The University Library Document Circulation System Specified in LCM[1]

Roel Wieringa Remco Feenstra[2]

Faculty of Mathematics and Computer Science, Vrije Universiteit
De Boelelaan 1081a, 1081 HV, Amsterdam
Email: roelw@cs.vu.nl, rbfeens@cs.vu.nl

December 20, 1993

Abstract

The specification language LCM (Conceptual Modeling Language) is used to specify a part of the university library document circulation system of the Free University [7]. LCM is version 3 of a language previously called CMSL (Conceptual Model Specification Language). The method used to specify the document circulation system is MCM (Conceptual Modeling Method). We draw a number of conclusions about the types of axioms that are encountered in this case study, as well as about the kinds of extensions that should be added to LCM to facilitate easier and more expressive modeling of this case study.

[1]This research is partially supported by Esprit Basic Research Action IS-CORE (working group 6071).

[2]Supported by the Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project 612-317-408.

Contents

Chapter 1

Introduction

LCM (Conceptual Modeling Language) is a language to formally specify conceptual models of database systems. LCM is version 3 of a language previously called CMSL (Conceptual Model Specification Language). The name change is motivated by the fact that the name "CMSL" is hard to pronounce. The syntax of LCM is described in a companion report [4]. Axiom systems and declarative semantics of LCM in terms of a Kripke structure combined with a process algebra is described in a number of papers [14, 19, 17]. First steps towards an operational semantics are taken by Spruit [12, 11].

The method used to analyze the case study is called MCM (Conceptual Modeling Method), described in another companion report [16]. This name is chosen to bring out the connection with LCM. MCM leads to two models, an informal and formal one. The informal model is meant as interface between the formal model on the one hand and the intuitive understanding of analysts and domain specialists on the other. The formal model is preferably specified in LCM. The intention is that it also could be specified in other languages, such as TROLL [9, 6], TROLL-*light* [2], Oblog [3] or FOOPS [5]. Further research is needed to see whether this can be realized.

Chapter 2 gives a description of the Library for Beta Sciences at the Free University based on a report by IJff [7]. Chapter 3 contains a model of the boundary of the library as a whole and of the library DBS in particular. Chapter 4 describes the informal UoD model resulting from applying MCM to the library case and chapters 5 and 6 give the corresponding LCM specification. Chapter 5 contains the formal specifications of all classes of objects and relationships found in the UoD registered by the circulation desk DBS. Any event occurring in the life of these objects or relationships will be registered by the DBS. Chapter 6 specifies the interface between the DBS and the UoD. This interface consists of (atomic) transactions, and each transaction consists of one or more events occuring in the UoD. If there is more than one event in a transaction, then these events are modeled as occurring synchronously. This means that chapter 6 not only specifies the boundary between the DBS and the UoD formally, but that it also specifies the communication structure in the UoD. Chapter 7 contains a discussion of the results and mentions subjects of current and future research.

Chapter 2

The Library case

The description in this chapter provides m ost of the information on which the formal specification is based.

2.1 The mission of the library

The mission of the Free University Library is described in the Guide for Library Users [13] as

1. to acquire documents containing information that is of use for scientific research and education,
2. to catalog these documents,
3. make them available,
4. preserve them, and
5. to act as custodian of the documents it acquired.

The collection is made available not only for the Free University but for any scientific research or education at all. Other universities and colleges have the right to use the library, and as a matter of fact do so. A prerequisite for getting registered as a library member is is showing a valid proof of identity. Private individuals can use the Library as well. However, most members are students or staff of the Free University.

2.2 Structure of the library

The library is divided into departments that more or less reflects the structure of the university in departments. Thus, there is the Biology library for the faculty of Biology, the Mathematics and Computer Science library for the Faculty of Mathematics and Computer Science, etc. Department libraries are located close to the Faculty they serve. Department libraries are grouped together into Scientific Area libraries. For example, there are α, β and γ areas. There is one administration that is used by all library departments. Library management has regular consultations with the university board and with member representatives. Other responsibilities of management include strategic planning, external reporting and budgeting.

2.3 Members

A member is someone who has a reader's pass. Any student or employee can acquire a pass, as well as citizens who are not otherwise related to the university. A group of employees can also acquire a pass, called a "group pass". There should be one person accountable for the actions of this group, but any member of the group can borrow a document using the group pass.

2.4 Documents

The library acquires a wide diversity of items, such as books, journals, series (e.g. the Springer Lecture Notes in Computer Science), Proceedings, internal reports from the Free University or from other universities, unpublished reports from research laboratories, Ph.D. theses, maps, microfiches, videotapes, old manuscripts, newspapers, microfilms, etc. Some of these are acquired for students (often several copies), most of these are for research purposes. Books themselves can come into a great variety of forms, such as multivolume works, multi-edition works, loose-leaved works that are regularly updated with new articles or with replacements of existing articles, manuals, etc. Some books can be split into two volumes as they go to their next edition, or an edition of a book can suddenly get a companion, called "Volume 2". Documents may occasionally go to the binder where they are repaired. Sets of journal issues go to the binder to be bound into single volumes.

Borrowing a document

Most documents can be lent to members, but some can only be read in special rooms. Similarly, old volumes of journals, when bound, can be borrowed, but loose issues cannot be borrowed. All items possessed by the library receive a unique code so that they can be identified, and they receive a unique location code so that they can be traced to a location.

A member can borrow a document for three weeks. Researchers in addition have the right to borrow it for three months. At the end of the allowed lending period, a member should return the document or else renew the borrowing. Renewal can only be done when there is no reservation for the document. If a member does not return a document or does not renew the lending period, action is only taken after 1 extra week, by sending him or her (or them) a reminder. So a member is reminded of his obligation to return the document 4 weeks after it was borrowed. If it is not yet returned or renewed, a second reminder is sent after 7 weeks. After the second reminder, the member has still one week to respond. If one week after the second reminder there is no message from the member, he or she must pay a fine of Dfl 70 and is not allowed to borrow any more documents until the document is returned and the fine is paid.

Reserving a document

Any member can reserve documents that are currently borrowed by someone else. He or she will receive a message when the document is available and the library will hold the document for one week so that this member can borrow the document. If the document is not fetched after one week, the document is allocated to the next reserver for that document, who receives a message that the document is available. If there is no next reserver, the document is returned to the shelf. There can be several reservers for a document, which are queued in a first-in first-out manner. There are two kinds of reservations, called "title reservation" and "copy reservation" respectively. A title reservation can be made when all copies of the title are out. When any copy of the title is returned, the reserver receives a notice and he or she has the first right to borrowing the copy. A copy reservation can be made when the individual copy desired by the library member is not available. This is done for particular editions of books, manuscripts, etc.

Losing a document

If a member loses a document, he or she has to report this to the administration, who will issue an invoice for the price of the document. If a member loses a pass, the pass is registered as lost and the library will issue a new pass at no cost. If the lost pass is found, then the finder has to return it to the library. Journal issues can be lost as well. Since issues are not lent to members, this cannot be attributed to any particular reader. In general, any document can be lost without this being attributable to any particular library member.

Chapter 3

Informal specification of the DBS boundary

3.1 Library boundary

Figure 3.1 contains a part of a function decomposition tree of the Library. The nodes of this function decomposition tree represent activities that can still be decomposed further. In general, a function decomposition tree may be grown until the leaves represent the transactions of the library. The tree in figure 3.1 tree was arrived at by observing the library and by using reference models of activities that can be found in any organization [16, chapter 5]. The tree only represents activities necessary to realize the library mission; it does not represent the structure of the library. It is true that we do find some functional departments such as Finance and Member services in the library, but there are also departments within the library that correspond to scientific areas and that have their own financial unit (but not a Member services unit). These departments do not correspond with any node in the function decomposition tree.

Even though the leaves of the tree are activities and not transactions, it is informative to draw a context diagram for the nodes of this tree. Let us call the intermediate nodes in the tree of figure 3.1 *functional areas.* Figures 3.2 and 3.3 show four context diagrams, corresponding to the four functional areas of the tree. The circle in the diagram is the system we focus upon, i.c. the library. The rectangles represent *types of* systems in the environment of the the library. Note that the circle represents exactly one individual, while the rectangles all represent types. The lines in these context diagrams represent activities, that will be decomposed later into transactions. If an activity involves interaction with more than one type of external system, these types are written inside the rectangle with which the activity is connected. At this level of abstraction, we do not represent the initiative of an activity. This is because at this level of abstraction, activities usually do not have a unique actor who initiates them, so that there is no (unique) initiator to indicate.

We now zoom in on two activities, document circulation and member services, which occur in the Primary Process and in Support, respectively.

3.2 Boundary of the circulation activity

Figure 3.4 shows the decomposition of the circulation activity into transactions, and figure 3.5 shows three of the four the corresponding context diagrams. The circle and the rectangles in the context diagrams now all represent *individual systems.* This is indicated by declaring a variable inside the

