A Knowledge Based Technique for the Process Modelling of Information Systems: the Object Life Cycle Diagram

Saimond Ip and Tony Holden

Information Engineering Division
Department of Engineering, University of Cambridge
Trumpington Street, Cambridge CB2 1PZ, United Kingdom

Abstract. This paper presents a new technique for IS process/behavioural modelling. Object Life Cycle is proposed as an extension of the conventional entity life history diagram with a Petri-Net based formalism and an Event-Precondition-Action process representation. A normalization approach for IS process modelling is suggested and several OLC norms discussed. Generalization and aggregation of OLCs are explored along with the concepts of a substate and a component event. Coordination of the objects via event raising is visualized by the Inter-Object Communication Model. Finally, we discuss how OLCs relate with RUBRIC rules and data flow diagrams.

1 Introduction

As part of an ongoing project to develop a new generation of knowledge based IS Methodology, a set of new techniques is developed for conceptual modelling, one of which is the Object Life Cycle Diagram (OLC) for behavioural/process modelling. We examine several major existing techniques and try to find out what constitutes a good process model. The result is presented in section 2. With these discoveries as our guide, OLC is developed and its basic constructs are presented in section 3. Two important methodological features are also developed alongside OLC. Firstly, a normalization approach is adopted in process modelling and several OLC normal forms are discussed in section 4. Secondly, two abstraction principles, namely, generalization and aggregation, are employed extensively in OLC diagrams and detailed discussions of how they are used can be found in sections 5 and 6. A major advantage of OLCs is that, at a certain normal form, they can be easily implemented and executed in a knowledge based systems environment. The last section briefly examines this issue.

2 Towards an Integrated Extension of Existing Techniques

Roughly speaking, IS Engineering involves three different parties: the end-user, the programmer, and the modeller [7], with the last (who constructs the conceptual

model) acting as the bridge between the first two. First and foremost, a conceptual model must be *user-oriented*, that is, it should be readily understood by an end-user and conform to his way of thinking about the problem. It would also be advantageous if the same techniques can be extended to model more implementational details for the programmer. The ideal solution is therefore a technique which can offer a *continuous* spectrum of forms, ranging from user-oriented high level models to programmer-oriented implementational models. Last but not least, a good model would lead itself to different *abstraction* principles.

What makes structured analysis techniques ([11] [15]) such as data flow diagrams so widely used? It appears that data flow is a very natural way for end-users to visualise a system and the diagrammatic representations are simple but very powerful. The important abstraction principle of functional decomposition is also naturally built into data flow diagrams and is often employed extensively. But their high level descriptions cannot be readily extended to provide more implementation details. Yet structured techniques is now so well established and popular that it is both unwise (since it is proven to be a useful approach) and impractical (since so many professionals are trained and so many specifications already written) to abandon them altogether. Any new technique should be complementary to structured analysis and integrate seamlessly with it.

Two other techniques have been applied fruitfully to IS analysis. First, there are interesting attempts to utilize different kinds of Petri Nets (eg. [13] [26] [32] [34] [36] [44]), most notably its successful integration with data flow and process models [13]. The main attraction of Petri Net is its ability to provide a continuous spectrum of forms covering both highly abstract user-oriented models and actually executable models with formal properties and implementation details. Several abstraction principles based on net refinements (which usually amounts to functional decomposition) have also been employed in IS analysis and design.[1] Second, the RUBRIC project[29] has demonstrated that business rule is a very useful paradigm for IS Engineering. It recognises the explicit separation of organizational policy from implementational details and is therefore geared towards the organization's way of thinking. On the other hand, sufficiently well defined rules can be readily analysed and are executable and, therefore, has a high degree of continuity. A rule is also a completely independent unit and free from considerations of procedural control flows.

It is observed that the only abstraction principle used in data flow diagrams and Petri Nets is functional decomposition. Generalization and aggregation, often in the form of "object-orientation", are now widely acknowledged as important and useful abstraction principles and are employed extensively in programming languages[42] [28] and databases[2]; but their use in ISE are so far confined to data modelling[45] [20]. Their extensive use becomes a major motivation behind the development of our new process model. The idea of an object is particularly appealing because it entails the concept of encapsulation[28] which tends to give more modular and re-usable designs. Entity Life History/Cycle (ELH/C) Diagram is a well-integrated

[1]Examples of net refinement principles are refinement of surroundings[26], refinement by S- and T-Sets[36], and refinement by primitives[13].

part of traditional structured methodologies, such as SSADM[16], and our approach is to extend it by giving it a new formalism and new concepts to become the Object Life Cycle (OLC) Diagram. The final representation chosen is Petri-Net based with additional rule-based constructs. This is made possible by the remarkable similarity between an event-condition-action rule ([29] [10]; the Remora IS methodology also employs similar constructs[35]) and a transition in a predicate/transition net with its input predicates represented by the conditions and output predicates determined by the rule's actions. The event of the rule is a special input predicate that will be consumed by another "garbage-collection" transition if it is not instantaneously used by the original transition. Finally, the integration of the OLC and data flow diagrams (section 8) is facilitated by the distinction between the two different ways which a process can involve any object, either to change its state (object flow) or simply to read and use some of its information (data flow) [32] [44].

3 The Object Life Cycle Diagram (OLC)

An OLC is constructed for every object class defined in Object Relationship Model (ORM) [20]. Figs. 1 & 2 give examples of different OLCs. Every object in a class must be in a certain *state*.[2] The possible states of an object class are shown as circles in an OLC. Every object class in ORM must also have "state" as one of its attributes. There are two special states that are universial to all OLCs: never-exist and cease-to-exist. A complete OLC must begin with the former and end with the latter. An object can be transformed from one state to another by a *process* (shown as a round-cornered box). A process may have more than one input and output states. An input and an output state of a process can be the same. If they have to be the same, a double-arrowed link is used. A double-lined and double-arrowed link next to a process means that it can operate on an object at any state (except never-exist and cease-to-exist) without changing the state.

The full definition of a process contains three distinct parts: event, precondition, and action. When an *event* (shown as a box with darkened left edge) is raised to an object at one of the input states of a corresponding process, the conditions are checked. If all the conditions are met, the actions will be performed and the object transformed to the output state of the process. A process can be triggered by more than one event and several processes may have the same triggering event (although once an event have triggered a particular process, it is retracted and cannot trigger further processes). For example, in fig. 1, if ?pub (an instance of Publication) is at the state borrowed or overdue and ?b (an instance of Borrower) at the state registered, the process Return-Publication will be triggered by the event Pub-Returned to check whether (?pub is-borrowed-by ?b). If the condition is met, the process will delete all the facts related to ?pub being borrowed by ?b and assert that a publication has been put back into the location ?loc. The state of ?pub will be changed to on-shelf.

[2]Similar to our approach, the Lifecycle diagram proposed by Shlaer[38] also uses events and states. However, Schlaer includes all the dynamic responses to incoming events in the concept of a state and does not put them in separate processes.

