from the complete specification contained in the third part of the specification) are presented as required, to show how the specification formally models the behaviour of the system as presented in the event chain and link diagrams. The same class schemata or operations may well be duplicated across various event chains, or even

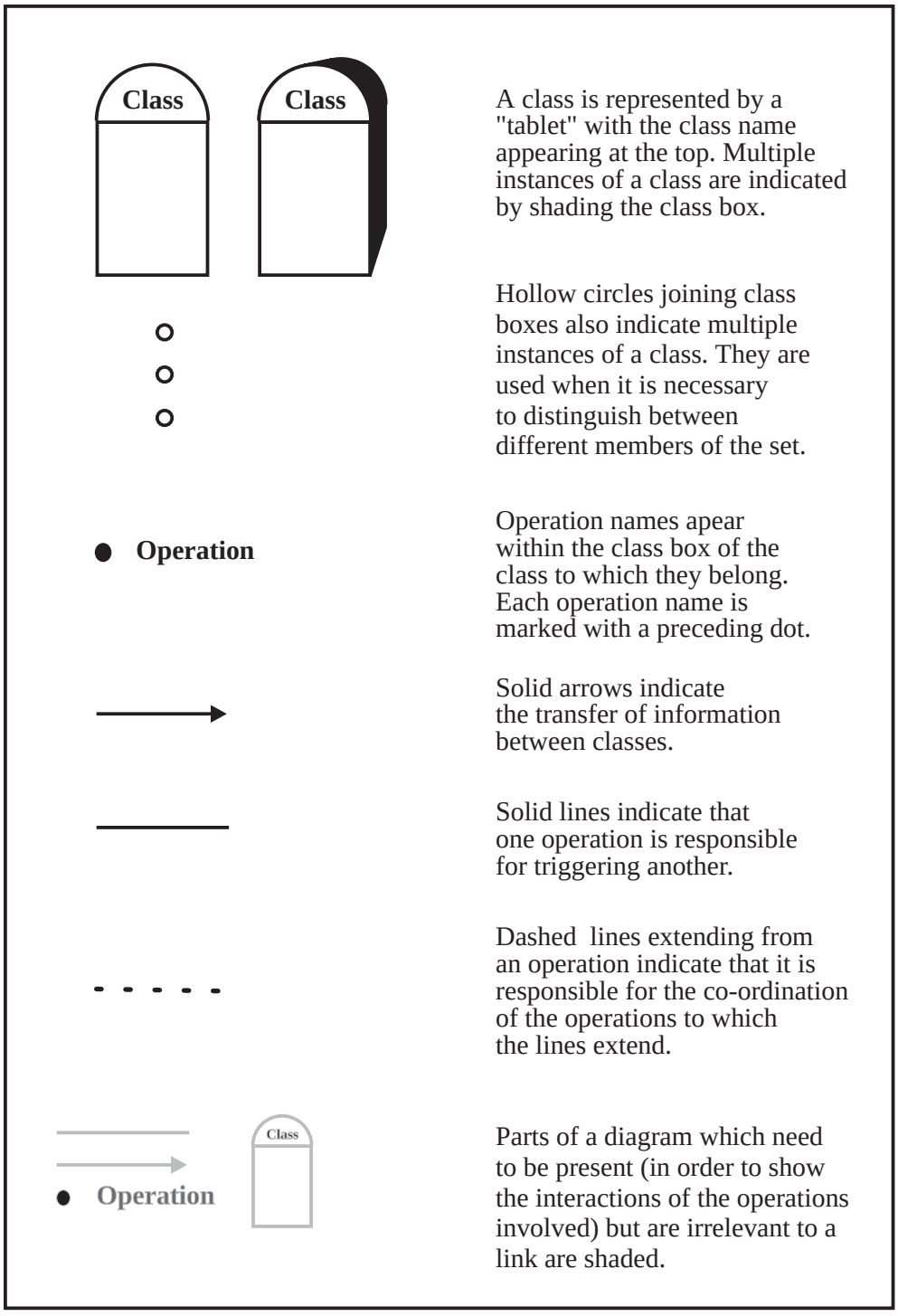


Figure 1: Components of the Event Chain Notation

within the same event chain[5]

- the last section contains the entire Object-Z specification, documented in the traditional way. It also contains traditional OO diagrams (showing aggregation, association, inheritance etc) to supplement the Object-Z and text. The section is aimed at designers and implementors who are already familiar with OO and Object-Z.

3 Example: A Simulation of an Operating System

We now demonstrate, by means of an example, how the event chain notation may be used, in conjunction with a object-oriented formal specification to present a requirements model for validation. Although, as we mention above, the event chain approach has been validated within the Information Systems domain through an action research study undertaken at the Western Australian Government Department of State Services, we choose to illustrate the use of event chains in this paper by means of a "toy" example, that of a simplified operating system. We do this since the subject matter of this example is likely to be familiar to most readers and, consequently, to allow readers to place themselves in a position analogous to that of a client attempting to validate a specification.

3.1 An Informal Statement of Requirements

The operating system exhibits the following characteristics:

- the underlying hardware is comprised of multiple, independent processors
- users may request services of the operating system at any time
- user requests fall into one of two general classes: 'online' and 'batch'
- requests from online users are given priority over those from batch users
- in the event of conflict between user requests of the same class, priority 1 requests are preferred over priority 2 requests
- priority 1 requests from online users may preempt resources from batch requests
- priority 1 requests require 3 nanoseconds of processor time, priority 2 requests require access to a processor for 4 nanoseconds
- each user is billed for each nanosecond of processor usage (3 dollars per nanosecond for online users, 2 dollars per nanosecond for batch)
- requests preempting a processor are charged a fixed service fee (2 dollars); preempted requests are completed for no charge

In the first section of the specification - the "executive summary" - we would describe the overall structure of the underlying requirements model which we illustrate, informally, in Figure 2. Briefly, the whole information system (labelled "Computerised System") is comprised of a set

[5]We believe consistency can be controlled by extraction from a single source of the formal specification. Research and development is currently ongoing into an analyst's workbench to support the methodology, and notation/documentation standards