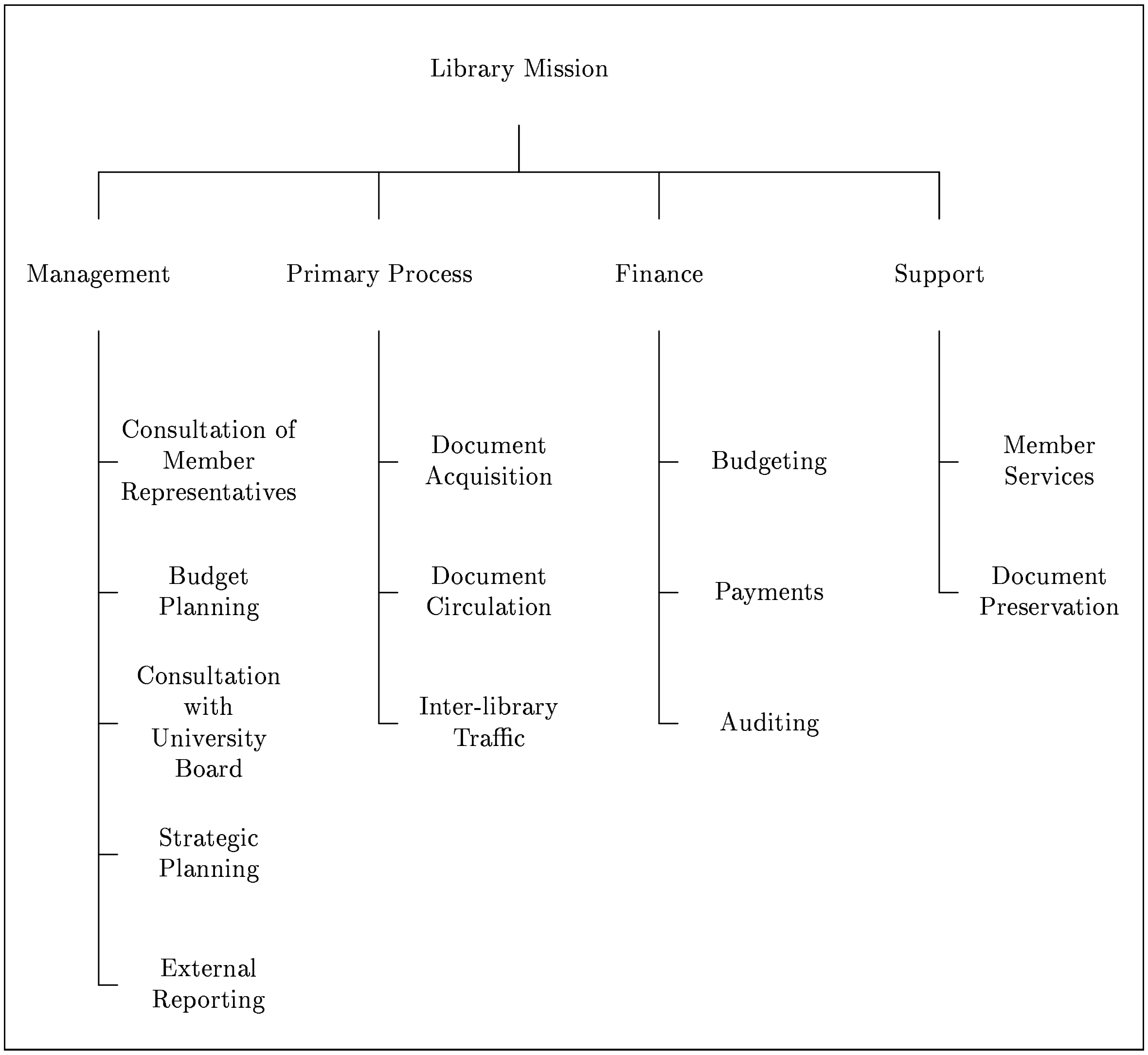


Figure 3.1: Part of a function decomposition tree of the library.

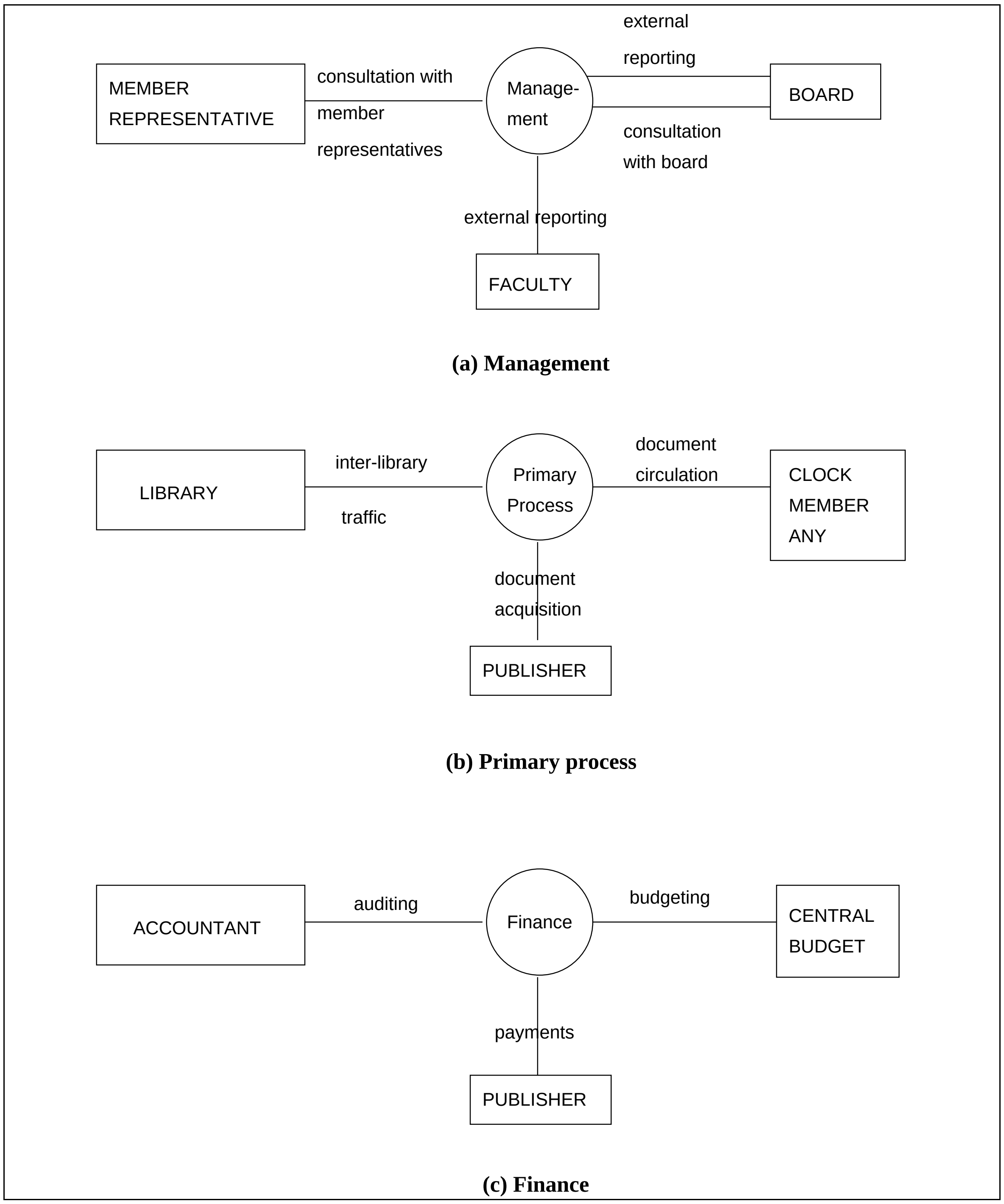


Figure 3.2: Context diagrams of Management activities, the Primary Process and Finance activities of the library function decomposition tree. The activity external reporting has an interface with two types of external entities, BOARD and FACULTY.

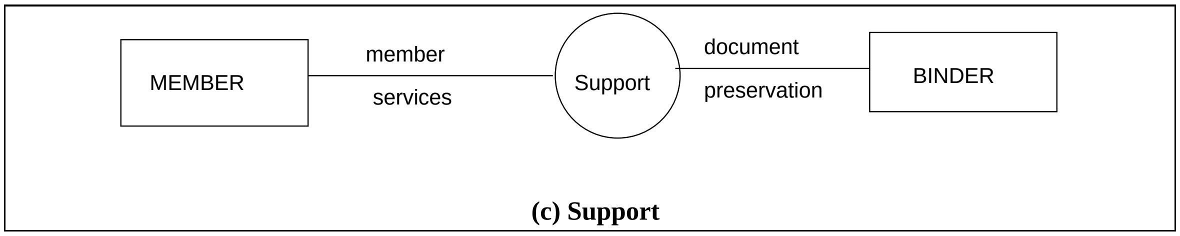


Figure 3.3: Context diagram the Support activities of the library function decomposition tree.

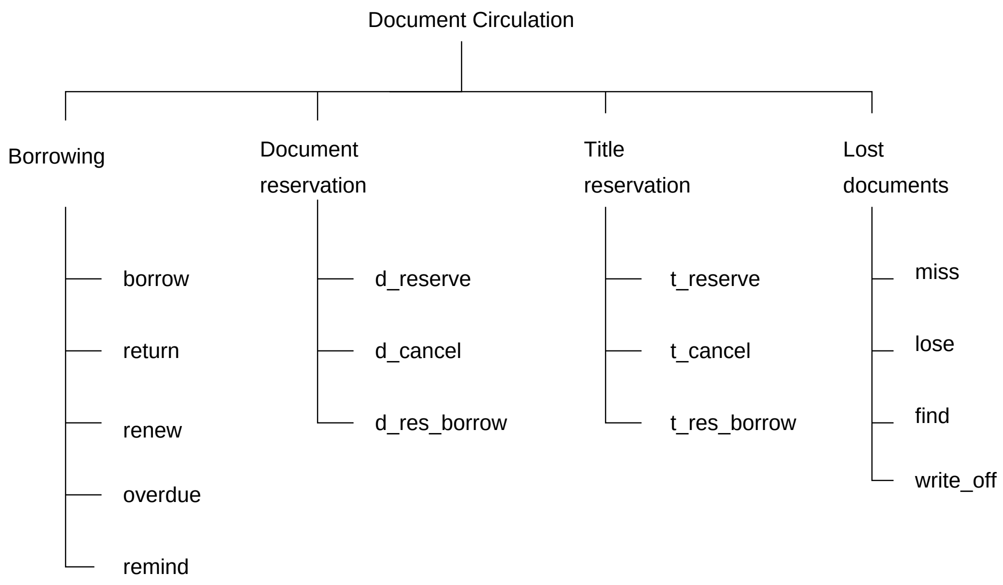
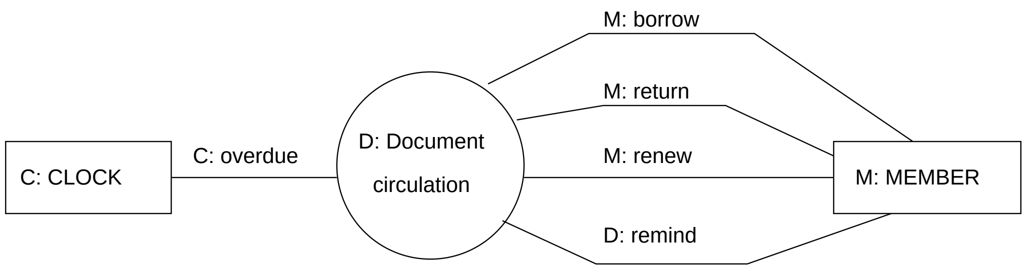
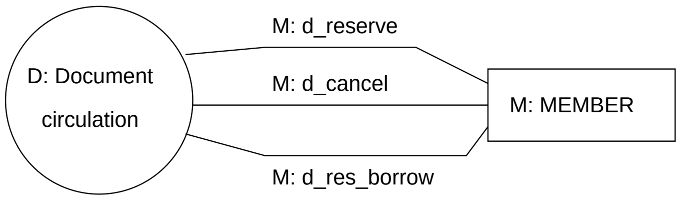


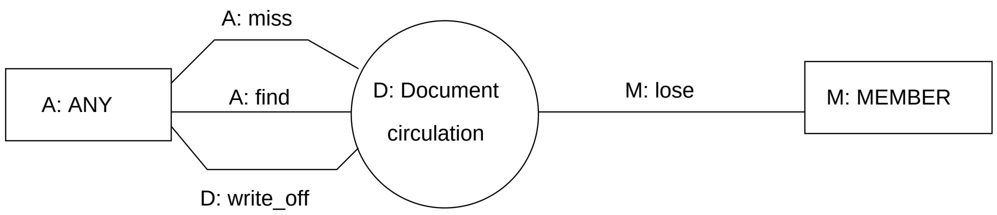
Figure 3.4: Decomposition of the circulation activities.