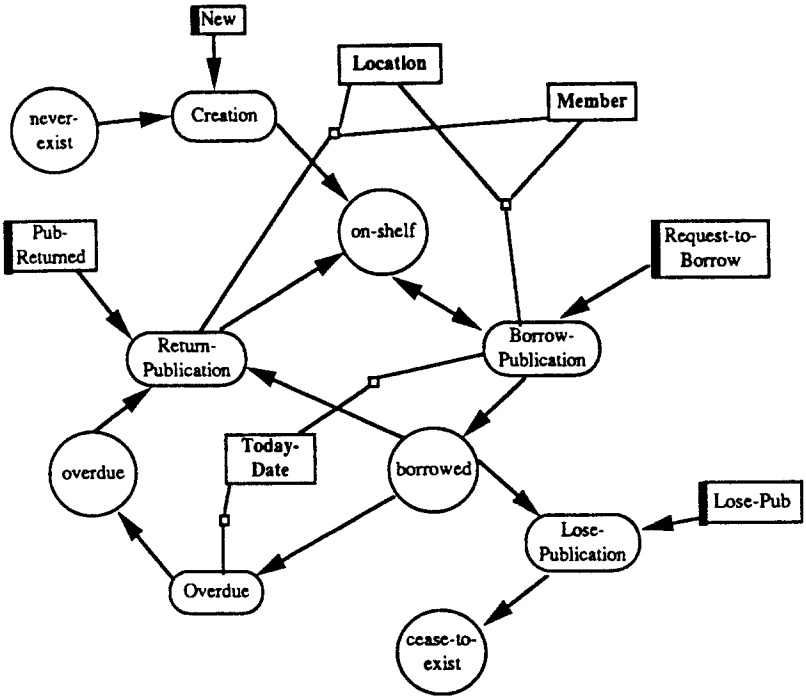


Fig. 1(a) OLC for Publciation

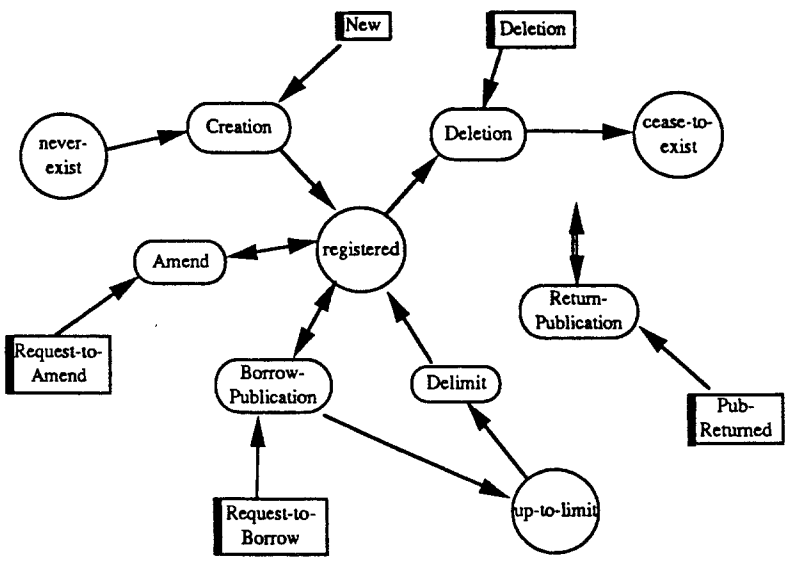


Fig. 1(b) OLC for Borrower

External object classes are simply shown as boxes. If a process belongs to more than one object, then, in the context of the OLC of any one of the objects, it can be linked to the appropriate external objects (shown as a link with a small box). For example, in fig. 1(a), the process Return-Publication involves objects of classes Publication (the class of the OLC itself), Member, and Location. If a process simply need to read-access the information about an external object, a simple link is used. The information required from an external object and an event can be (optionally) written on the link. For example, in fig. 2(a) the process Borrow-Publication needs the information about the identity of the publication to be borrowed (?pub) from the triggering event Request-to-Borrow, the state and the location of ?pub from Publication, and the accessibility of the location of ?pub from Location.

The events that trigger processes to create an object from the state never-exist (defaulted to be New) and to delete it into the state cease-to-exist are really raised to the class of the object instead of the instance itself. For example, in fig. 1(b) the event New tells the object class Borrower to create an instance of itself with certain parameters.

Where do events come from? Some are external events (eg. Request-to-Borrow in fig. 1) while others are internal ones (eg. Pub-Borrowed in fig. 2(c)) and are generated by the action of a process. Hence, an external event might trigger off a chain of events. But as far as an OLC is concerned, the source of an event is completely irrelevant. This independence is clearly an advantage since the boundary of a system often changes enormously. A process may also raise an event to the external environment. Such events to and from the external forms the sole interface between the process model and its environment.

Occasionally, a process has to be automatically triggered when all its preconditions are met. For example, in fig. 2(c) the process Overdue should be triggered to warn of an overdue whenever the due-date of the publication is today. This amounts to a process with a triggering event permanently and repeatedly raised (shown simply as a "TRUE" event). The process would simply be reduced to a normal forward chaining rule, that is, the actions would be performed if all the preconditions *become* true. The process would therefore only be triggered by some change in the system and not repeatedly by the preconditions remaining true.

4 A Normalization Approach to Conceptual Modelling

A *norm* (or a normal form) of any model is a particular form with some predefined desirable properties. But a normalization approach is more than just using norms in modelling. It is the provision of a spectrum of successively more restrictive norms. It is argued in section 2 that the ability of a modelling technique to provide such a spectrum (what we call the technique's continuity) is clearly advantageous.[3]

[3]The success of the normalization approach in the design of relational database [9] is at least a good indication of its potential in process modelling. In fact, relational normal forms (in particular, up to the third normal form) are so successful that normalization becomes almost synonymous with relational analysis.

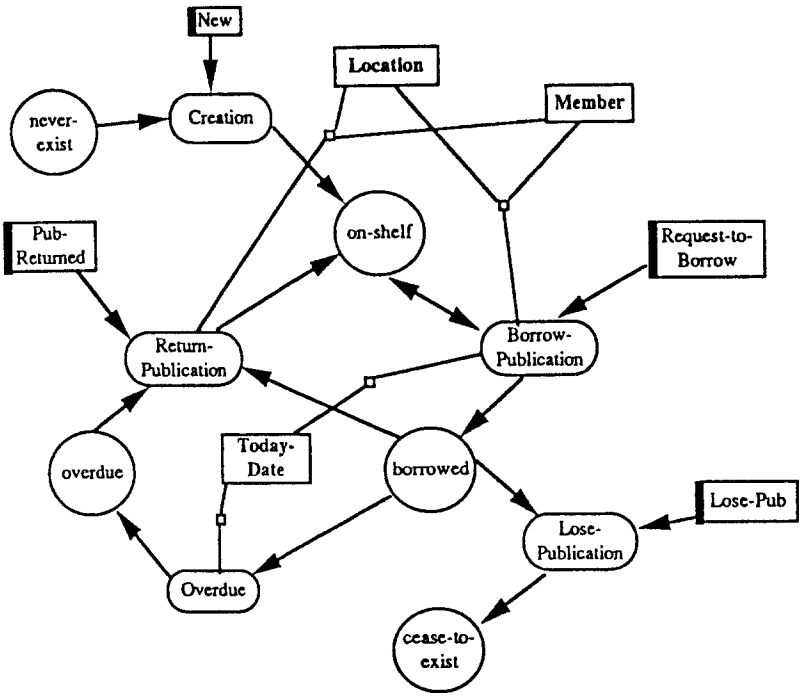


Fig. 1(a) OLC for Publciation

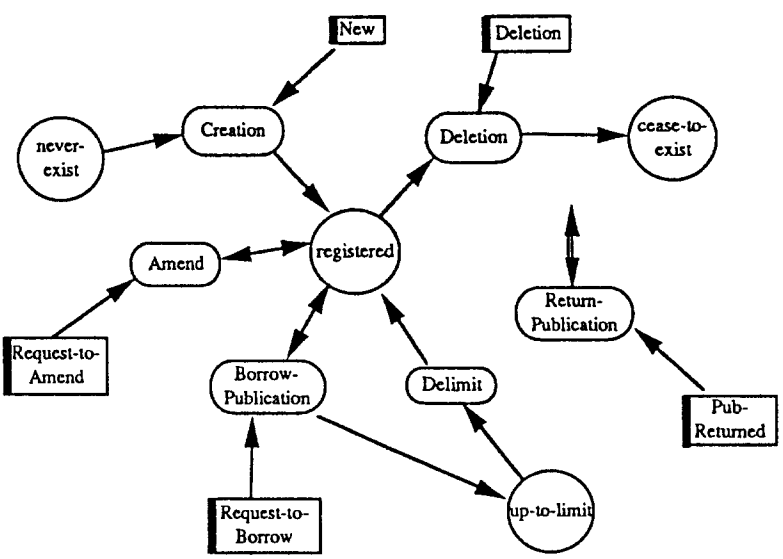


Fig. 1(b) OLC for Borrower

A normalization approach is appropriate for process modelling because of the different benefits offered by the models at the two ends of the spectrum. Relatively unnormalized forms are more abstract (easier to understand and to construct as a kind of "first draft"), less formal (requires less technical knowledge to use), and usually conform better to the end-users' way of thinking. On the other hand, successively more normalized forms contain more detailed information (valuable for implementation and possibly automatic code-generation) and is more formal (allow more formal analyses) and more similar to the way the IS is eventually implemented. Norms can also act as a template to standardize a model by enforcing specific arrangements of some model information. The beauty of the normalization approach is that no one is forced to go through a rigid set of norms. Experienced analysts (or even end-users) might skip the "early" unnormalized forms and increasingly model directly in more normalized forms. Modellers might not want to "go all the way" and use all the norms. They might find some norms too detailed to be used for conceptual modelling and others with properties unnecessary for their circumstances. Everyone just uses the norms they find useful and convenient.