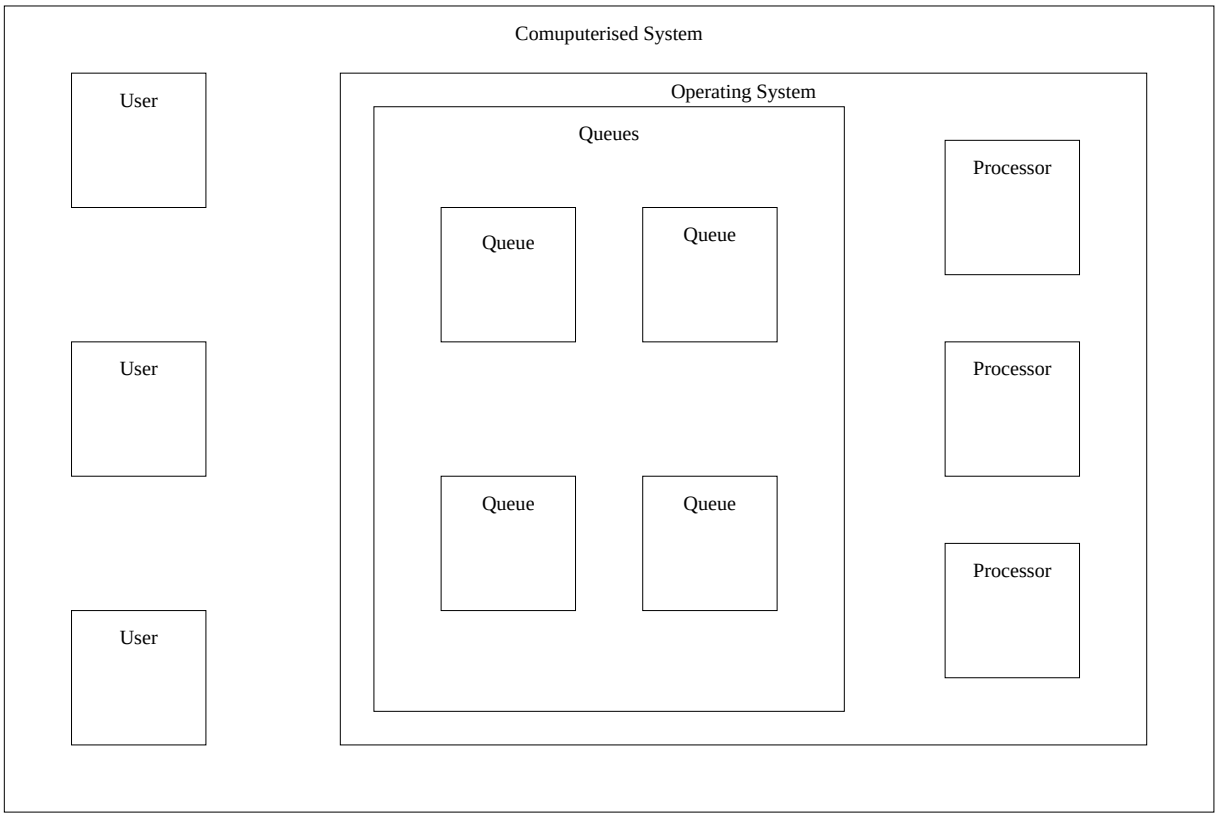


Figure 2: Components of an Operating System

of "Users" and an "Operating System". Encapsulated within the operating system is a set of processors and a bank of queues (labelled "Queues") which itself encapsulates a set of four queues (each labelled "Queue"). Requests (not shown in the figure) are passed between objects within the system. For example, a User might pass a request to the Operating System which would pass the request to Queues which would direct the request to the appropriate Queue where, for example, it may be stored to await an idle processor.

In the second part of the specification we present the event chains which represent, at a detailed level, the behaviour of the system as seen by the users of the system, whether they be human users or other systems. In the case of the operating system specification there are several event chains which together model the behaviour of the system. For the purposes of this paper we will consider just one, the case where a user of the system sends a request into the operating system for processing. In this presentation, we show how the diagrams supplemented by explanatory text are used to give insight into the formal, Object-Z specification, relevant fragments of which are included. The complete Object-Z specification would be presented in the third part of the specification document, though, for brevity, this has been omitted from this paper[6]. For readers unfamiliar with the Object-Z language, Appendix A contains a brief tutorial.

3.2 Introducing Validation-Oriented Formality

The event chain we will examine begins with the user sending a request into the operating system, and ends when the request has been processed. The interaction of classes at the highest

[6] A complete specification document for this example is available from the authors in the form of a technical report.

level is shown in Figure 3, the event chain overview diagram. At this highest level the only classes involved are the *User* and the *Opsys* and the only information being transferred is the *Request* being passed from one class to the other.

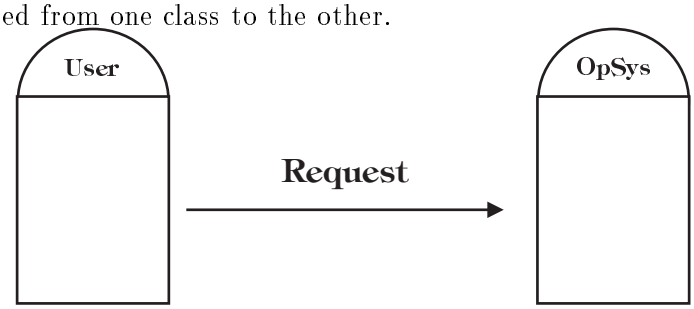


Figure 3: Submission of a Request: Overview.

This event chain as a whole is made up of a number of discrete steps, or *links*, which may occur either in sequence or in parallel. As mentioned previously, alternative branches in the connection of links are possible, allowing selection between different paths through the chain. The links which make up this particular event chain are:

link 1; the user sends the request into the operating system, where it is added to a queue

link 2a: when a processor is free, the request is passed from the queue to the processor

link 2b: alternatively, if the request is an online, priority 1 request and there are no free processors, it may preempt a batch request

link 3: batch requests may be preempted. If preempted, the request is returned to a queue, and the chain loops back to a state in which link 2a is enabled (i.e. a state where the request may be passed from the queue to an idle processor)