(a) Borrowing



(b) Document reservation



(c) Lost documents

Figure 3.5: Context diagrams of the circulation activities. The context diagram of Title Reservation is isomorphic to that of Document Reservation and has been omitted.

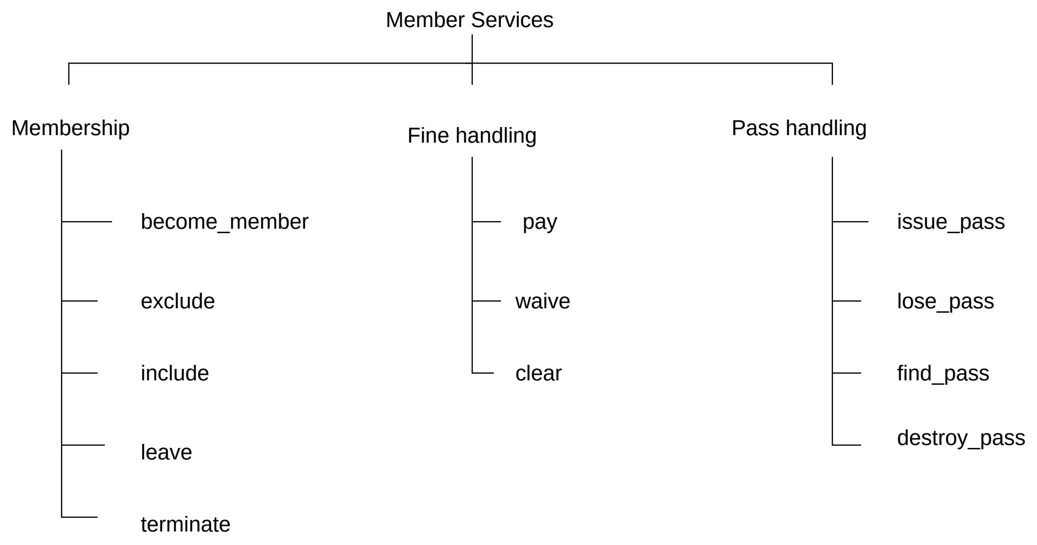


Figure 3.6: Decomposition of the membership services.

circle and in each box. It is possible that some variables have only one possible value. For example, in this case study, there is only one possible value of D, the document circulation system under study, and there is only one possible value of C, because there is only one clock. However, there are many possible values of M, because the circulation department under study has many members.

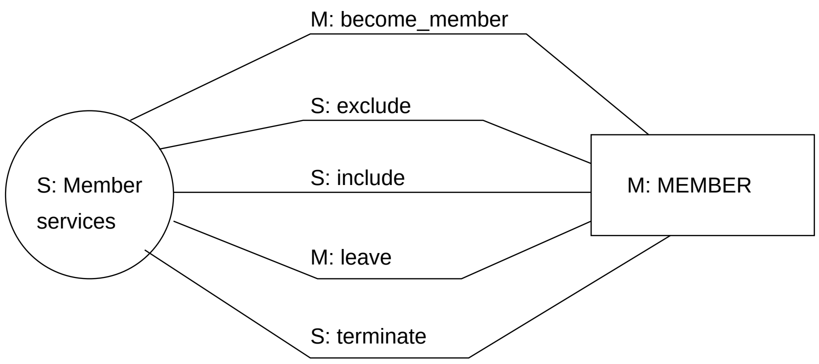
Note that in a different case study, we may very well have many instances of the Document circulation system, so that D has many different possible values. We could also add more realism by allowing many different instances of Clock, which may indicate different times. This will not be done in the present case study.

The lines in the context diagram represent transactions, i.e. atomic interactions between the library and its environment. If D is connected to system S in its environment by a line labeled S : t then this means

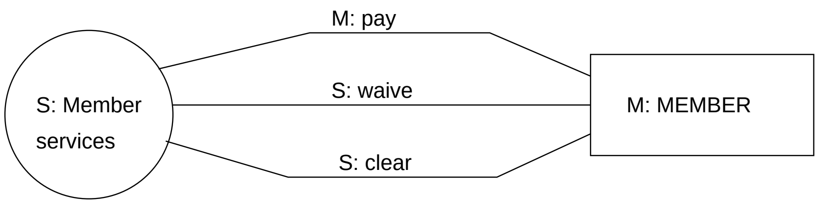
Transaction t is initiated by system S.

This does **not** mean that D and S are the only objects involved in the transaction; we will see later that t may consist of a synchronous occurrence of several local events in the life of different objects. The *contents* of the message is then that objects $o_1, \ldots, o_n$ engaged in a communication. One of the objects $o_1, \ldots, o_n$ may be S, but if S is not part of the formal UoD model, even that is not the case. What S : t means is that the transaction is a message from S to D, and what D : t means is that t is a message from D to S.

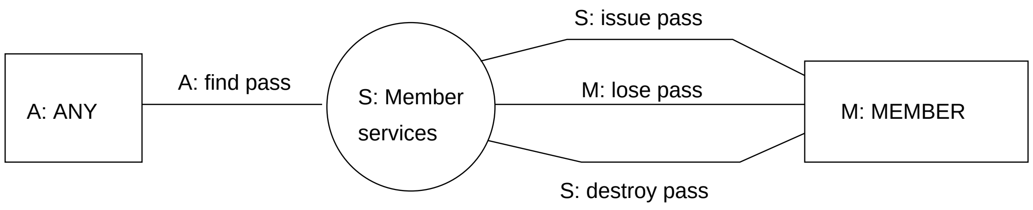
This means that the context diagrams must not be confused by the communication diagrams that are part of the UoD model. It is true that the context diagrams are communication diagrams, but they represent the communication between the library (or a subsystem of the library, like D) and its environment. By contrast, the communication diagrams used in the UoD model represent the communications in the UoD as seen by the DBS. Each communication in the UoD is observed by the DBS, and such an observation is a DBS transaction. We just saw that the initiator of the transaction need not be modeled as a partner in the observed communication. In addition, we can add that the DBS itself will never appear as an object in our UoD model. Thus, even though a context diagram is a communication diagram in which the communications are labeled by transactions, it differs from the communication diagrams used in the UoD model.



(a) Exclusion



(b) Fine handling



(c) Pass handling

Figure 3.7: Context diagrams of the member services.

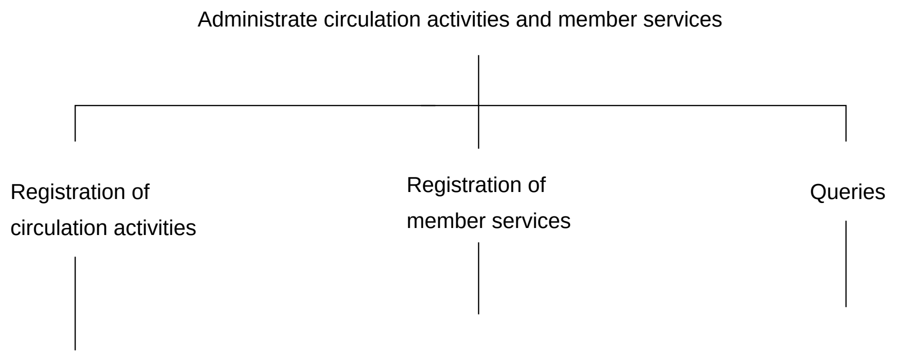

Administrate circulation activities and member services
Registration of circulation activities
Registration of member services
Queries

Figure 3.8: Top of the decomposition tree of the function of the library DBS. The Circulation and Membership Services nodes are identified with the root nodes of the corresponding decomposition trees. The Query node must still be decomposed further into individual queries.

In some cases, there is no unique type of system that initiates the transaction. For example, the transactions miss and find in figure 3.5 are not necessarily initiated by a member M. They could also be initiated by an employee of the library, by a student, or by the husband of the person who borrowed a document, etc. Since we don't want to distinguish all these cases, this is represented by the external system type ANY. The alternative would be to draw all these external system types and connect them with miss and find transactions to L. The choice for ANY has been made because adding all these external systems would not add any useful insight into the situation.

3.3 Boundary of member services

Figures 3.6 and 3.7 show the decomposition and context diagrams of membership services.

3.4 DBS boundary

The transactions in the circulation activity and member services are to be registered by the DBS. This means that we can transform the function decomposition trees and context diagrams of the library into a function decomposition tree and context diagrams for the DBS. The function of the library DBS for its environment is the following:

> To administrate the circulation activities and membership services.

This is a traditional administrative application, which contains registration and query functions. Figure 3.8 shows the top of the decomposition tree of this function. The Circulation and Membership Services nodes can be decomposed identically to the way they are decomposed earlier. The difference with the function decomposition of the *library* is that in figure 3.8, a DBS function is served, whereas in figures 3.4 and 3.6, the library mission is served. This difference is visible in the DBS function decomposition tree at most in the name of the nodes: "register borrow event" instead of "borrow" etc. This replacement is rather elaborate and we will not do this here. The Query node in the DBS function decomposition must be decomposed further by listing the relevant queries, such as "show all document loans that are overdue", "show all reservations for this title", etc. In the query model of the DBS, the desired contents of the answer to each query should be specified. We ignore the query model in this report (and in `MCM`).

Part of the context diagram corresponding to figure 3.8 is shown in figure 3.9. The initiative of

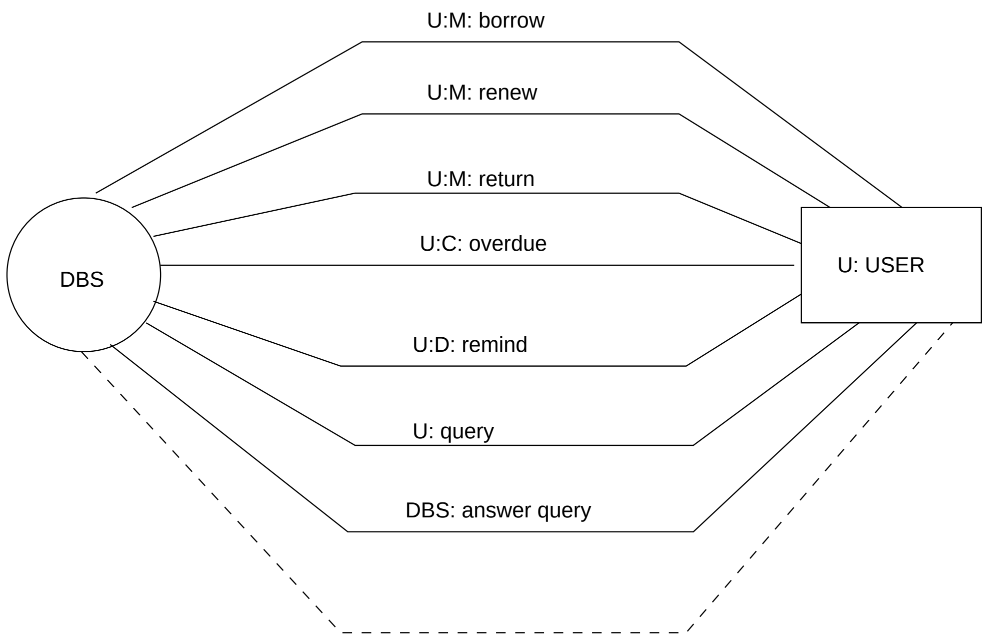


Figure 3.9: Context diagram of the library DBS. M is a MEMBER, D is the Document circulation system.

all transactions except query answering lies with the DBS (end) user U. U : M : borrow means that U registers the fact that action M : borrow occurs. The registration is initiated by U, the registered event is initiated by M.