The five norms of OLC are given in table 1. The encapsulation norm is based on the classification norm (ie. a model in the encapsulation norm must also be in the classification norm) and all the other three norms are based on the encapsulation norm. These last three norms are basically independent of each other. The list is not meant to be exhaustive and new norms might be developed as the need arises. It should also be pointed out that the encapsulation norm is regarded as the corner stone of our process model and any OLC should eventually be transformed into this norm while the last three norms are relatively optional and are useful only for specific reasons.

Table 1 Norms of OLC

The Classification Norm

Definition: An OLC is in the *Classification Norm* if and only if each process and each event belongs to one and only one object class.

Example: OLCs in fig. 1 are unnormalized since Borrow-Publication and Return-Publication belongs to two object classes but fig. 2 contains OLCs in the classification norm. Return-Publication of Borrower in fig. 2(a) and Return-Publication of Publication in fig. 2(c) are now two different processes.

Rationale: This norm forces the modeller to assign each process (and hence its triggering events) to an object class and hence makes all processes subordinate to objects and the abstraction of classification uniformly endorsed. Unnormalized forms allow assignment decisions to be delayed and recorded explicitly (possibly for future revisions).

The Encapsulation Norm

Definition: An OLC is in the *Encapsulation Norm* if and only if each process can only modify the object instance to which its triggering event is raised.

Example: Borrow-Publication in fig. 1 is broken into Borrow-Publication of Borrower, Borrow-Publication of Pub, and Remove-Pub of Location (not shown) in fig. 2 (all OLCs in encapsulation norm).

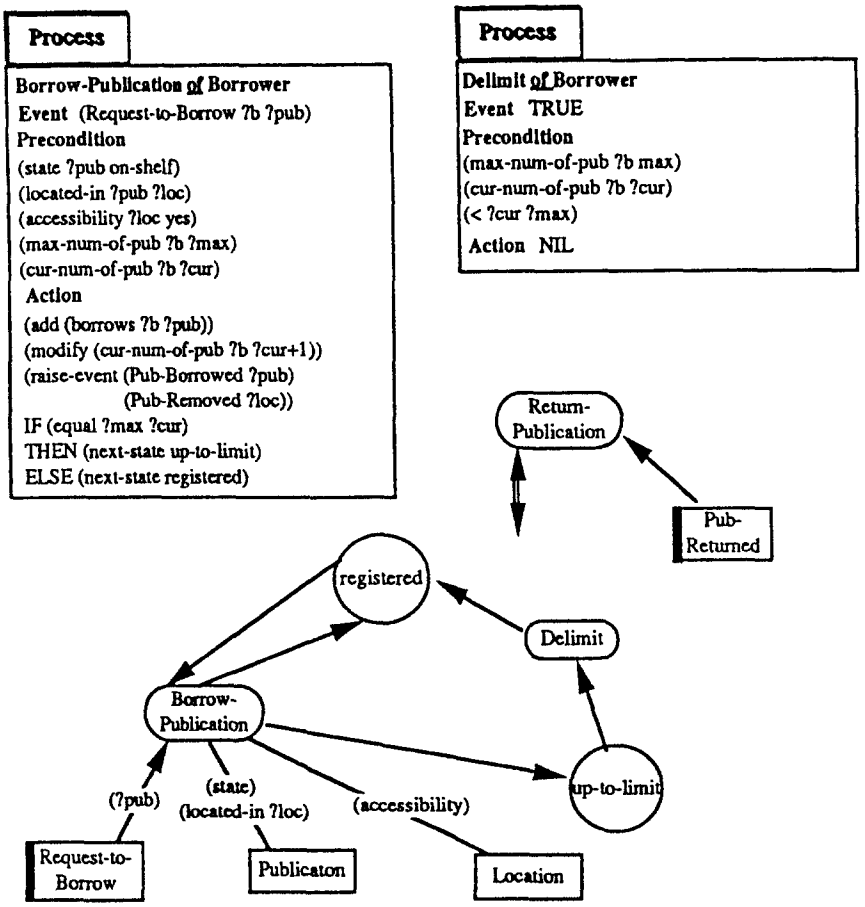


Fig. 2(a) The Processes Borrow-Publication & Delimit of Borrower in Classification and Encapsulation Norm

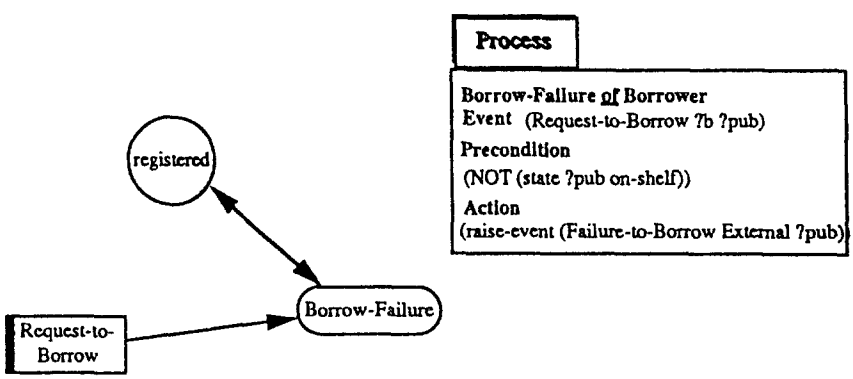


Fig. 2(b) The event Request-to-Borrow triggering an alternative Process

Rationale: Modular style of representation enforced by encapsulation or information hiding [28] [42]. A process can only modify another object by raising events to it; hence the triggering events to an object form its only interface with the rest of the system. The process model becomes a set of independent objects communicating via events.

The Weak and The Strong ORM Norms

Definitions: An OLC of an object class A is in the *Weak ORM Norm* if and only if an object to which one of its processes can raise an event must be: 1) an instance of an object class (or the object class itself) directly related to A in the ORM or 2) given to the process as an argument of its triggering event. The OLC will be in *Strong ORM Norm* if and only if it is in the weak ORM norm and each process can only read-access objects of the above two categories[27].

Examples: Let us assume that in the ORM Borrower is only related to Publication which is also related to Location. Borrow-Publication of Borrower in fig. 2(a) wants to raise an event Pub-Removed to Location and, in the weak ORM norm of fig. 3, this job is delegated to Borrow-Publication of Publication. Fig. 4 shows how strong ORM norm causes additional events to be raised for accessing an indirectly related object (note the final event Loc-info raised at the other end of the access path to pass back the information to the original enquiring object Borrower).

Rationale: Further enhance information hiding and decoupling of control by narrowing the accessibility of an object to its related objects in the ORM, cf. the inter-object communication model in section 7. If any process (or data structure) of an object class A is modified, one needs only to consider its effect on related objects in the ORM and objects taking A as an event argument. Standardization in the decomposition of event chains is also enforced along a path on the ORM. The strong ORM might hinder readability due to the large number of events being raised.

The Discrete Time Norm

Definition: An OLC is in the *Discrete Time Norm* if and only if each of its processes has no duration at all.

Example: Fig. 7 shows how part of an OLC can be transformed into the discrete time norm. Any non-discrete time elements are pushed into the states of the OLC and/or to other objects (in this case the state potentially-deleted and External).

Rationale: This norm ensures the independence of a process and its sole access to the relevant information. It tackles the problem of I/O uncohesiveness [1] and forces all complicated (and asynchronous) interactions among processes to be be modelled explicitly. It also makes an OLC executable (section 9).

The Explicit-Condition Norm

Definition: An OLC is in the *Explicit-Condition Norm* if and only if each of its processes has a unique output state.

Example: The Explicit-Condition norm of Borrow-Publication in fig. 2(a) is given in fig. 5.

Rationale: To explicitly model the conditions for choosing the output states as the preconditions of the process.