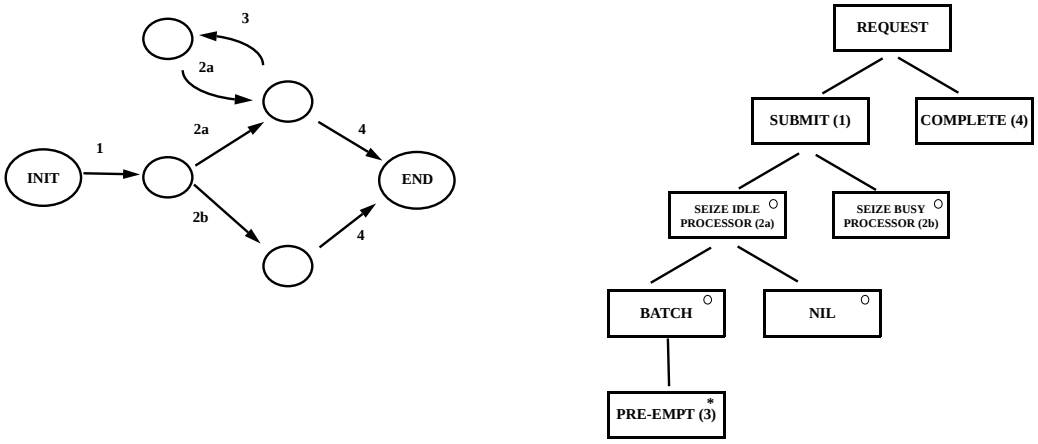
link 4: the request is complete, and the charge to the user is calculated.

The event chain map depicted in Figures 4(a) and 4(b) shows the relationships between these links. The two figures show exactly the same event map, but in different formats—in Figure 4(a) the event map is constructed as a state transition diagram, while Figure 4(b) presents the event map as a Jackson Structure diagram (Jackson, 1983). We will now examine the links which make up our chosen event chain more thoroughly.

Link 1: The Request Is Submitted

The first link in the chain, represented diagrammatically in Figure 5, involves a number of classes. A user (an object of class *User*) sends a request to the operating system (an object of class *OpSys*). The operating system stores the request in a queue (an object of class *Queue*) which is itself contained within a managing object *waiting* of class *Queues*. It is at this point that Object-Z is introduced to define the link. The first class we will consider is the *Request*, which is defined below. A *Request* has a number of pieces of information associated with it:

- the UserId of the user sending the request (*uid*)
- the priority (*pri*) of the request, i.e. priority 1 or priority 2
- the *type* of the request, i.e. *ONLINE* or *BATCH*; and



(a) State Transition Diagram

(b) Jackson Structure Chart

Figure 4: Event Chain Maps

- whether the request has been *preempted*.

The attributes of a *Request* which are actually relevant to this link in the event chain are its *uid*, *pri* and *type*. As the other attribute (*preempted*) is relevant to the preceeding links in the event chain however, the entire state schema for a *Request* is presented here. The initialisation operation reflects the fact that a request can not have been preempted when first created.

Request

$$uid : UserId$$
$$pri : PRIORITY$$
$$type : LoginType$$
$$preempted : \mathbb{B}$$

INIT

$$\neg\ preempted$$

Having introduced *Request*s, the next class we will examine is the *User* class. The only piece of information we need to know about a *User* for this event chain is that each user will have a constant UserId (*uid*) associated with it. The *User* sends out a request via the *Send* operation (which ensures that the *Request* which is sent is initialised, and the *uid* of the *Request* is equal to the *uid* of the *User*), as follows:

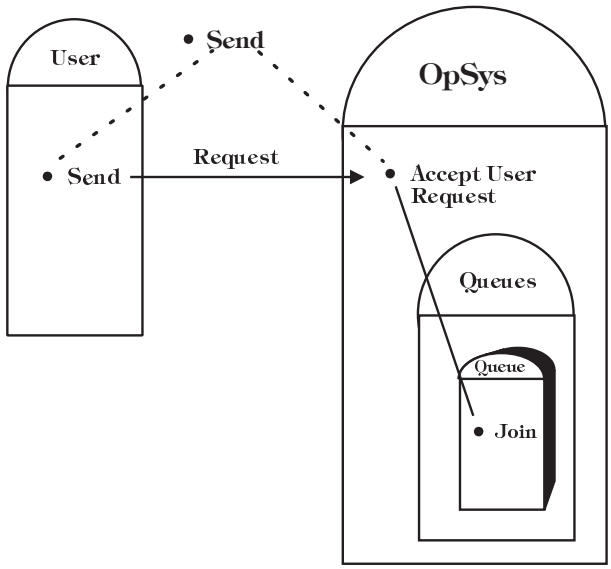


Figure 5: Link 1

User

$uid : UserId$

Send

$r! : Request$

$r!.init$
$r!.uid = uid$

Once the *Request* has been sent out by the user it is received by the *OpSys*, via the *Accept User Request* operation. There a number of attributes in the state schema of an *OpSys* which are used by this operation:

- *waiting* is of type *Queues*, the class which manages each *Queue* in the system
- *online* is the set of identifiers of *Users* which are logged in as online users
- *batch* is the set of identifiers of *Users* which are logged in as batch users

OPSys

$waiting : Queues$
$online : \mathbb{P}\, UserId$
$batch : \mathbb{P}\, UserId$

AcceptUserRequest

$\Delta(waiting)$
$r? : Request$

$(r?.uid \in online \land r?.type = ONLINE) \lor$
$(r?.uid \in batch \land r?.type = BATCH)$
$waiting.Join$

The request received by the *Accept User Request* operation is checked to ensure that its *uid* matches that of a *User* logged on to the system (either as an *online* user or a *batch* user), and that the type of the request matches this user group, i.e. if the *uid* of the request matches that of an *online* user, then the *type* of the request must be *ONLINE*. Assuming these conditions are satisfied, then the *Accept User Request* operation triggers the *Join* operation within *waiting* (of class *Queues*, which consists of a set of *Queue*). As the *Queues.Join* operation is simply a promotion of the *Join* operation defined within the *Queue* class it is not shown on the link diagram. The state schema of *Queues* is relevant however, as it ensures that there will exist a (unique) queue with the same type and priority as the request:

Queues

$$queues : \mathbb{P}\, Queue$$

$$\forall\, l : LoginType; p : PRIORITY \bullet \exists\, q : queues \bullet q.type = l \land q.pri = p$$

$$\forall\, q_1, q_2 : queues \bullet (q_1.type = q_2.type \land q_1.pri = q_2.pri) \Rightarrow q_1 = q_2$$

The class definition for a *Queue* is presented below. Each queue has associated with it a priority (*pri*) and a *type* (*ONLINE* or *BATCH*), which remain constant for the queue. A *Queue* also consists of *items*, which is a sequence of *Requests*. The *Join* operation within *Queue* takes the incoming *Request* and adds it to the end of *items*. The state invariant of *Queue* ensures that the request will be added to the queue only if it has the same *pri* and *type* as the queue. The initialisation operation ensures that the *Queue* is created empty.