More functionality could have been added to the DBS by improving the temporal transactions. For example, the system clock of the DBS could be used to let the DBS itself initiate the transaction overdue. Note that the type CLOCK would then have at least two existing instances, the system clock and at least one external clock. This introduces the problem of synchronizing these clocks. In this enhanced functionality, overdue would be a *temporal transaction*.

The functionality could be enhanced further by adding more temporal transactions, such as "at midnight of Sunday through Thursday print a report containing all overdue loans". Since LCM does not yet contain constructs for temporal transactions, these functions are not modeled.

Added functionality to the DBS, such as the ability to answer queries or the initation of temporal transactions, does not lead to a change in the interactions between the library and its environment. This does however lead to changes in the function decomposition trees and context diagrams of the DBS; only the implementation of these transactions changes.

Chapter 4

Informal specification of the UoD model

4.1 The class model

Figure 4.1 shows a class diagram of a part of the library UoD. Due to lack of space in this figure, no attributes or events are shown. Figures 4.2 to 4.4 give the same class diagram of the library, this time with attributes and events.

4.2 Communication model

The transaction decomposition is given in figures 4.5 to 4.14. All class names in these tables should occur in the class diagram. The set of transactions in the tables should be equal to the set of leaves of the DBS function decomposition. We show a local event in the table only in the row corresponding to its most general class. Inheritance of the local event by more specific classes is not shown but is implied by the tables. This is analogous to the declaration of events in `LCM`, which is only done for the most general class to all of whose instances the event is applicable. The set of local events allocated to a class C in these tables should be equal to the set of events declared in the specification of C in the formal `LCM` specification of the UoD model.

If a transaction t consists of a communication of two instances of C with each other, then the two local events by which this is done are both shown in the corresponding entry of the table (at the intersection of C and t).

It is interesting to observe that the transaction decomposition tables are visually more pleasing and are easier to understand than the communication diagrams. The transaction decomposition tables are actually the heart of the model. Assuming that we have given intuitively clear names to the local events and to the transactions, they almost explain the meaning of the transaction to the domain specialist. Together with the class diagram and the life cycles, they give a very good insight into the structure of the conceptual model.

4.2.1 Circulation activities

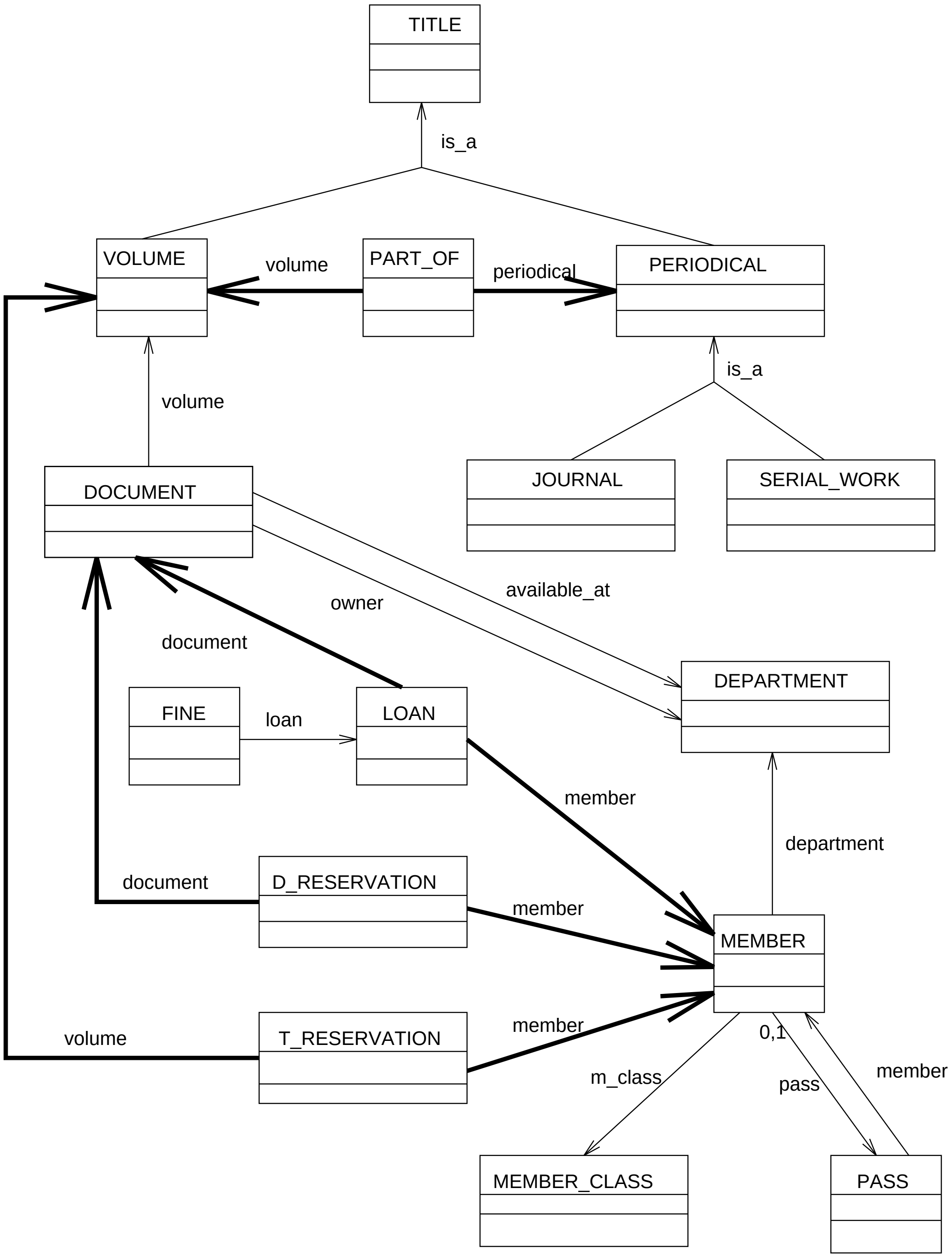


Figure 4.1: Class diagram of the UoD of the circulation activities and member services, without any attributes or events indicated.

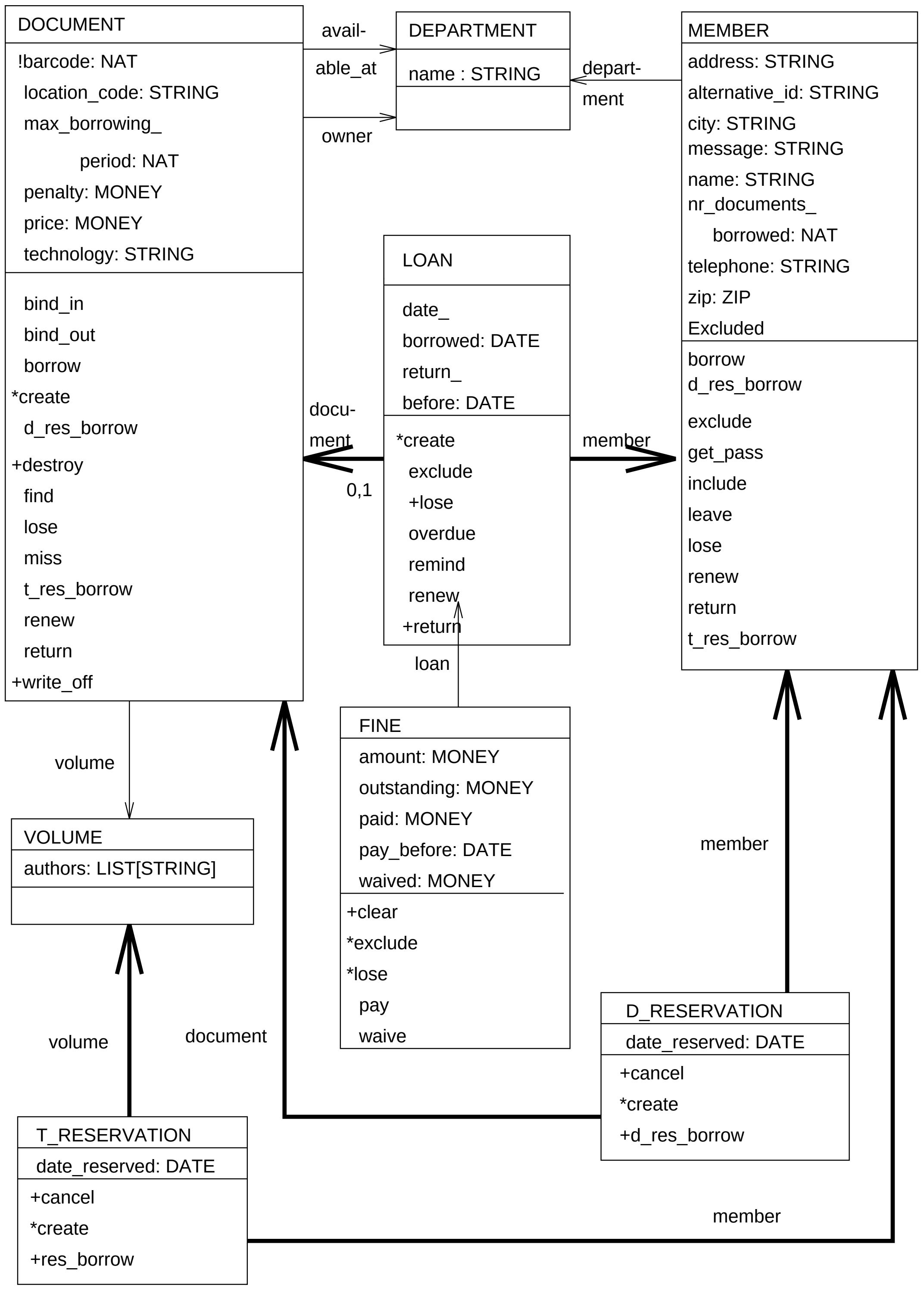


Figure 4.2: Class diagram of the UoD of the circulation activities and member services — part 1.