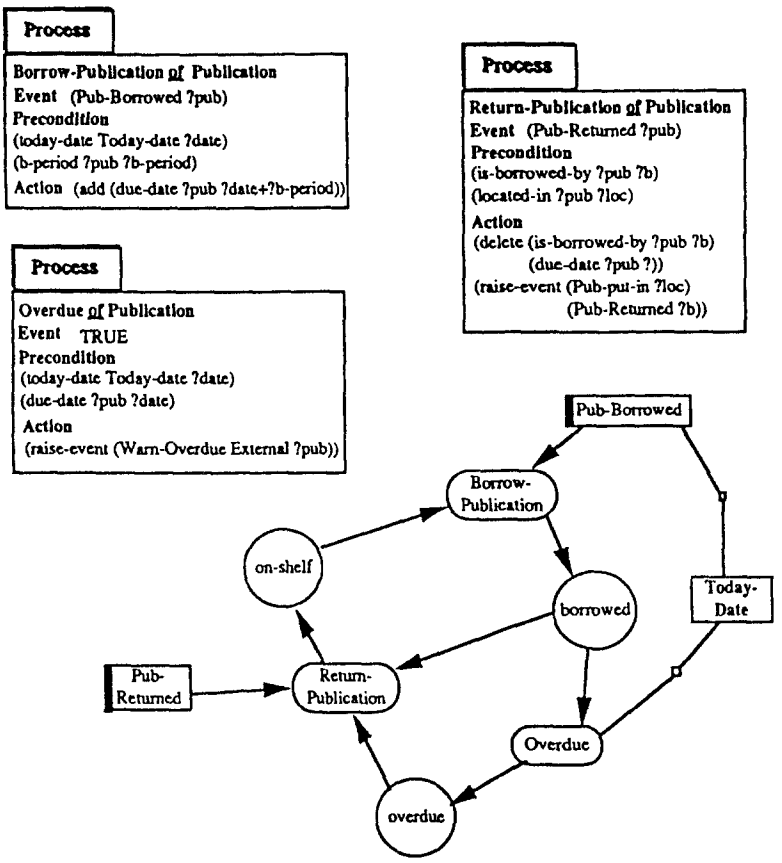
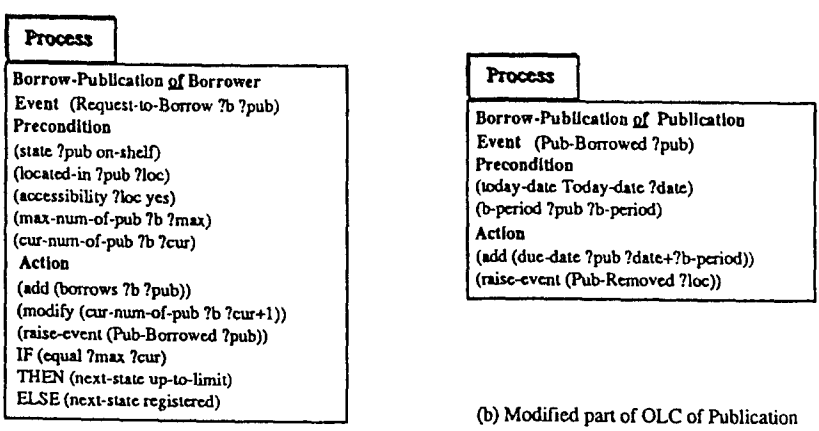


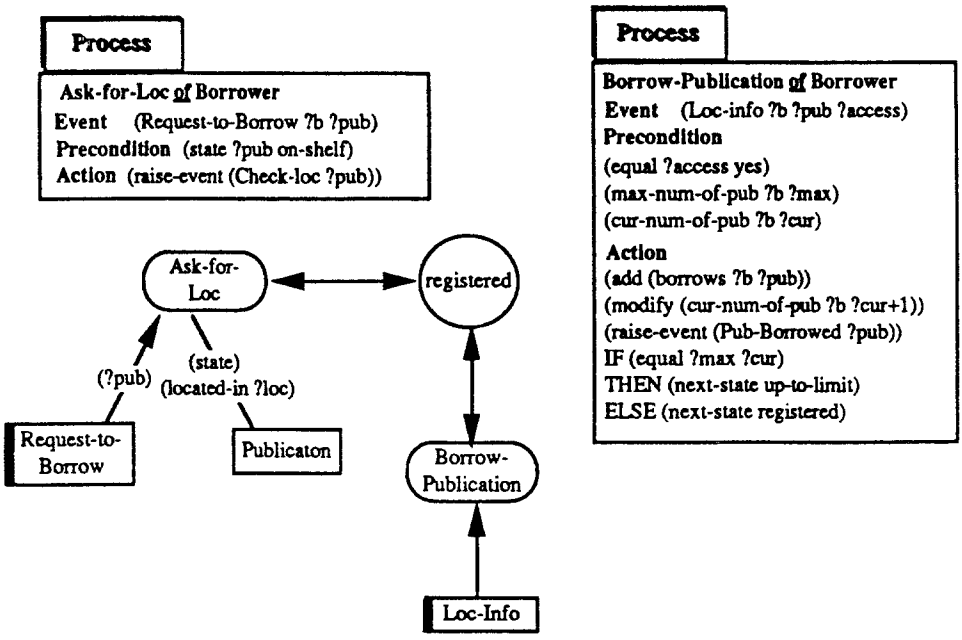
Fig. 2(c) Part of the OLC of Publication in the Encapsulation (& Classification) Norm



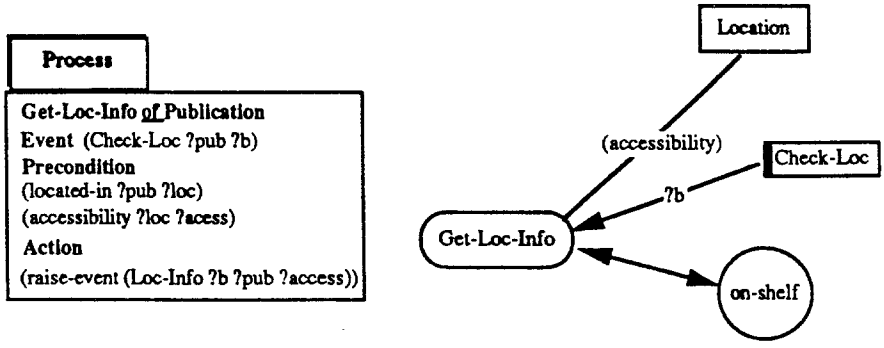
(a) Modified part of OLC of Borrower

(b) Modified part of OLC of Publication

Fig. 3 Borrower and Publication in Weak ORM Norm



(a) Modified Part of OLC of Borrower



(b) Modified Part of OLC of Publication

Fig. 4 OLCs in Strong ORM Norm

5 Generalization of OLCs

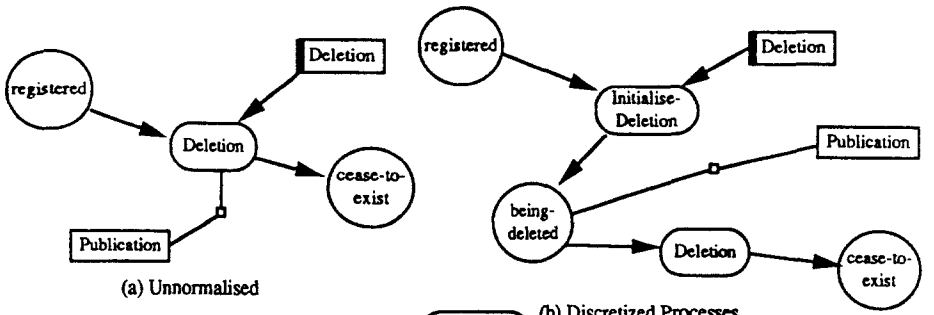
The idea behind generalization[4] is to group similar objects together in a representation with their common properties. Classification is therefore one form of generalization. This section, however, focuses on how classes can be generalized to give a concise and meaningful representation of the process model. Any object class (the superclass) can be specialized to form a subclass. This specialization can be thought of as a *restriction* in the structure and behaviour of the superclass. By definition, any member of the subclass must also belong to the superclass. The subclass inherits both the structural and the behavioural information (the entire OLC including all the processes, events, and states) from its superclass. A subclass can modify the behaviour of its superclass in two different ways[5]: either by adding and defining new states, events, and processes or by re-defining processes.