Queue

$$pri : PRIORITY$$
$$type : LoginType$$

$$items : \mathrm{seq}\, Request$$

$$\forall\, r : \mathrm{ran}\, items \bullet r.type = type \land r.pri = pri$$

INIT

$$items = \langle\, \rangle$$

Join

$$\Delta(items)$$
$$r? : Request$$

$$items' = items \frown \langle r? \rangle$$

The User *Send* and the OpSys *Accept User Request* occur in parallel, to ensure that the request sent by the user is the one received by the operating system. This co-ordination of operations between the two classes, indicated on the link diagram by the dotted lines, is handled at the *System* level, via the *Send* operation:

System

$users : \mathbb{P}\, User$
$computer : OPSys$

$Send \mathrel{\widehat=} u : users \bullet u.Send \parallel computer.AcceptUserRequest$

3.3 Link 2a: An Idle Processor Is Seized

Figure 6 illustrates this link, in which the request is passed from a queue to an idle processor.

The passing out of the request is handled by the *Leave* operation within *Queues*. The only attribute defined in the state schema of *Queues* is *queues*, which is a set of *Queue*. The

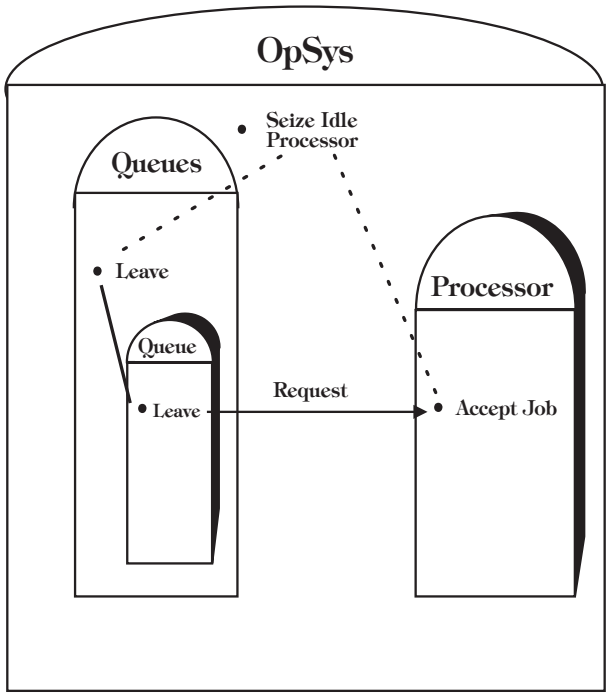


Figure 6: Link 2a

state invariants ensure that there is exactly one queue within *queues* with each of the possible combinations of priority and type, i.e. there will be four queues in *queues*:

- a priority 1, online queue
- a priority 2, online queue
- a priority 1, batch queue; and
- a priority 2, batch queue.

Queues

$$queues : \mathbb{P}\, Queue$$

$$\forall\, l : LoginType, p : PRIORITY \bullet \exists\, q : queues \bullet q.type = l \land q.pri = p$$

$$\forall\, q_1, q_2 : queues \bullet (q_1.type = q_2.type \land q_1.pri = q_2.pri) \Rightarrow q_1 = q_2$$

Leave

$$\Delta(queues)$$

$$\exists\, q, q' : Queue \bullet q = HighestPriority(queues) \land q.Leave \land queues' = (queues - q) \cup q'$$

The *Leave* operation passes out a *Request* from one of the queues in *queues*. The Queue selected cannot be empty (i.e. the initialisation condition cannot be true) and has the highest priority of those queues which are not empty (*HighestPriority* is a function which returns the queue with the highest priority out of those which are not empty—for an explanation of all of the functions used within the specification, including *HighestPriority*, see Appendix B). The operation triggers the *Leave* operation of the *Queue* chosen.

The *Leave* operation within the *Queue* class simply removes a *Request* from the front of the sequence (which cannot be empty), and passes it out:

Queue

$\vdots$

Leave

$$\Delta(items)$$
$$r! : Request$$

$$items \neq \langle\,\rangle$$
$$items = \langle r! \rangle \frown items'$$

The *Request* which is passed out from *Queues* is received by a *Processor*, via the *Accept Job* operation. Each *Processor* has two attributes associated with it: *jobs*, which is a set of *Requests*; and *timeRemaining*, which is a counter of how much processing time is required to complete a request. The number of requests in *jobs* must be either 1 or 0. If the number is 0 (i.e. the processor is empty), then the *timeRemaining* counter must 0 or less. The *Accept Job* operation receives the incoming *Request* and adds it to *jobs*, provided that *jobs* is empty. The *timeRemaining* counter is set to the amount of time it will take to process the request (*Time* is a function which returns the amount of processing time associated with a given priority, and is also presented in Appendix B).

Processor

$jobs : \mathbb{P}\, Request$
$timeRemaining : \mathbb{Z}$

$\#jobs \leq 1$
$\#jobs = 0 \Rightarrow timeRemaining \leq 0$

AcceptJob

$\Delta(jobs, timeRemaining)$
$r? : Request$

$jobs = \varnothing$
$jobs' = \{r?\}$
$timeRemaining' = time(r?.pri)$

The Queues *Leave* and the Processor *Accept Job* occur in parallel, to ensure that the request sent by the *Queues* is the one received by the processor. This co-ordination of operations between the two classes is handled at the *OpSys* level, via the *Seize Idle Processor* operation:

OPSys

$waiting : Queues$
$procs : \mathbb{P}\, Processor$

$SiezeIdleProcessor \mathrel{\widehat=} p : procs \bullet waiting.leave \parallel p.AcceptJob$

3.4 Link 2b: A Busy Processor Is Seized

An alternative to waiting for an idle processor is the seizing of a busy one. This link, illustrated in Figure 7, involves the request preempting a busy processor (the request must be a priority 1, online request for this to occur).