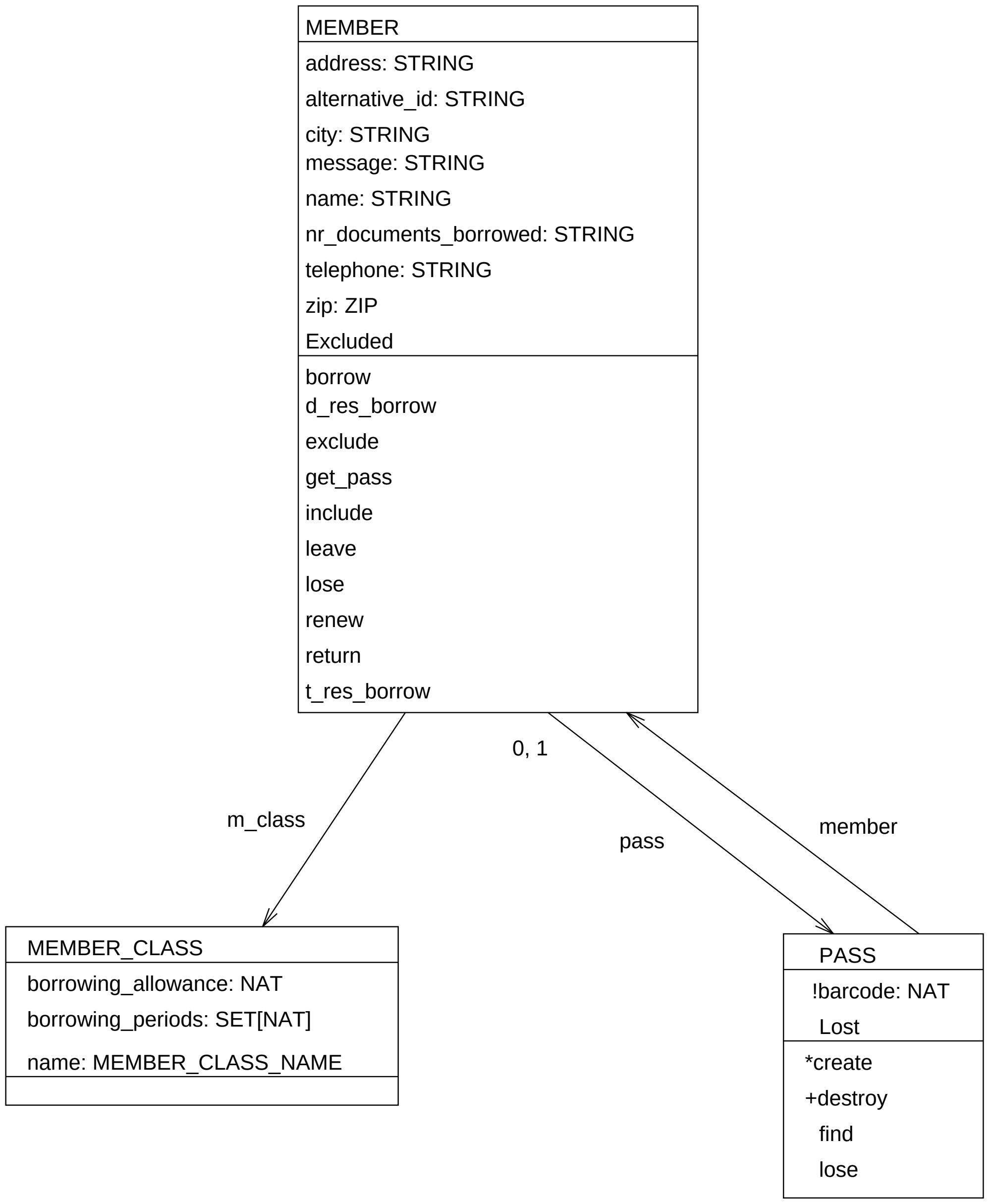


Figure 4.3: Class diagram of the UoD of the circulation activities and member services — part 2.

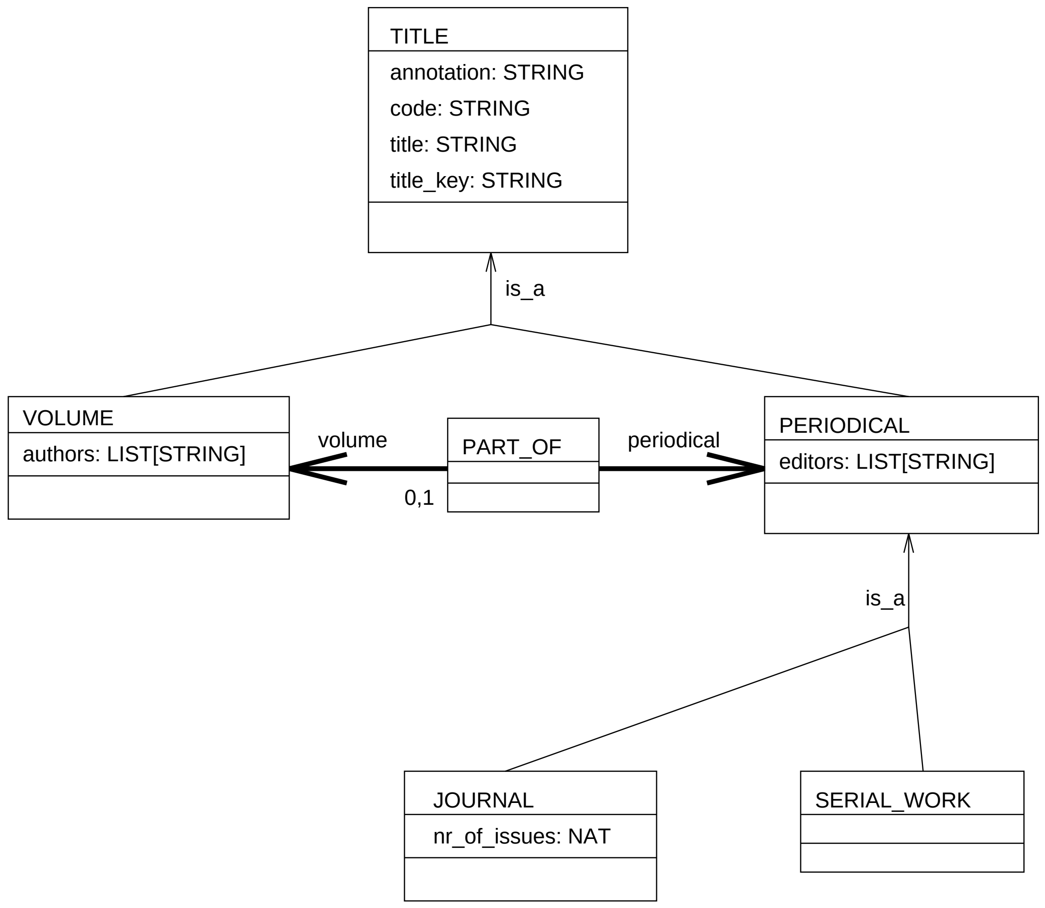


Figure 4.4: Class diagram of the UoD of the circulation activities and member services — part 3.

Borrowing

	borrow	return	renew	overdue	remind
LOAN	create	return	renew	overdue	remind
DOCUMENT	borrow	return	renew		
MEMBER	borrow	return	renew		

Figure 4.5: Transaction decomposition table for the Borrowing service.

LostDocumentHandling

	miss	lose	find	write_off
LOAN		lose		
DOCUMENT	lose	lose	find	write_off
FINE		lose		
MEMBER		lose_copy		

Figure 4.6: Transaction decomposition table for the Lost Documents service.

	TitleReservation			DocumentReservation		
	t_reserve	t_cancel	t_res_ borrow	d_reserve	d_cancel	d_res_ borrow
LOAN			create			create
DOCUMENT			t_res_ borrow			d_res_ borrow
MEMBER			t_res_ borrow			d_res_ borrow
D_RESERVATION				create	cancel	res_ borrow
T_RESERVATION	create	cancel	res_ borrow			

Figure 4.7: Transaction decomposition table for the DocumentReservation and TitleReservation services.

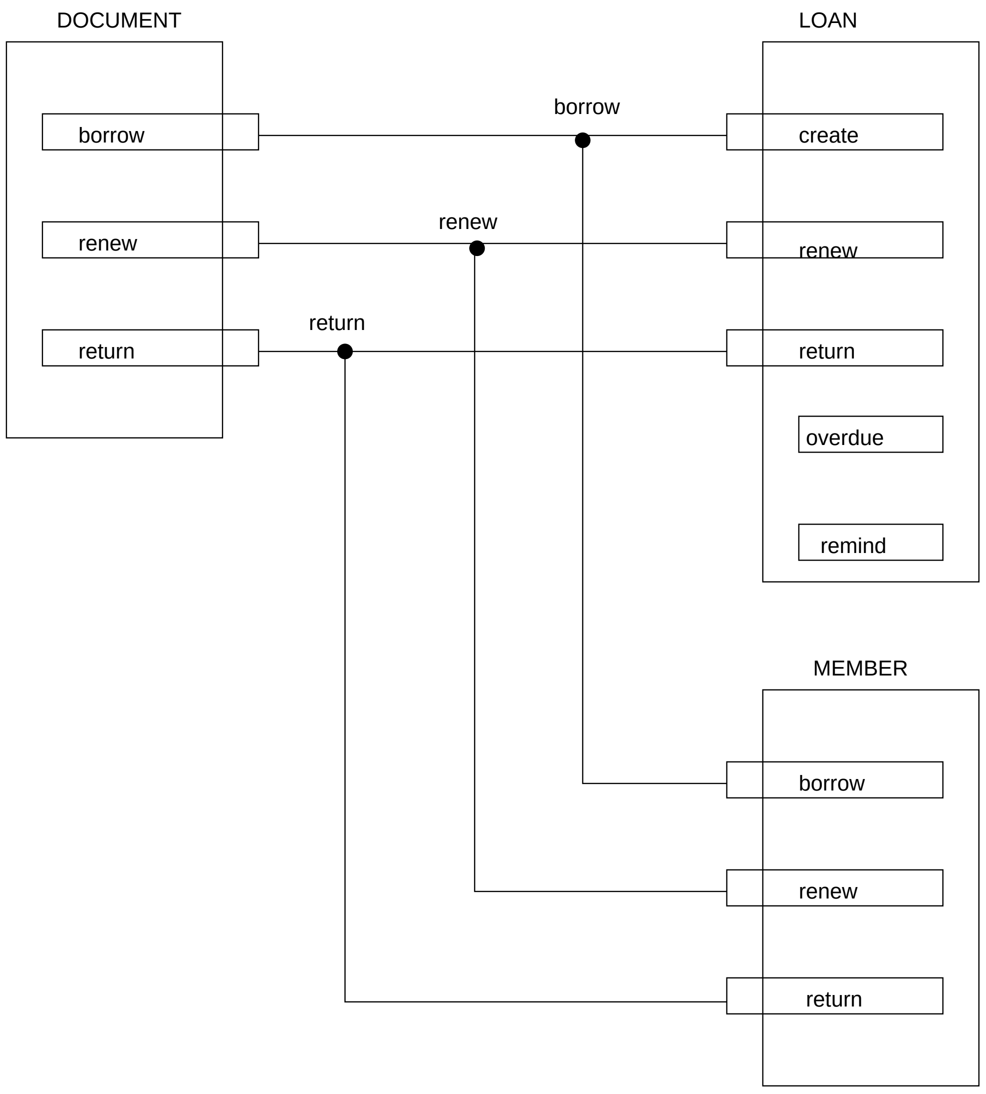


Figure 4.8: Communication diagram of the Borrowing service.