The *granularity of inheritance* has long been recognized as an important factor in the usefulness of any generalisation/specialization relationships[42]. Inheritance will be much more powerful if incremental additions can be made to a process. OLC has a finer granularity than conventional object-oriented programming languages since an event can be raised to different processes of the same object and any modifications can be limited to only some of these (and only the necessary parts of the process, including i/o states, preconditions, or actions, are changed). There are a number of attempts in object oriented languages to provide incremental specialization of functions[42], most notably the declarative method combination (eg. the division of a method into before/after/main parts in Flavors) and the use of commands like "Super" in Loops which invokes the method of a superclass in a local method that re-defines it. The separation of processes and states allows specialization of OLCs to effectively subsume these two schemes. Firstly, by re-defining the input and/or output states of a "main" process, an arbitrary number of processes can be added both before or after it (or in any other topology) ,possibly with the triggering event raised to preceding processes instead. Secondly, the "Super" invoking command can be achieved by simply having the original process intact (except for input/output states). Any additional behaviour can be added on with the triggering event possibly raised to a different process. For example, in fig. 9(d) the process deletion of privileged borrower is inherited unchanged from the OLC of borrower. But an additional condition (check-address) is checked (with a new state mid-check added) and an extra action (send-farewell-letter) performed (with another state to-send-farewell introduced).

Multiple inheritance is so useful that most object oriented systems incorporate it in various degrees. OLC is rather flexible and can be associated with many different strategies of conflict resolution. For demonstration purpose, we have chosen a scheme that gives precedence to subclasses over superclasses and to different superclasses

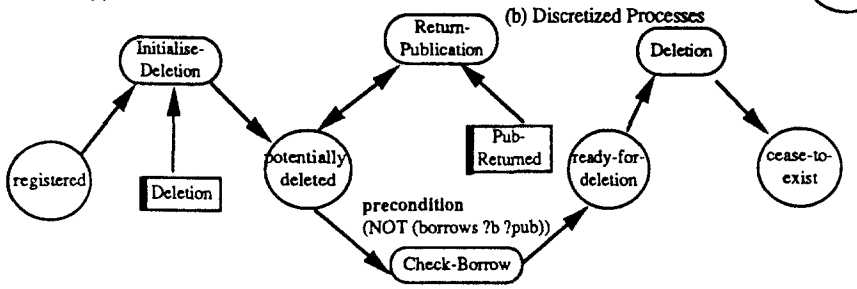
[4] [22] gives a more detailed discussion of abstraction principles, including the concepts of a mixin and non-excluding-subclass norm in generalization, the concept of a perspective in aggregation, and examples of functional decomposition, as applied to OLCs.

[5] A third type of modification that excludes the processes or states of a superclass is discussed in [22].

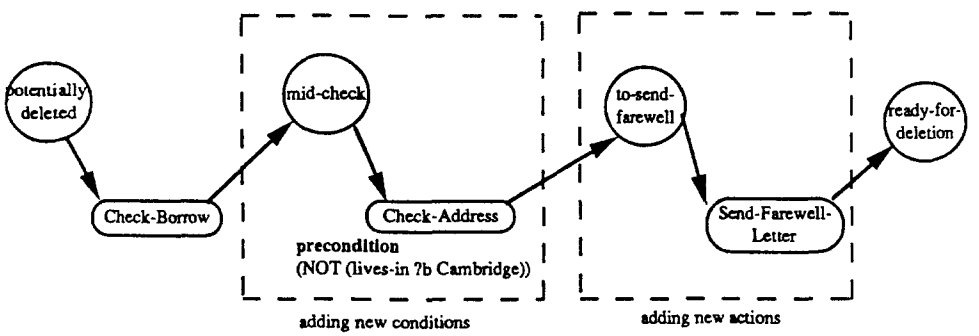


(a) Unnormalised

(b) Discretized Processes



(c) Discrete (and Encapsulated) Norms



(d) Incremental Specialization for Privileged-Borrower

Fig. 7 The Process Deletion in OLC of Borrower

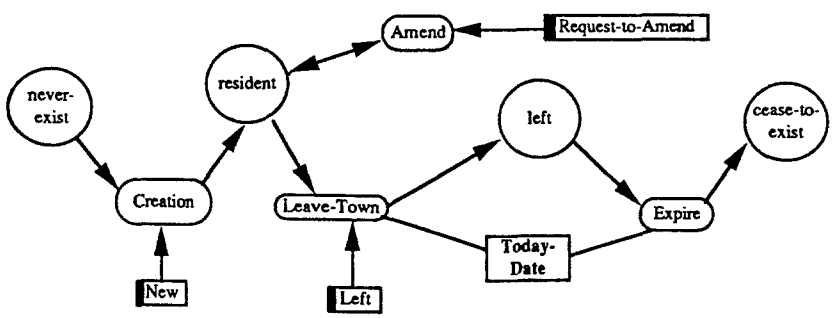


Fig. 8 OLC for Member in Encapsulated Norm

according to a predefined priority list (with warning issued to the designer). The result is similar to Loops[42] and is often known as a "left-right depth first up-to-joint (and including joint)[6]" selection strategy. How do we combine the OLCs of the different superclasses to form the OLC of the subclass? This is not a straight forward task and requires in-depth knowledge of the behavioural interaction between the superclasses. Typically it is only possible to trigger a process inherited from a superclass A when the object is at certain particular states inherited from another superclass B. Sometimes it is even necessary to decompose a state inherited from B to differentiate whether certain processes from A can be triggered. We introduce the concept of a *substate*[7] to deal with such circumstances.

State-1 of a subclass is defined to be a substate of state-2 of its superclass if and only if any process that can be triggered from state-2 (as its input state) can also be triggered from state-1. We also say that state-2 is the superstate of state-1. A state can have many substates and/or superstates. Needless to say, a process with a superstate as its output state must, in the subclass, select one or more of its substates as a substitute[8]. Fig. 9 gives an example of how the concept of substate is used to combine the OLCs of Member-Borrower's two superclasses Member (fig. 8) and Borrower. It is observed that processes of Borrower triggered from the state registered are always applicable at the state resident of Member but only sometimes applicable from the state left. The solution is to split left into two substates just-left and extended and to declare extended and resident to be substates of registered. The processes governing the transitions between just-left and extended (Extension and Extension-Expire) have to be defined. Additional events and processes operating on these new substates can also be introduced. Note that there is a potential conflict between the two processes Amend inherited from different superclasses[9] operating on the state resident of Member-Borrower. But the substate extended can only be operated on by the Amend inherited from Borrower.

6 Aggregation of OLCs

Aggregation is used extensively in programming languages (eg. Loops[42]) and in object oriented database systems[2] but most investigations so far concentrate on the structural relationships between an object and its components. This section looks at its use in behavioural modelling. But what are the benefits of using so many (and, some may add, so complicated) abstraction principles? We would like to reiterate our opinion that the usefulness of any abstraction can be seen from three different views[20]: 1) the representational view that allows us to model the

[6] Loops' strategy is left-right depth first up-to-joint but excluding joint.

[7] This "substate of specialization" must not be confused with the more commonly used substates and subprocesses in the functional decomposition of a process. Another very useful concept to tackle these problems is "perspective"[22].

[8] A process of the superclass may always leave a superstate unchanged in which case an instance of the subclass will also remain unchanged at whichever substate it happens to be at.

[9] If the two processes are triggered by events of different names, they can co-exist. But in this case, they are triggered by events of the same name and a choice has to be made between the two.