As with the previous link, the request is passed out from *Queues*. In this case, however, the operation involved is not *Leave*, but rather *Preempt*:

Queues

$Preempt \mathrel{\widehat=} q_1, q_2 : queues \bullet q_1.type = ONLINE$
$q_1.pri = 1$
$q_2.type = BATCH$
$q_1.Leave$
$q_2.Rejoin$

The request is passed out from the priority 1, online queue via its *Leave* operation. At the same time, an incoming request (the one being preempted) is added back on to its queue via the *Rejoin* operation. This part of the *Preempt* operation is not relevant to this link, so *Rejoin* will not be explained here. It is relevant to link 3 however, and will be discussed then.

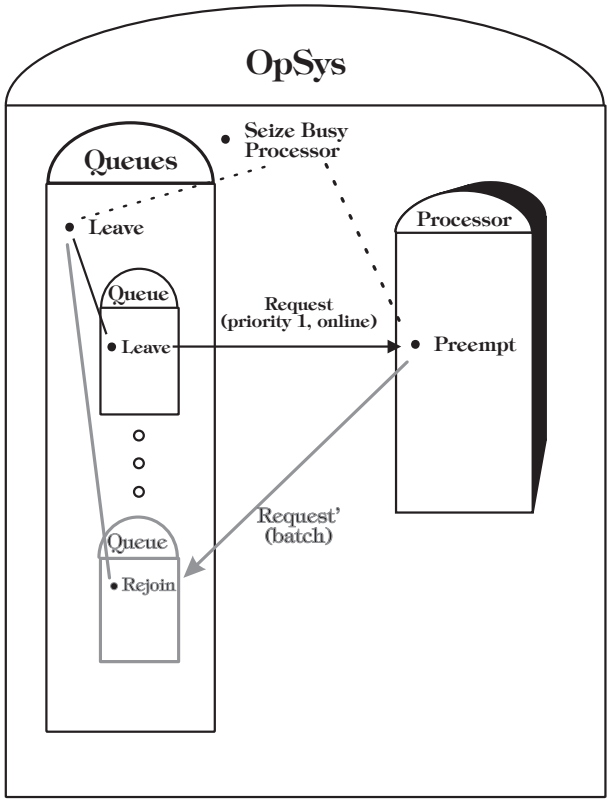


Figure 7: Link 2b

The request which is passed out is received by a processor via its *Preempt* operation, which is shown below. The request which is currently being processed (which must be of type $BATCH$) has its *Preempt* operation triggered, then it is passed out. The incoming request becomes the only member of *jobs*, and the *timeRemaining* for the processor is set to the time required for the new request.

Processor

$\vdots$

Preempt

$\Delta(jobs, timeRemaining)$

$r?, batchreq! : Request$

$\exists\, r, r' : Request \bullet\ r.type = BATCH$

$r \in jobs$

$r.preempt$

$batchreq! = r'$

$jobs' = \{r?\}$

$timeRemaining' = time(r?.pri)$

The purpose of the *Preempt* operation for a *Request* is to set the boolean attribute *preempted* to true:

Request

$$preempted : \mathbb{B}$$

Preempt

$$\Delta(preempted)$$

$$type = BATCH$$
$$preempted'$$

The Queues *Preempt* and the Processor *Preempt* occur in parallel, to ensure that the request sent by the Queues is the one received by the processor. This co-ordination of operations between the two classes is handled at the *OpSys* level, via the *Seize Busy Processor* operation. In addition to coordinating these parallel operations, this operation also updates the account information for that user, to reflect the $2 preemption charge. The *Seize Busy Processor* operation cannot occur if the *Seize Idle Processor* operation could occur instead.

OpSys

$$waiting : Queues$$
$$procs : \mathbb{P}\, Processor$$
$$accounts : UserId \pfun \mathbb{N}$$

SeizeBusyProcessor

$$\Delta(procs, waiting, accounts)$$

$$\neg\, (\mathrm{pre}\, SiezeIdleProcessor)$$
$$\exists\, p, p' : Processor \bullet p \in procs$$
$$waiting.Preempt$$
$$\|$$
$$p.Preempt$$
$$\|$$
$$[accounts' = accounts \oplus \{(r?.uid, accounts(r?.uid) + 2)\}$$
$$procs' = procs - \{p\} \cup \{p'\}]$$

3.5 Link 3: The Request Is Preempted

This link, illustrated in Figure 8, involves the request being preempted by another request. This can only occur if the type of the request is batch. Note that the link diagram is essentially the same as that for link 2b. This is because all of the operations associated with preemption, across several classes, are tightly coupled—they all occur in parallel. Indeed, the operations involved with this link are the same as those for the previous link: *Queues* performs a *Preempt* and a *Processor* performs its *Preempt* operation. In this case however, the request we are concerned with is not the one being passed in to the processor, but rather the one which is being preempted.

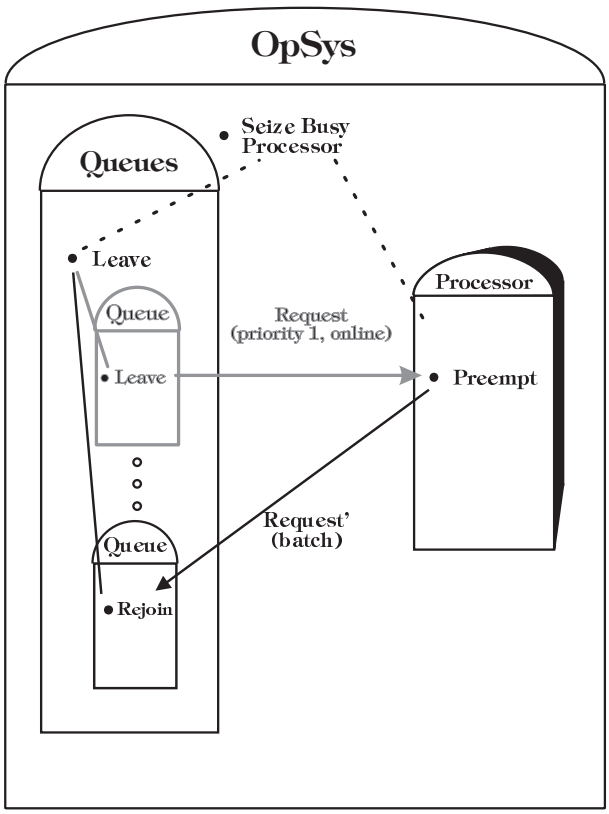


Figure 8: Link 3

In this case, we are interested not in the *Queue* performing the *Leave* operation, but rather the *Queue* performing the *Rejoin* operation (mentioned in the previous link):

Queue

$\vdots$

Rejoin

$\Delta(items)$

$batchreq? : Request$

$type = BATCH$

$items' = \langle batchreq? \rangle \frown items$

The operation receives a request of type batch (the request which has been preempted) and adds it to the front of *items*. The priority is not specified because either batch queue could be involved—the state invariants for a *Queue* ensure that the request can only be placed back onto the correct queue. At this point, the event chain loops back to link 2a, as the request will again be passed from the queue to an idle processor. Note that the *Preempt* operation of the *Processor* is the same as for the previous link. In this case, however, the request with which we are concerned is not the one being passed in to the *Processor*, but rather the one being replaced (*batchreq!*). Similarly, the *OpSys* operation responsible for coordinating these operations is again *Seize Busy Processor*.