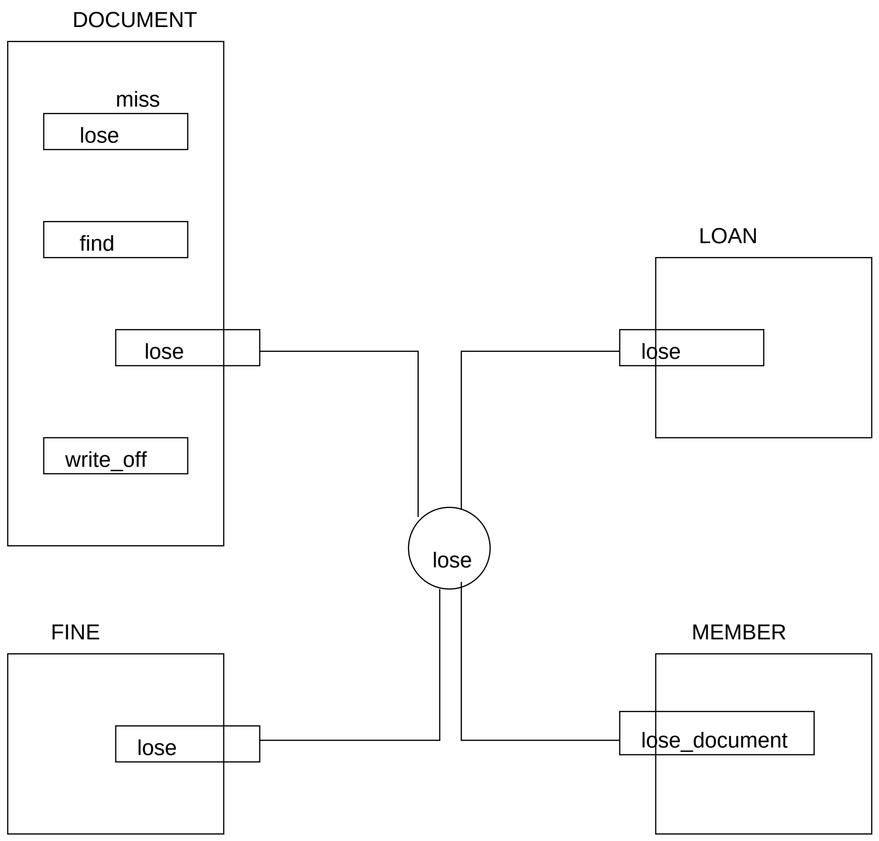


Figure 4.9: Communication diagram of the Lost Document Handling service.

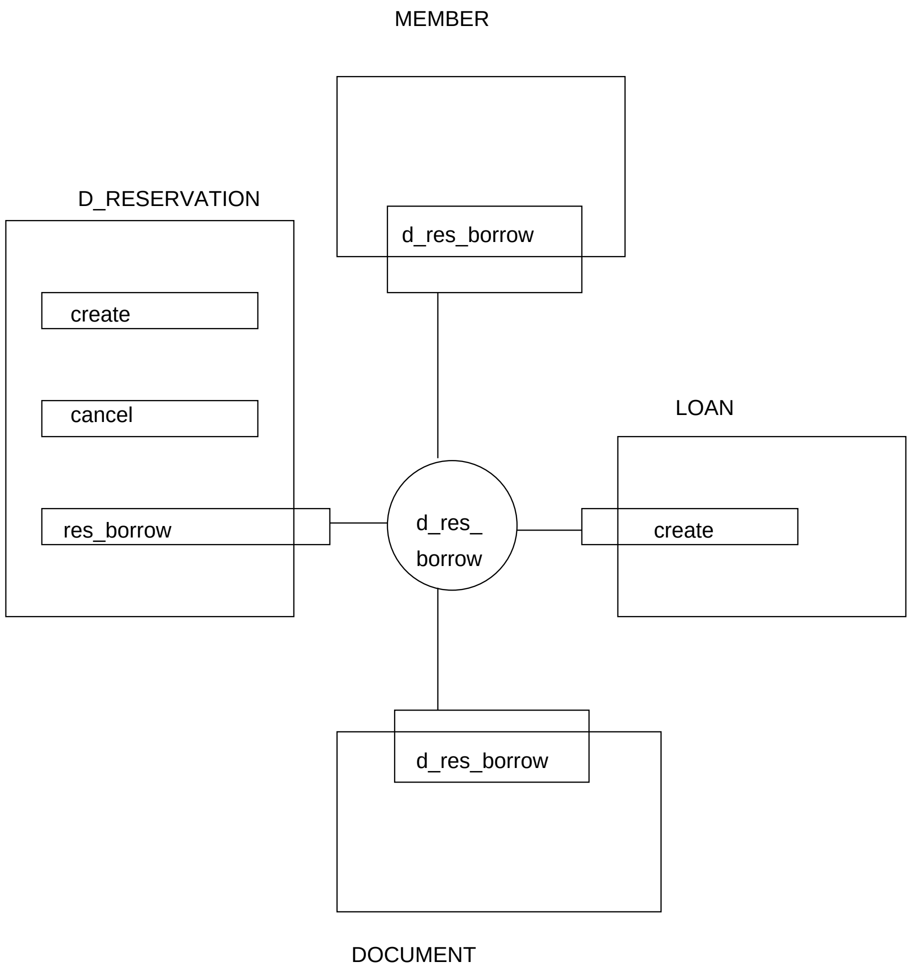


Figure 4.10: Communication diagram of the CopyReservation service.

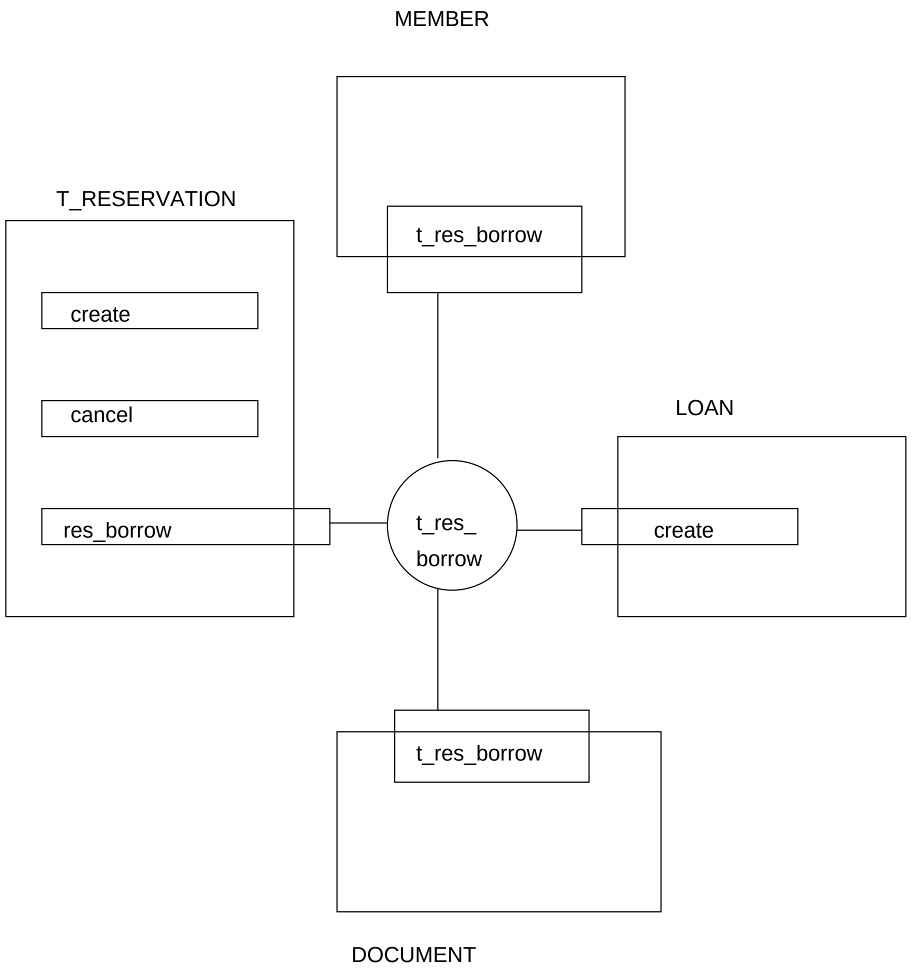


Figure 4.11: Communication diagram of the TitleReservation service.

Membership

	become_member	exclude	include	leave	terminate
LOAN		exclude			
FINE		exclude			
MEMBER	create	exclude	include	leave	terminate
PASS	create				destroy

Figure 4.12: Transaction decomposition table for the Membership service.

FineHandling

	pay	waive	clear
FINE	pay	waive	clear

Figure 4.13: Transaction decomposition table for the FineHandling service.

4.2.2 Member services

Passhandling

	issue_pass	lose_pass	find_pass	destroy_pass
MEMBER	get_pass			
PASS	create	lose	find	destroy

Figure 4.14: Transaction decomposition table for the PassHandling service.