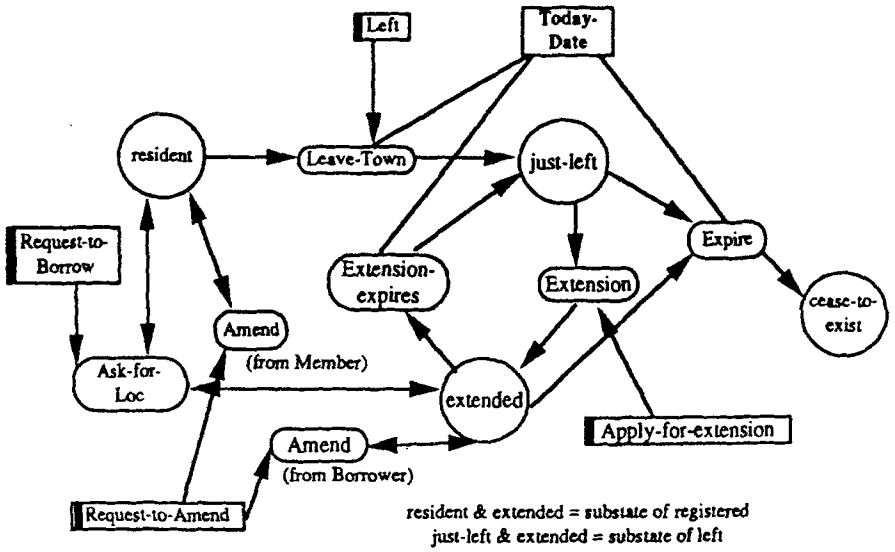


Fig. 9 Multiple Inheritance using Substate in the OLC for Member-Borrower (subclass of Member and Borrower)

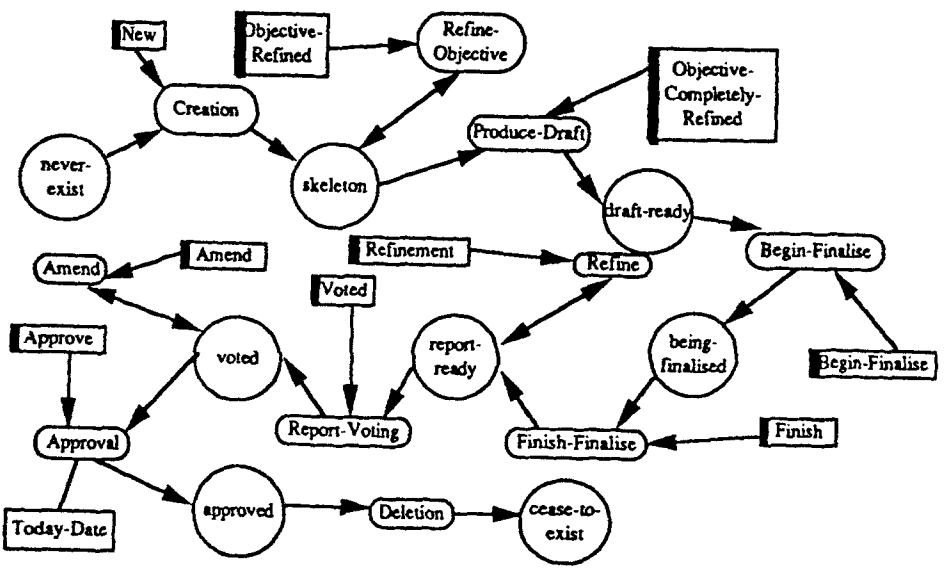
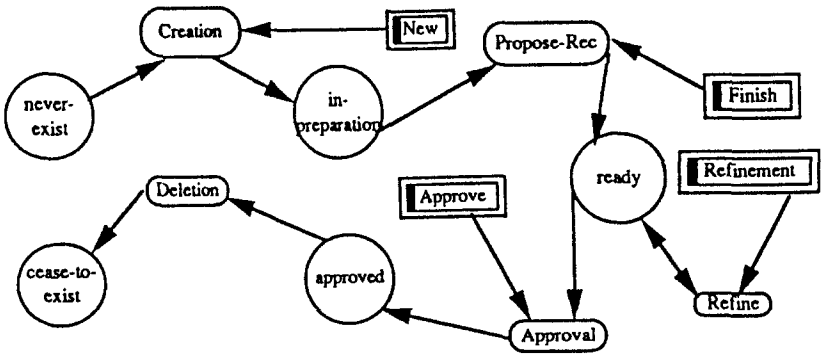
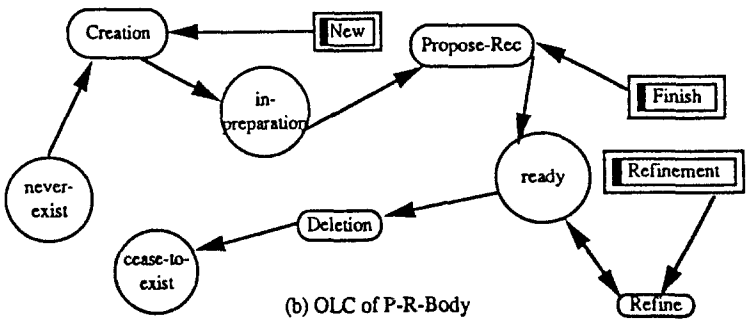


Fig. 10 OLC of Project-Report



(a) OLC of Recommendation



(b) OLC of P-R-Body

Process

Finish-Finalise of Project-Report
Event (Finish ?p-r)
Precondition
(has-rec ?p-r ?rec)
(has-body ?p-r ?p-r-body)
Action
(raise-component-event
(Finish ?rec)
(Finish ?p-r-body))

Process

Refine of Project-Report
Event (Refinement ?p-r ?x ?y)
Precondition
(has-rec ?p-r ?rec)
(has-body ?p-r ?p-r-body)
Action
(raise-component-event
(Refinement ?rec ?x)
(Refinment ?p-r-body ?y))

Process

Amend of Project-Report
Event (Amend ?p-r ?x)
Precondition
(has-rec ?p-r ?rec)
Action
(raise-component-event
(Refinement ?rec ?x))

Process

Approval of Project-Report
Event (Approve ?p-r ?exco)
Precondition
(has-rec ?p-r ?rec)
(today-date Today-date ?today-date)
Action
(raise-component-event
(Approve ?rec))
(add (has-date ?p-r ?x))

(c) Definitions of Processes of Project-Report that raises component events (to Recommendation and P-R-Body)

Fig. 11 OLCs of Recommendation and P-R-Body as Components of Project-Report

real world more closely and naturally; 2) the methodological view which sees an aggregate as a temporary "abbreviation" for further detailed expansion; and 3) the documentational view that presents relevant information of various degrees of details to users with different needs.

The use of complex objects in our data model ORM is reported in [20]. For our present purpose, it is suffice to know that a complex object class may contain many component object classes which may relate with each other. The example used here is the complex object class Project Report which has-rec an instance of Recommendation class and has-body an instance of P-R-Body which in turn may has-section one or many P-R-Section. As far as OLCs are concerned, a complex object is a sort of "supersystem" governing the behaviour of its subsystems, ie. its component objects. The current state of a complex object should indicate the availability of certain processes both of the complex object itself and of some of its components. Hence, when a process is performed on the complex object (and thereby attempting to change its state), some of the states of its components have to be changed simultaneously. Moreover, a process of the complex object often requires several simultaneous subprocesses of its components to achieve the desired effects. We propose the concept of a *component event* as the means to accomodate these characteristics.

The independent rule-like characteristic of an OLC process means that, under normal circumstances, after the process has raised an event, it would simply proceed to finish other actions and finally the object instance would be transformed to the output state[10]. But if a complex object raises component events to its component objects, the complex object instance can only proceed to the output state of the process when all the component events are "completely consumed". A component event is completely consumed when either 1) it does not trigger any process at all and is therefore deleted or 2) the component object instance is transformed to the output state of the process it triggers. Therefore, when a complex object reach the output state of a process, we can be certain that all the appropriate "subprocesses" are finished and corresponding state changes achieved. A complex object can, however, raise "ordinary" events to any of its components if the desired effect of component events is not needed. It is also noted that, in the weak ORM norm, a complex object can only raise component events to its direct components.

A rather detailed example of how the OLCs of the complex object Project-Report interact with the OLCs of its components (and with each other) is given in figs. 10 and 11. For instance, when the external event Finish is raised to a Project-Report (when it is at the being-finalised state), it triggers the process Finish-Finalise which raises two component events (both called Finish too!) to the Project-Report's Rec (Recommendation) and P-R-Body. The Project-Report can only proceed to the state report-ready when the processes triggered by the two Finish events are completed and both its Rec and P-R-Body have proceeded to their own ready states. It is interesting to observe that some of the processes of Project-Report, for example, Refine-Objective and Begin-Finalise, do not involve any of its components at all.

[10]This contrast sharply with the idea of a "method" in conventional object oriented programming language like Smalltalk-80. When a method send a message, it has to wait for an object to be returned before it can proceed.