3.6 Link 4: The Request is Completed

The final link in the event chain, link 4, is depicted in Figure 9. This link involves the processor completing the request submitted.

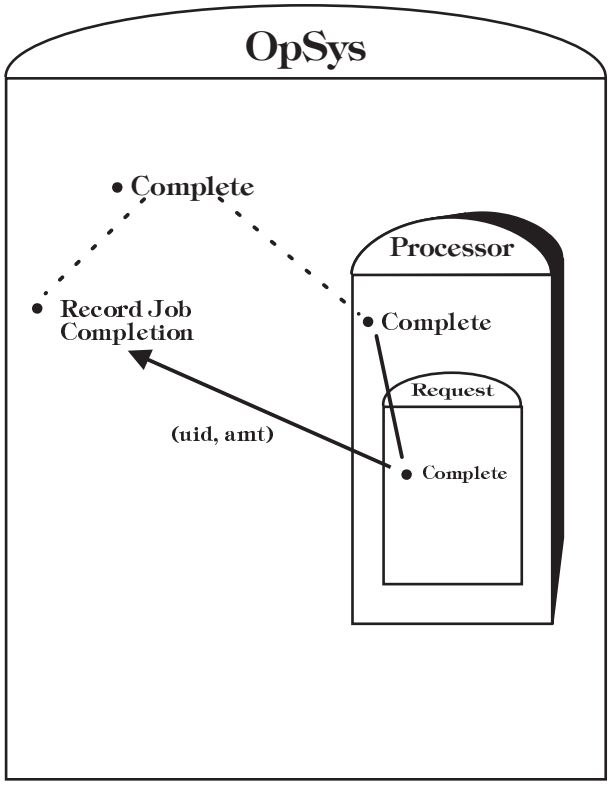


Figure 9: Link 4

The *Processor* handling the request performs its *Complete* operation. For the processing of the request to be complete, the *timeRemaining* of the processor must be equal to 0. The *Complete* operation of the *Request* is triggered, then *jobs* is reset to being empty.

Processor

$$jobs : \mathbb{P}\, Request$$
$$timeRemaining : \mathbb{Z}$$

Complete

$$\Delta(jobs)$$

$$timeRemaining = 0$$
$$\exists\, r, r' : Request \bullet r \in jobs$$
$$r.Complete$$
$$jobs' = \varnothing$$

The *Complete* operation of the *Request* passes out the *uid* of the request, as well as the amount (*amt*) owing for the request. If the request has been preempted then the charge is 0, otherwise it is equal to the processing time taken multiplied by the cost for that type of request (*cost*, like *time* is a function, which returns the charge per nanosecond for the type of request involved—again, for the definition of *cost* see Appendix B).

Request

$\vdots$

Complete

$uid! : Userid$

$amt! : \mathbb{N}$

$uid! = uid$

$preempted \Rightarrow amt! = 0$

$\neg\ (preempted) \Rightarrow amt! = time(pri) * cost(type)$

The completion of a request is handled within the *OpSys* via the Φ*Record Job Completion* operation. This operation updates the *account* information of the user submitting the request, and adds the charge associated with the request to the existing total. The *uid* and *amt* involved are received as inputs, and are equal to those which were passed out by the *Request.Complete* operation triggered by the processor's *Complete* operation. To ensure that the *uid* and the *amt* passed out from the *Request.Complete* operation are the ones received by the Φ*Record Job Completion* operation, these operations occur in parallel. This is coordinated at the *OpSys* level by the *Complete* operation.

OpSys

$accounts : UserId \pfun \mathbb{N}$

$\Phi RecordJobCompletion \mathrel{\widehat=} uid? : UserId;\ amt? : \mathbb{N} \bullet$
$\qquad accounts' = accounts \oplus \{(uid?, accounts(uid?) + amt?)\}$

$Complete \mathrel{\widehat=} p : procs \bullet p.complete \parallel \Phi RecordJobCompletion$

4 Conclusion

The work reported in this paper is a result of an extensive action research project undertaken at the Western Australian Department of State Services. The main aim of this project was to evaluate and enhance an information systems development methodology (FOOM) originally described in Swatman & Swatman (1992). One difficulty addressed within this project was the validation of formal (Object-Z) specifications by clients expert in neither mathematics nor object-orientation. The diagrammatic notation described within this paper—an extension to Henderson-Sellers and Edwards' MOSES notation which explicitly describes the manner in which the interactive and cooperative behaviour of components supports system level observable behaviour—was found to have a significant favourable impact on this process.

In this paper we have presented an argument that the perspective offered by the event chain notation is valuable within the process of validating object- oriented systems requirement models. We have demonstrated, in the context of our development methodology, how we utilise this notation to assist user acceptors to investigate and, thus, validate mathematically formal requirements specifications written in Object-Z.

While, in our approach, the Object-Z specification is the definitive statement of requirements to which the eventual implementation is required to conform, the event chain notation is clearly applicable more generally. Indeed, the event chain notation may be applied as an extension

to a broad range of object- oriented requirements notations with minor modifications to the concrete icons.

In further work, we are investigating the (semi-) automatic generation of diagrams in the MOSES notation (with the extensions presented here) from specifications in Object-Z; and to the development of an analyst's workbench which supports the use of our development methodology within the information systems domain. Once a prototype of this tool has been implemented, we intend to conduct a series of field studies to further evaluate both FOOM generally and, more specifically, the event chain notation.

References

Booch, G. (1991). *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, California, 2nd edition.

Checkland, P. & Scholes, J. (1990). *Soft Systems Methodology in Practice.* Wiley, Chichester.

Checkland, P. B. (1981). *Systems Thinking, Systems Practice.* Wiley, Chichester.