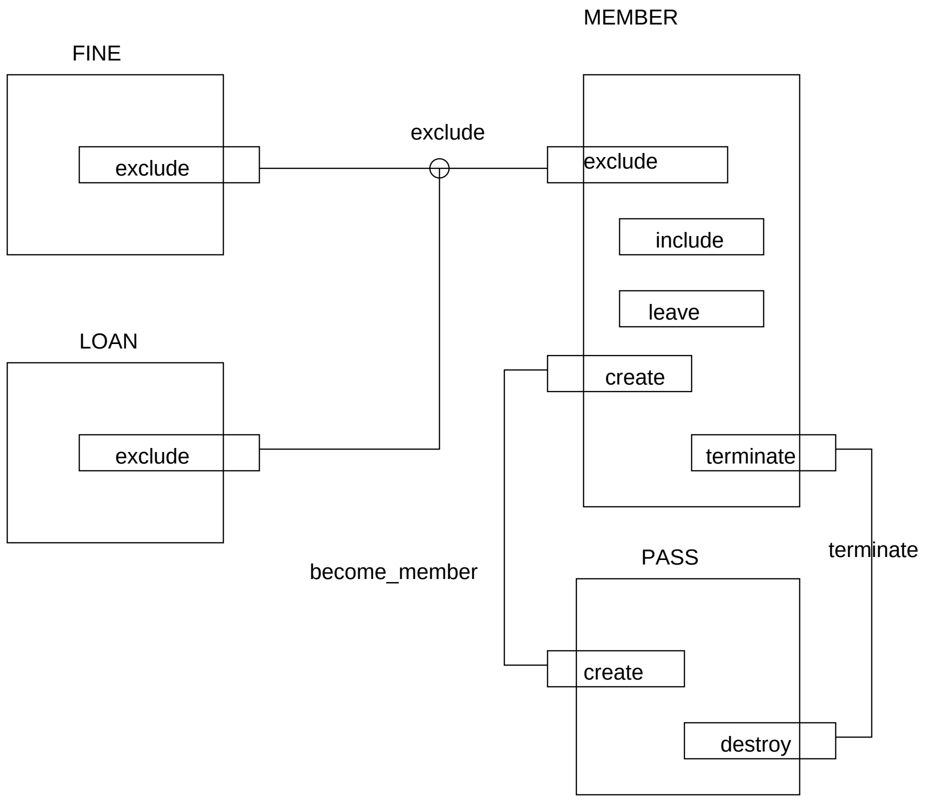


Figure 4.15: Communication diagram of the Membership service.

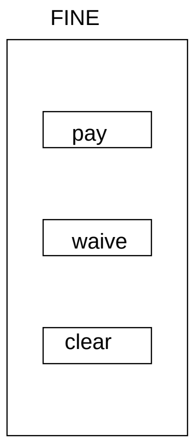


Figure 4.16: Communication diagram of the Fine Handling service.

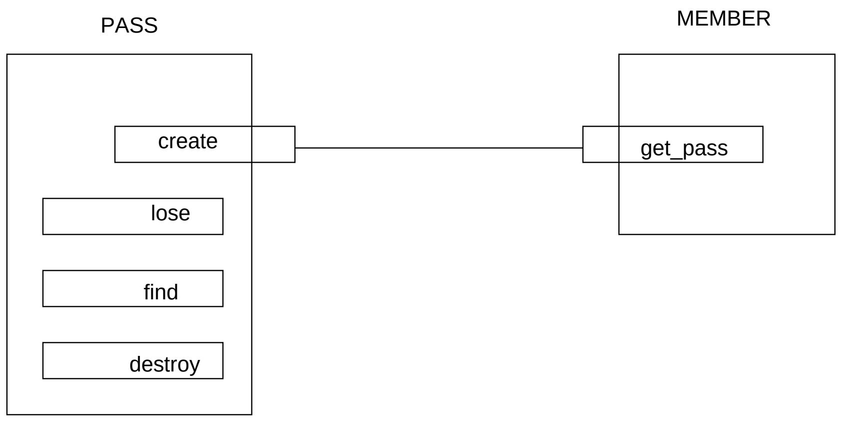


Figure 4.17: Communication diagram of the Pass Handling service.

4.3 Life cycle model

Figures 4.18 to 4.23 show the life cycles of objects in the UoD. Life cycles not shown have no significant structure, i.e. after creation any non-creation event can happen at any moment. Each life cycle is defined for a class of objects and is followed by all existing instances of the class. Different instances will in general be at different stages of their life cycles. The set of events in the life cycle for instances of C must be equal to the set of events declared for C in the class diagram; this is the set of events written in the box for C in the class diagram, plus all the events declared for superclasses of C. The life cycle must be the solution of the life cycle specification for instances of C in the `LCM` specification.

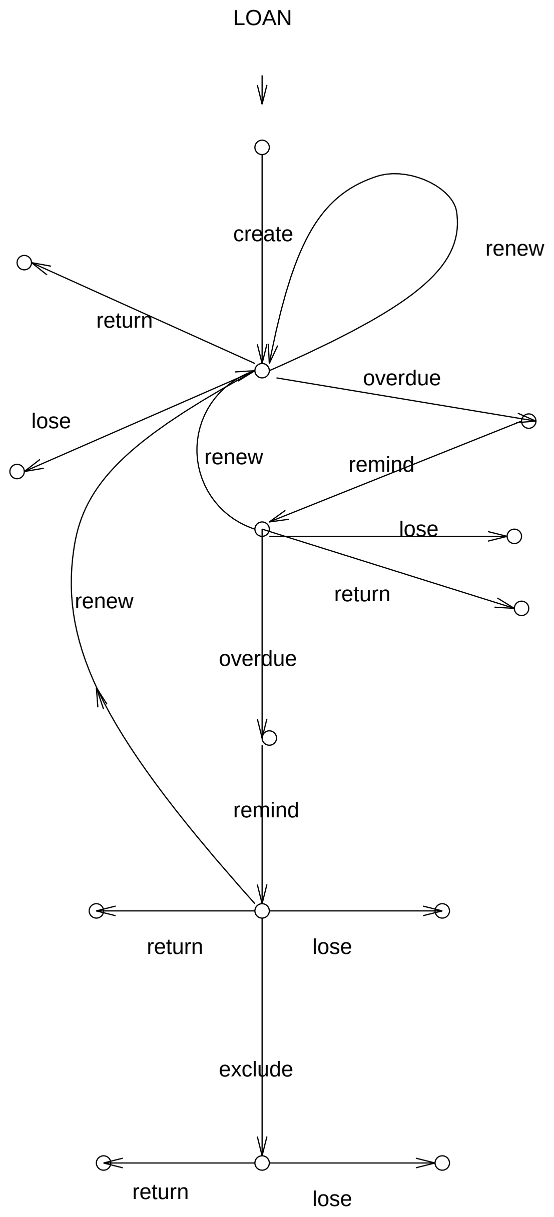


Figure 4.18: Life cycle of *LOAN* instances.

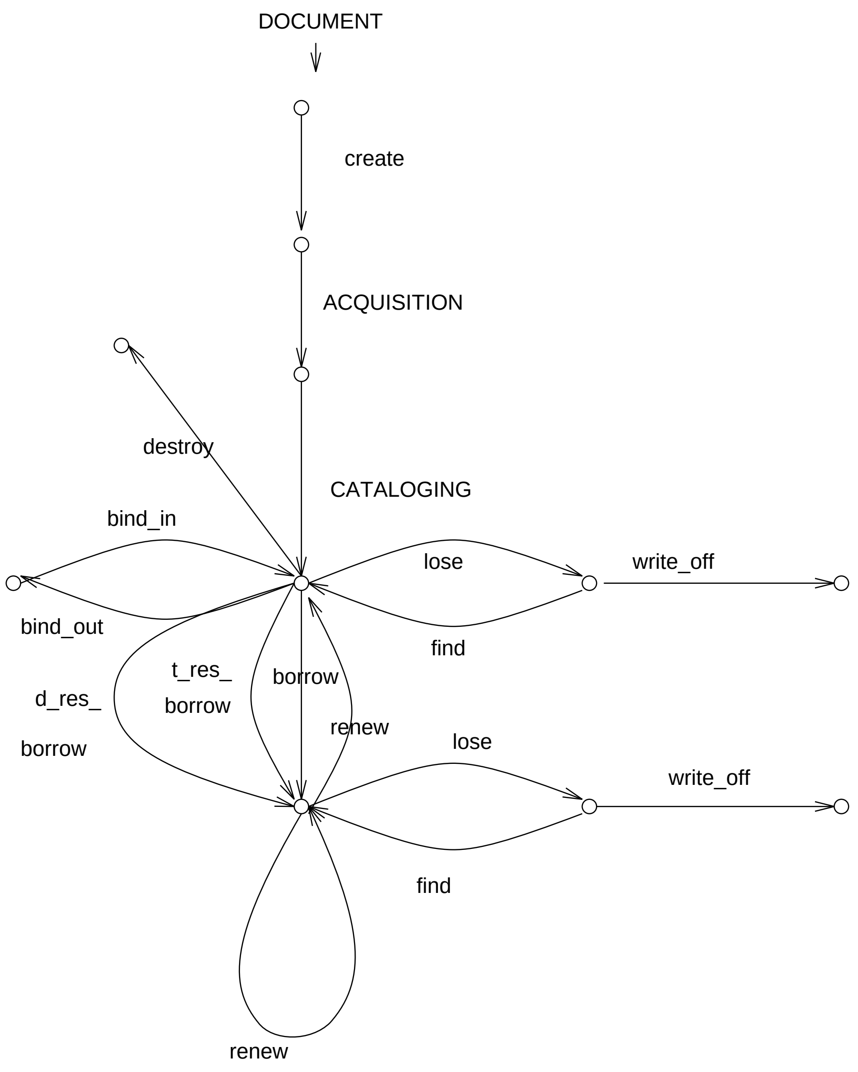


Figure 4.19: Life cycle of $DOCUMENT$ instances. The ACQUISITION and CATALOGING processes are not specified here. The destroy event is part of the acquisition function and not specified in this report.

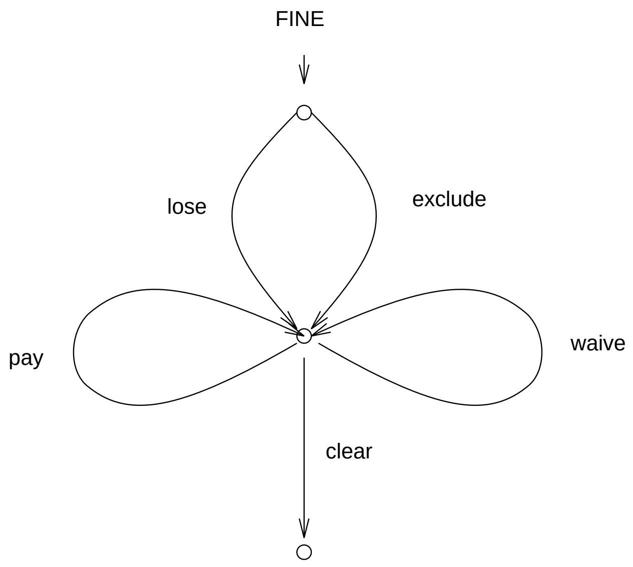


Figure 4.20: Life cycle of $FINE$ instances.

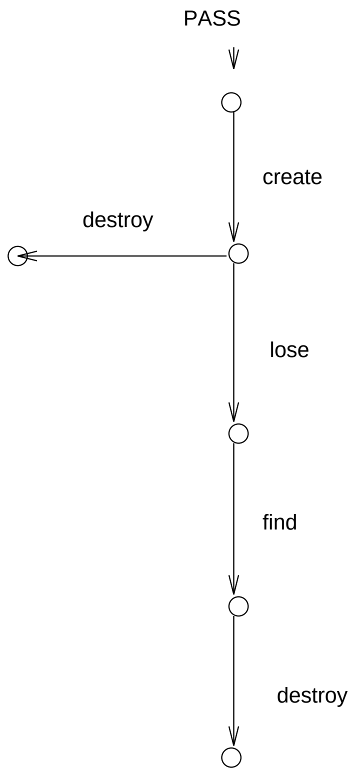


Figure 4.21: Life cycle of $PASS$ instances.