These can be thought of as part of the *emergent* behaviour of a complex object.

7 The Inter-Object Communication Model (IOCM)

The OLC of an object class is the formalization of the typical behaviour of instances in the class. All instances (of all object classes) are assumed to operate independently, concurrently, and asynchronously and any coordinations and synchronizations should be modelled explicitly through the event raising mechanism. In order to help us to visualize these interactions, an *inter-object communication model* (IOCM) can be constructed[38]. It is simply a diagram with all the object classes joined by directional links labelled with the events raised. Fig. 12 shows part of the IOCM for the examples discussed in this paper. All specialization relationships should also be included in the IOCM because all events raised by/to a superclass are also raised by/to its subclasses. Component events raised by a complex object to its components is specially marked as such (in this case with dotted links). Following [38], we think it is sometimes useful to organize the large number of object classes (possibly dozens) into layers to facilitate the understanding of a IOCM. A set of very rough and informal guidelines is used. Examples include: an object usually receives guidance, requests, and coordinations from the objects above it (with complex objects always above their components), objects of a higher layer have more interaction with the "External", and the lowest layer consists of completely passive and unintelligent objects.

8 OLC and Related Process Modelling Techniques

This section will examine how OLC relates with other process modelling techniques, in particular, the RUBRIC project and conventional structured analysis. The dynamic rules of the process model of RUBRIC[29] is basically equivalent to a process in OLC with the same division into trigger (= OLC's event), precondition, and message (= action). In fact, OLC can be seen as an attempt to organize the vast amount of rules in a process model along the concept of an encapsulated object and to use a Net-based formalism to visualize their interactions. Some static RUBRIC rules are captured in our data model ORM while others are modelled as an OLC process with a "TRUE" event as explained in section 3 (eg. a constraint rule can be represented by such a "TRUE" event rule with warning actions).

OLC is seen as complementary to the use of data flow diagrams (DFD)[11] with the latter either developed in parallel with or act as the source specifications for the former. Most constructs in DFD can be represented accurately in OLC. A process in DFD can initially be modelled as an OLC process belonging to many different object classes in unnormalized forms and eventually be classified and attached to one specific object class (often one of its input data flow)[11]. An input data flow

[11] A data store is an object class with all its roles and attributes in ORM. All external entities are grouped into the "External" in OLC; though it might of course be broken up if necessary.

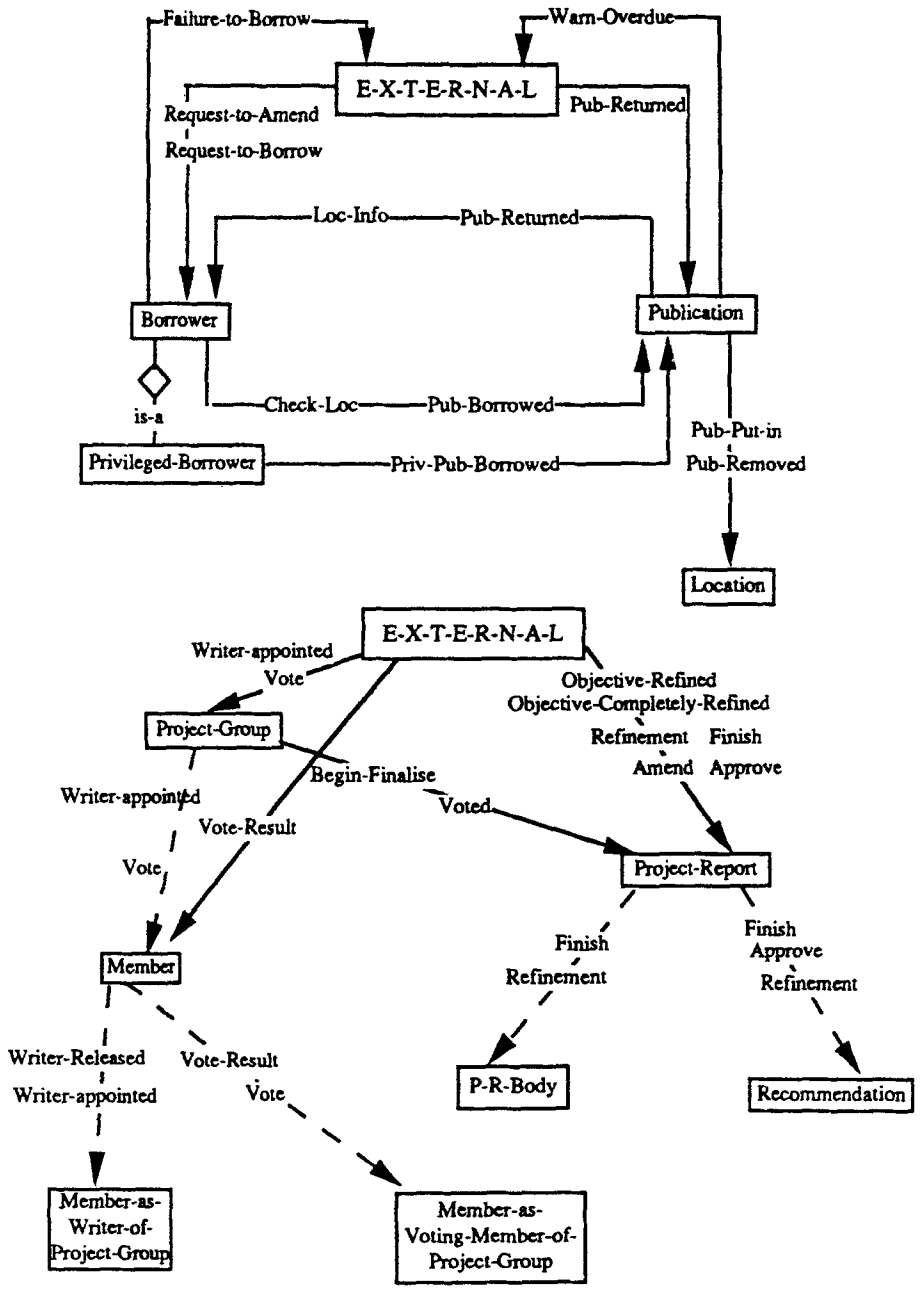


Fig. 12 Inter-Object Communication Model for Parts of the Case Study

to a process can be an "object flow", that is, a transformation of the state of the object if it is from a data store (especially the one that the OLC process belongs), a modification of the object's roles/attributes (where there will be both an input from and an output to the data store), or a precondition to the OLC process. Output flows to other data stores (that the process does not attach) signifies an attempt to change other object classes and should be normalized and broken up. A data flow from one process to another can either be a precondition (involving the former associated object class) for the latter or a triggering event raised by the former to the latter. Finally, functional decomposition of a DFD may be modelled in OLCs as the breaking up of processes in their normalization, processes of complex objects and subprocess of their component objects (triggered by component events), or simply as the functional decomposition of processes into small subcycles within an OLC[12].

9 Conclusion

It is believed that OLC can facilitate rapid prototyping by being transformed into a semi-executable conceptual model. With heavy designer interaction, an OLC in discrete-time norm can be represented by forward-chaining rules (with an event being a special left hand side condition to be deleted by a garbage-collection rule if it is not immediately consumed by a process rule). Such a model can be directly executed with sample object instances and data. As completely independent units, OLCs can be individually tested before integrated into larger systems. Powerful interactive facilities should be provided for browsing and experimenting with various scenarios to detect any errors and unintended behaviour. We are experimenting with ART[8] as the prototyping environment.

We set out to develop an user-oriented behavioural modelling technique that facilitates the use of abstraction, provides a continuous spectrum to different parties and is fully integrated with conventional structured techniques. We then demonstrated how a normalization approach for OLC can provide the spectrum, how generalization and aggregation can be applied meaningfully to OLCs, how OLCs can be simulated in a knowledge based environment, and finally how data flow diagrams integrate with OLCs. We are working on a prototype knowledge based support system on a Symbolics Lisp Machine for the construction of OLCs (and ORMs). A graphical editor is being built using Maxim[17] and translated into an ART[8] knowledge base. These are then validated and consolidated with other models. We are experimenting the techniques on a number of case studies. Examples in this paper are mainly from a case study involving a nation wide campaigning body which runs an information centre and has regular reports and publications.