Checkland, P. B. (1989). Soft systems methodology. *Human Systems Management*, **8**(4), 237–289.

Coleman, D., Hayes, F., & Bear, S. (1992). Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, **18**(1), 9–18.

Diller, A. (1990). *Z: An Introduction to Formal Methods.* John Wiley & Sons.

Duke, R., King, P., & Smith, G. (1991a). Combining formal methods with object-oriented design: An emerging trend in software engineering. In *Proc. Australian Computer Conference—ACC'91.*

Duke, R., King, P., Rose, G., & Smith, G. (1991b). The Object-Z specification language: Version 1. Technical Report 91-1, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia.

Fichman, G. & Kemerer, C. (1992). Object-oriented and conventional analysis and design methodologies. *Computer*, pages 22–39.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, **8**, 231–273.

Henderson-Sellers, B. & Edwards, J. M. (1994). *BOOKTWO of Object-Oriented Knowledge: The Working Object.* Prentice Hall, Sydney.

Henderson-Sellers, B., Edwards, J. M., & Constantine, L. L. (1992). The extended uniform object notation. submitted for publication.

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

Jacobson, I., Christerson, M., Jonsson, P., & Overgaard, G. (1992). *Object-Oriented Software Enginnering: A Use Case Driven Approach.* Addison-Wesley, Wokingham, England.

Lee, Y. & Park, S. (1993). Opnets: an object-oriented high-level petri net model for real-time system modelling. *Journal of Systems Software*, **20**, 69–86.

Lorenz, M. (1993). *Object-Oriented Software Development.* Prentice Hall, Englewood Cliffs.

Monarchi, D. & Puhr, G. (1992). A research topology for object oriented analysis amd design. *Communications of the ACM*, **35**(9), 35–47.

Olle, T. W., Hagelstein, J., Macdonald, I., Sol, H., Assche, F. V., & Verrijn-Stuart, A. (1991). *Information Systems Methodologies: A Framework for Understanding.* Addison Wesley, Wokingham, 2nd edition.

Peterson, J. (1977). Petri nets. *ACM Computing Surveys*, **9**(3), 223–52.

Peterson, J. (1981). *Petri Net Theory and the Modelling of Systems.* Prentice Hall, Englewood Cliffs.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorenson, W. (1991). *Object-Oriented Modelling and Design.* Prentice Hall, Englewood Cliffs, N.J.

Spivey, J. M. (1989). *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire HP2 4RG, UK.

Susman, G. I. & Evered, R. D. (1978). An assessment of the scientific merit of action research. *Administrative Science Quarterly*, **23**, 582–603.

Swatman, P. A. (1993). Using formal specifications in the acquisition of information systems: Educating information systems professionals. In J. P. Bowen & J. E. Nicholls, editors, *Z User Workshop: London 1992*, Workshops in Computing, pages 205–239. Springer Verlag, London.

Swatman, P. A. & Swatman, P. M. C. (1992). Formal specification: An analytic tool for (management) information systems. *Journal of Information Systems*, **2**(2), 121–160.

Wirfs-Brock, R., B.Wilkerson, & Wiener, L. (1990). *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, N.J.

A A Brief Introduction to Object-Z

The specification language **Z** (Spivey, 1989) developed at the Oxford University has been extended at the University of Queensland (Duke *et al.*, 1991b) with the object-oriented concepts of *Class* and *Instance*. Here, we will introduce features of the language which are used in the simplified operating system specification presented in the main body of this paper. A number of textbooks introducing **Z** are available, including (Diller, 1990), though there are, as yet, no textbooks concerned with Object-Z.

When specifying using Object-Z, one must first identify the components (or objects) which, together with their interactions, comprise the system. The behaviour of each class of object identified is then specified by means of a *Class Schema*:

StaffMember

We begin by defining constants which apply to each instance of this class. In this case, any members of the class *StaffMember* will have *maxYears* associated with them. *maxYears* will be constant for any particular *StaffMember* but may vary from *StaffMember* to *StaffMember*. *maxYears* has been declared to be of type $\mathbb{N}$, that is, *maxYears* is a natural number—in the set {0,1,2,3,...}.

$$maxYears : \mathbb{N}$$

The following (unnamed) box is called the *State schema* and, above the line, contains variables which will represent the internal state of each instance of the class. As each instance of the class passes through its "life" the values of these variables and, thus the internal state of the instance, will change. Below the line in the *State schema*, we describe constraints on what we consider to be valid *StaffMember*s—in this case that no staff member may be on staff for longer than the period *maxYears* which was set when s/he joined.

$$\begin{array}{l} name : NAME \\ department : DeptName \\ yearsService : \mathbb{N} \\ \hline yearsService \leq maxYears \end{array}$$

In the following box, INIT, we describe the condition which must hold for a *StaffMember* to be in his/her initial state—in this case, that initially s/he has completed no service.

INIT

$$yearsService = 0$$

Each schema below represents an operation which objects of this class may perform. In each *Operation schema* variables are declared above the line and predicates, which constrain the relationships between the variables, are set out below the line. The object may only be manipulated by means of its operations—its state should not be altered directly, though interrogation of the state variables is allowed.
It is sometimes useful to declare operations which are used by other operations within the object, but which may not be invoked directly by other objects within the system. These auxiliary operations, called *Framing schemas* are distinguished from ordinary operations by the initial letter Φ.

Join

The first line lists those state variables which this operation may alter—in this case, name and department. State variables which have not been named in the delta list remain unchanged.
We include the initial condition schema INIT here which adds the condition which was defined in INIT to the condition which we define here—in this case we strengthen the precondition for this operation (only people who have no years of service can join).
Information which is passed in from the outside world when the operation is called have the suffix "?".
Information which is to be passed out of the object when the operation is called is given the suffix "!".

$\Delta(name, department)$

$newPerson? : NAME$

$newDepartment! : DeptName$

We define the state of the *StaffMember* object after undergoing the join operation to have the input name and to have been assigned to some department. Variables after the operation are distinguished from those before the operation by the decoration $'$—for example, $name'$. Operations may be more or less deterministic as desired. In this example we set $name'$ deterministically, but allow *department* to take any (valid) value, then report the value set in *newDepartment!*, the output variable.