Acknowledgement

We are grateful to Prof P. Loucopoulos and his colleagues at the Information Systems Group, Department of Computation, UMIST, UK, for discussions and the exchange of research information.

[12] We do not pay much attention to this important aspect of OLC modelling because there have already been a lot of investigations, especially as related to Petri Net-based models [13] [36] [26].

References

[1] ALABISO, B. "Transformation of Data Flow Analysis Models to Object Oriented Design". *OOPSLA '88 Proceedings*, pp.335–353, Sept 1988.

[2] BANERJEE, J. et al. "Data Model Issues for Object-Oriented Applications". *ACM Transactions on Office Information Systems*, 5(1):3–26, Jan 1987.

[3] BOBROW, D. G. et al. "CommonLoops: Merging Lisp and Object-Oriented Programming". *OOPSLA '86 Proceedings*, pp.17–29, Sept 1986.

[4] BOOCH, G. "Object-Oriented Development". *IEEE Transactions on Software Engineering*, SE-12(2):211–221, Feb 1986.

[5] BRUNO, G. and BALSAMO, A. "Petri Net-Based Object-Oriented Modelling of Distributed Systems". *OOPSLA '86 Proceedings*, pp.284–293, Sept 1986.

[6] CAUVET, C., PROIX, C. and ROLLAND, C. "Information Systems Design: An Expert System Approach". In MEERSMAN, R. A., (eds), *Artificial Intelligence in Databases and Information Systems (DS-3)*, North-Holland, 1990.

[7] CHEUNG, L., IP, S. and HOLDEN, T. "A Survey of AI Impacts on Information Systems Engineering". *Information and Software Technology*, To be Published.

[8] CLAYTON, B. D. *Inference ART: Programmers' Tutorial*, Inference Corporation, 1987.

[9] CODD, E. F. "A Relational Model of Data for Large Shared Data Banks". *Communications of ACM*, 13, 1970.

[10] DAYAL, U. et al. "Rules are Objects Too: A Knowledge Model for an Active Object-Oriented Database System". *Lecture Notes in Computer Science*, 334:129–143, Springer-Verlag, 1988.

[11] DE MARCO, T. *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

[12] ESSINK, L. J. B. and ERHART, W. J. "Object Modelling and System Dynamics in the Conceptualization Stages of Information Systems Development". *Object Oriented Approach in Information Systems*, pp.89–116, Holland, 1991.

[13] FALQUET, G. et al. "Concept Integration as an Approach to Information Systems Design". In OLLE T. W. et al., (eds), *Computerized Assistance During the Information Systems Life Cycle*, North-Holland, 1988.

[14] FRANCE, R. B. and DOCKER, T. W. G. "Formal Specification using Structured Systems Analysis". *Lecture Notes in Computer Science*, 387:293–310, Springer-Verlag, 1989.

[15] GANE, C. and SARSON, T. *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[16] HARES, J. S. *SSADM for the Advanced Practitioner*, Wiley, 1990.

[17] HOLDEN, T., WILHELMIJ, P. W. and APPLEBY, K. A. "Object-Oriented Design of Visual Software Using MAXIM". *European Conference on the Practical Applications of Lisp*, 1989.

[18] HOLDEN, T., CHEUNG, L. and IP, S. "Intelligent Support for the Information System Design Process". *European ART User-group Conference, Rome*, 1990.

[19] HULL, M. E. et al. "Object-Oriented Design, Jackson System Development (JSD) Specifications and Concurrency". *Software Engineering Journal*, pp.79–86, March 1989.

[20] IP, S., CHEUNG, L. and HOLDEN, T. "Complex Objects in Knowledge Based Requirement Engineering". *6th Knowledge-Based Software Engineering Conference, Syracuse, New York*, Sep 1991.

[21] IP, S., CHEUNG, L. and HOLDEN, T. "A Knowledge Based Requirement Engineering Assistant". *BCS CASE on Trial II Conference, Cambridge*, Mar 1992.

[22] IP, S. and HOLDEN, T. "Abstraction and Object Life Cycles in Process Modelling". submitted to *Journal of Information Systems.*

[23] IP, S. and HOLDEN, T. "A Knowledge Assistant for the Design of Information Systems". In DEEN, S. M. and THOMAS, G. P., editors, *Data and Knowledge Base Integration, Proceedings of the Working Conference on Data and Knowledge Base Integration held at the University of Keele, England on October 4–5, 1989*, Pitman, 1990.

[24] JACOBSON, I. "Object Oriented Development in an Industrial Environment". *OOPSLA '87 Proceedings*, pp.183–191, Oct 1987.

[25] KARAKOSTAS, V. "Modelling and Maintenance Software Systems at the Teleological Level". *Software Maintenance: Research and Practice*, 2:47–59, 1990.

[26] LAUSEN, G. "Conceptual Modelling Based On Net Refinements". *Database Semantics (DS-1)*, pp.41–57, North Holland, 1986.

[27] LIEBERHERR, K. et al. "Object-Oriented Programming: An Objective Sense of Style". *OOPSLA '88 Proceedings*, pp.323–334, Sept 1988.

[28] LOCKEMANN, P. C. "Object-Oriented Information Management". *Decision Support Systems*, 5:79–102, 1989.

[29] LOUCOPOULOS, P. "Improving Information System Development and Evolution Using a Rule-Based Paradigm". *Software Engineering Journal*, pp.259–267, Sept 1989.

[30] LOUCOPOULOS, P. "The Process Model of TEMPORA". UMIST, U.K., 1991.

[31] MANFREDI, F. et al. "An Object-Oriented Approach to the System Analysis". *Lecture Notes in Computer Science*, 387:395–410, Springer-Verlag, 1989.

[32] OBERQUELLE, H. "Human-Machine Interaction and Role/Function/Action-Nets". *Lecture Notes in Computer Science*, 255:171–190, Springer-Verlag, 1986.

[33] PALASKAS, Z. and LOUCOPOULOS, P. "AMORE: The RUBRIC Implementation Environment". UMIST, U.K., 1989.

[34] RICHTER, G. and DURCHHOLZ, R. "IML-Inscribed High-Level Petri Nets". *Information Systems Design Methodologies: A Comparative Review*, pp.335–368, North Holland, 1982.

[35] ROLLAND, C. and RICHARD, C. "The REMORA Methodology for Information Systems Design and Management". *Information Systems Design Methodologies: A Comparative Review*, pp.335–368, North Holland, 1982.

[36] REISIG, W. "Petri Nets in Software Engineering". *Lecture Notes in Computer Science*, 255:63–95, Springer-Verlag, 1986.

[37] SERNADAS, C. et al. "In-the-large Object-Oriented Design of Information Systems". *Object Oriented Approach in Information Systems*, pp.209–232, Holland, 1991.

[38] SHLAER, S. and MELLOR, S. J. "An Object-Oriented Approach to Domain Analysis". *ACM SIGSOFT Software Engineering Notes*, 14(5):66-77, Jul 1989.

[39] SIBERTIN-BLANC, C. "Co-operative Objects for the Conceptual Modelling of Organizational Information Systems". *Object Oriented Approach in Information Systems*, pp.297–321, Holland, 1991.

[40] SMITH, J. M. and SMITH, D. C. P. "Database Abstractions: Aggregation and Generalization". *ACM Transactions on Database Systems*, 2(2):105–133, Jun 1977.

[41] SNYDER, A. "Encapsulation and Inheritance in Object-Oriented Programming Languages". OOPSLA '86 Proceedings, pp.38–45, Sept 1986.

[42] STEFIK, M. and BOBROW, D. G. "Object-Oriented Programming: Themes and Variations". *The AI Magazine*, Winter 1986.

[43] STROUSTRUP, B. "What is Object-Oriented Programming?"

[44] STUDER, R. and HORNDASCH, A. "Modelling Static and Dynamic Aspects of Information Systems". *Database Semantics (DS-1)*, pp.13–26, North Holland, 1986.

[45] THEODOULIDIS, C., WANGLER, B. and LOUCOPOULOS, P. "Requirements Specification in TEMPORA". *Presented at Conference CAiSE'90, Stockholm*, May 1990.

[46] VOSS, K. "Nets in Office Automation". *Lecture Notes in Computer Science*, 255:234–257, Springer-Verlag, 1986.