$name' = newPerson?$

$department' = newDepartment!$

In the specification of the *Join* operation, *newPerson*? was declared to be of type *NAME* and *newDepartment*! of type *DeptName*. Only a limited number of types are predeclared in Object-Z—typically, the well known mathematical sets such as $\mathbb{N}$, the Natural Numbers. We can, however, define new types. In the case of *Name* and *DeptName* our interest is not in the form which these types may take but, simply, in the existence of a set of things which can be considered to be *NAME*s and a set of things which can be considered to be *DeptName*s. Object-Z allows us to declare the existence of such types in the following way (though, strictly, such type declarations should be made before they are used):

$[NAME, DeptName]$

In addition to its use of schema boxes, Object-Z makes use of many symbols drawn from the world of formal logic and mathematics. The most important symbols are set out in Figure 10.

$==$	is defined to be
$\times$	cartesian product
$\forall$	for all...
$\exists$	there exists...
$\mid$	a delimiter – can often be read as **where**...
$\bullet$	another delimiter – can often be read as **such that**...
$\wedge$	logical **and**
$\vee$	logical **or**
$\neg$	logical **not**
$\Rightarrow$	implies
$\{\ \}$	bracket a set
$\varnothing$	the empty set
$A : \mathbb{P}B$	A is of type powerset of B i.e. A is some set of B
$a \in A$	a is a member of the set A
$\cup$	set union
$\cap$	set intersection
$\subseteq$	subset
$\subset$	proper subset
$\setminus$	set subtraction
$\leftrightarrow$	relation
$\pfun$	partial function
$\fun$	total function
$\mapsto$	maplet – links the two items in an ordered pair
(a,b)	an alternative representation of an ordered pair
$a\ \underline{rel}\ b$	equivalent to $(a, b) \in rel$
dom	domain of a relation
ran	range of a relation
seq	sequence
Φ	as initial letter indicates a framing schema
Δ	introduces a list of variables which may be changed by an operation
a.op	the operation *op* on variable *a*
$'$	as suffix, indicates a variable name after the operation
!	as suffix, indicates an output variable
?	as suffix, indicates an input variable
;	a separator used in variable declarations

Figure 10: Notation for Specifications in Object-Z

Typically, we build up system specifications by combining schemas. For example, we can model a very simple personnel system as follows:

PersonnelSystem

No constants this time.
The personnel system is just a set of *StaffMember*s—initially an empty set.

$$employees : \mathbb{P}\, StaffMember$$

INIT

$$employees = \varnothing$$

We can now specify an operation which allows a person to join the company. We have declared input variable newPerson? and output variable newDepartment!. These variables are identified with the variables of the same name in the StaffMember's *Join* operation when we make the statement $s.Join$. As a consequence, a *Join* operation occurring at the system level means that the *Join* operation occurs to the *StaffMember* called *NewPerson* (who is not already an *employee*) and the newly *Join*ed *StaffMember* becomes a member of the set which comprises the *employees* recorded in the system.

Join

$$\Delta(employees)$$
$$newPerson? : NAME$$
$$newDepartment! : DeptName$$

$$\exists\, s, s' : StaffMember \bullet s \notin employees \\ s' \notin employees \\ s.Join \\ employees' = employees \cup \{s'\}$$

To complete our rather trivial system, we define an operation which allows employees to leave the company. This operation illustrates one of the advantages of a formal specification language over a programming language. We do not need to do any housekeeping. When an employee leaves service, we don't care about him/her any more and this is mirrored in our specification. The employee leaves service when s/he is removed from our set of *employees*—and that's all we need to say.

Leave

$$\Delta(employees)$$
$$leaver? : NAME$$

$$leaver? \in employees$$
$$employees' = employees \setminus \{leaver?\}$$

B Specification Functions

This appendix contains the definitions of the types and functions used within the example operating system specification provided in Section 3.

To begin with, *UserId* and *ProcId* are given types which are used to identify users and processes, respectively:

[UserId, ProcId]

Each *Request* and *Queue* within the system has a *LoginType*, which will be either *ONLINE* or *BATCH*:

$$LoginType ::= ONLINE \mid BATCH$$

Each *Request* and *Queue* within the system will also have a *PRIORITY* associated with it. As there are only two possible priorities within the system, *PRIORITY* contains 1 and 2.

$$PRIORITY == \{1, 2\}$$

There are three functions used throughout the operating system specification—*time*, *cost* and *HighestPriority*. *Time* associates every *PRIORITY* with a natural number, which represents the number of nanoseconds of processor time required by a request of that *PRIORITY*. Specifically, a priority of 1 returns the number 3 (i.e. 3 nanoseconds) and a priority of 2 returns the number 4 (i.e. 4 nanoseconds). *Cost* associates every *LoginType* with a natural number, which represents the cost per nanosecond of processor time for requests submitted by users logged in with that *LoginType*. Specifically, *ONLINE* requests cost $3 per nanosecond and *BATCH* requests cost $2 per nanosecond.

$$\begin{array}{l} time : PRIORITY \rightarrow \mathbb{N} \\ cost : LoginType \rightarrow \mathbb{N} \\ \hline time(1) = 3 \land time(2) = 4 \\ cost(ONLINE) = 3 \land cost(BATCH) = 2 \end{array}$$

Lastly, *HighestPriority* returns the *Queue* in the system with the highest priority. Empty *Queue*s are discounted. Any *ONLINE Queue* has precedence over any *BATCH Queue*s. *Queue*s with a *PRIORITY* of 1 have precedence over those with a *PRIORITY* of 2, within the same type—i.e. a priority 1 *BATCH Queue* has precedence over a priority 2 *BATCH Queue*, but not over a priority 2 *ONLINE Queue*.

$$\begin{array}{l} HighestPriority : \mathbb{P}\, Queue \nrightarrow Queue \\ \hline \forall\, Q : \mathbb{P}\, Queue; q : Queue \bullet q = HighestPriority(Q) \Rightarrow \\ \quad q \in Q \\ \quad \neg\, q.init \\ \quad ((q.type = ONLINE \land \\ \qquad \nexists q_1 : Q \bullet (\neg\, (q_1.init) \land \\ \qquad\quad q_1.type = ONLINE \land \\ \qquad\quad q_1.pri < q.pri)) \lor \\ \quad (q.type = BATCH \land \\ \qquad \nexists q_1 : Q \bullet (\neg\, (q_1.init) \land \\ \qquad\quad (q_1.type = ONLINE \lor \\ \qquad\quad q_1.pri < q.pri)))) \end{array}$$