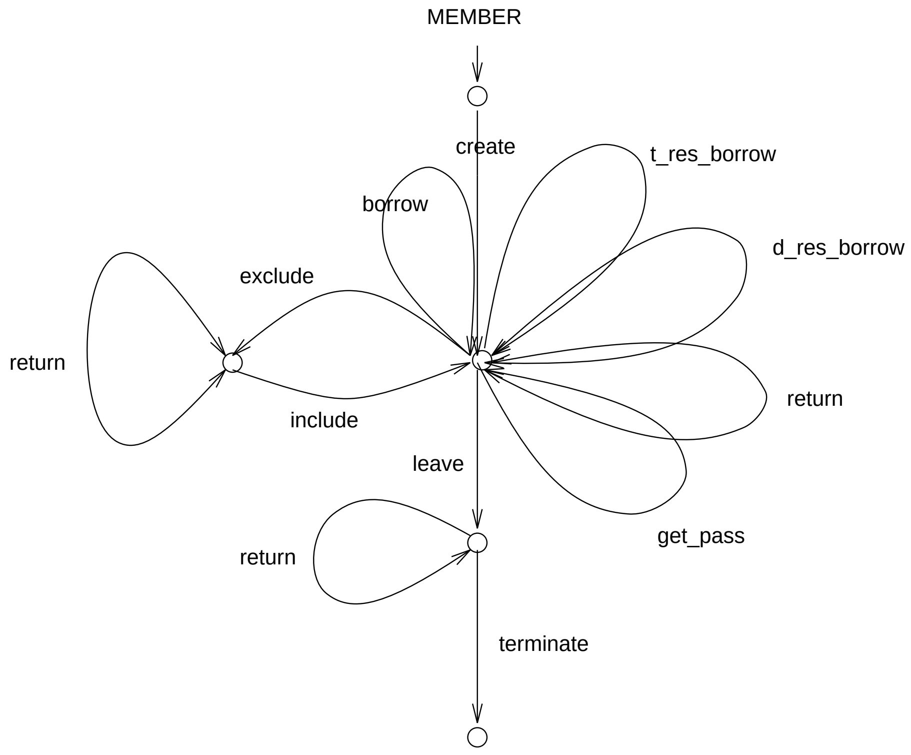


Figure 4.22: Life cycle of $MEMBER$ instances.

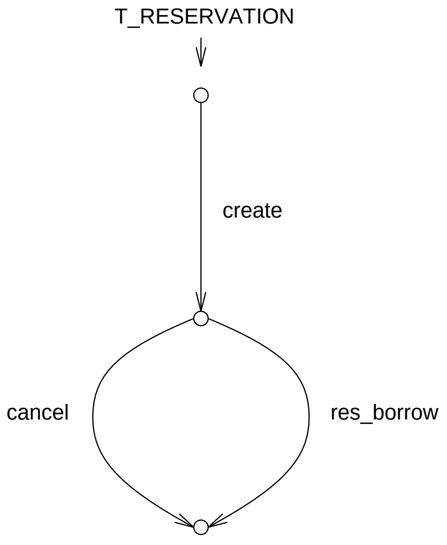


Figure 4.23: Life cycle of $T\_RESERVATION$ instances.

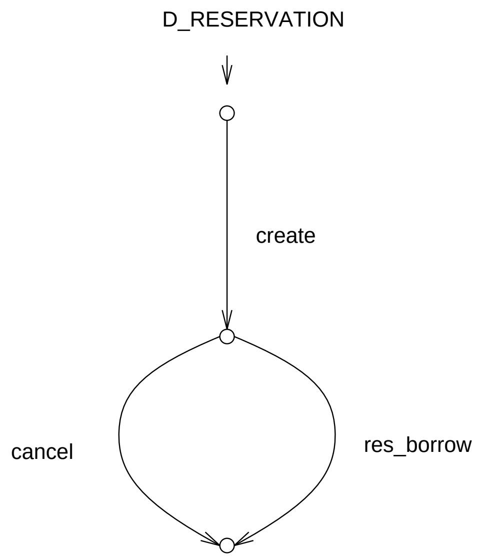


Figure 4.24: Life cycle of $D\_RESERVATION$ instances.

4.4 Class dictionary

The dictionary entries are listed in alphabetical order.

4.4.1 DEPARTMENT

Class name: *DEPARTMENT*

Class definition:
An organizational unit of the university in which scientific research is done and which is the smallest unit of budgetary allocation for the university board.

Identity:
• The identity criterion for departments is determined by the university board.

Attributes:	
name	Name of the department.

4.4.2 DOCUMENT

Class name: *DOCUMENT*

Class definition:
A *DOCUMENT* is a carrier of information that is of use to library members. The library exists to make *DOCUMENT* instances available to library members. Each *DOCUMENT* is a document of a *VOLUME*, which itself is a kind of *TITLE*. See these two classes for further descriptions.

Identity:
A *DOCUMENT* instance is identified by a barcode.

Attributes:	
available_at	The department where the document can be borrowed.
barcode	Unique identification of the document issued by the library. Natural number consisting of 14 digits, built up as follows: • First digit: 3 (indicating that a document is identified) • Digits 2–5: identification of the department that owns the document • Digits 6–13: unique identification of the document within this department • Digit 14: checksum
location_code	The location in the store room where the document is stored when it is not borrowed or out to the binder.
owner	The department out of whose budget the document was paid for by the library.
max_borrowing_ period	The maximal number of days that the document can be borrowed. This may be • 0 (the document can only be read in the reading room) • 1 (the document must be returned the day it is borrowed) • 3 weeks • unrestricted
penalty	The price to be paid by a member who loses this document.
price	The price paid by the library to acquire this document.
volume	The volume of which this document is an instance.
technology	The reading, storage or writing technology of the document. This may be printed paper, hand-written manuscript, microfilm, microfiche, movie, video, photo, fine print, or globe.

Predicates:	
Available	The document is available for borrowing.

Events:	
bind_in	A document returns from the binder.
bind_out	A document goes to the binder.
borrow	A document is borrowed by a library member.
create	A document is ordered or is received as a gift.
destroy	A document that is not lent, missing or lost is destroyed. This deletes the document instance.
d_res_borrow	A document is borrowed by a library member who reserved it.
find	A missing or lost document is found.
lose	It is discovered that the actual whereabouts of a document is unknown.
renew	The borrowing period of a borrowed document is extended.
return	A document is returned by a library member.
t_res_borrow	A document is borrowed by a library member who reserved the document title.
write_off	A lost or missing document is written off. This deletes the document instance.

Business rules:

- A document that has an allowable borrowing period of 0 days is not borrowable (axioms 8, 9, 10).
- A document can only engage in a *borrow* event only if there are no title– or document reservations applicable to the document (axioms 11, 12).
- A document can only engage in a *t_res_borrow* event only if there are no document reservations applicable to the document (axiom 13).
- A document can only be borrowed by a *t_res_borrow* event if it is an instance of a volume for which a reservation is outstanding (axiom 14). The member borrowing the document is then the member who placed the reservation (see the decomposition of *t_res_borrow* in the *Reservation* service).
- A borrowing period for a document can only be renewed when there are no title– or document reservations applicable to the document (axiom 15, 16).

4.4.3 D_RESERVATIONS

Class name: *D_RESERVATION*

Class definition:
A *D_RESERVATION* represents a document reservation. It is a relationship between a member and the particular document that is reserved.

Identity:

- The identity of a *D_RESERVATION* relationship consists of the identity of the reserved document and of the member who reserved the title.

Components:	
document	The reserved document.
member	The member who reserved the document.

Attributes:	
date_reserved	The date at which the reservation is created.

Events:	
cancel	The reservation is terminated either because the member cancels it or because the deadline before which the reserver could collect the document has passed. Destroys the *D_RESERVATION* relationship.
create	Reserve a document.
d_res_borrow	The reserver borrows the reserved document. Destroys the *D_RESERVATION* relationship.

Business rules:

- A member can reserve any number of copies and a document can have any number of reservers. The reservers are served in order of reservation date.
- A reserved document that is returned can only be borrowed by a member who has a reservation not younger than any other reservation for that document (axiom 1).
- A document can only be reserved if it is not available (axiom 2).
- A member cannot place two or more title reservations for a document simultaneously (axiom 3).

4.4.4 FINE

Class name: *FINE*

Class definition:
A *FINE* is an obligation on a library member to pay for undesirable behavior caused to the library. Fines may be created by losing a document owned by the library or by failing to return a document after two reminders for returning it are sent to the member by the library.

Identity:
• Each instance of an undesirable event creates a different fine. Different instances of *exclude* create different fines and different instance of *lose* create different fines.

Attributes:	
amount	The total amount to be paid.
outstanding	The amount still to be paid.
borrow_process	The *LOAN* instance that contains the reason for the fine (not responding to reminders or losing a document).
paid	The amount already paid.
pay_before	The deadline before which payment for the total *amount* must be received by the library.
waived	The amount waived by the library.

Events:	
clear	The *FINE* is deleted.
exclude	A *FINE* is created because the member does not respond to two reminders in sequence for the same *LOAN* process.
lose	A *FINE* is created because a member loses a document borrowed by him or her.
pay	A member pays (part) of the amount fined.
waive	The library waives the obligation on the member to pay (part of) the amount fined.

Business rules:

- The fine for not responding to two consecutive reminders in the same borrowing process is 70 (axiom 1).
- The fine for losing a borrowed document is the *penalty* associated with the document (axiom 3).
- A fine cannot be deleted unless there is no amount outstanding (axiom 6). A fine can be created with an amount of 0, though, if the penalty associated with a document is 0.

4.4.5 JOURNAL

Class name: *JOURNAL*

Class definition:
A title published as volumes, which consist of single issues published at regular intervals.

Identity:
• Same as identity criterion for *PERIODICAL*.

Attributes:	
nr_of_issues	Number of issues per volume.

4.4.6 LOAN

Class name: *LOAN*

Class definition:
A *LOAN* instance is a relationship between a *DOCUMENT* and a *MEMBER* that represents the fact that the member has borrowed the document.

Identity:
LOAN instances are relationships, so the identifier of a *LOAN* instance consists of a tuple of two identifiers, that of a *DOCUMENT* and a *MEMBER*. This means that one *LOAN* instance can occur several times. Each time member *m* borrows document *c*, the same *LOAN* instance is created. Thus, a *LOAN* instance is really an event; one and the same *borrow* event can occur many different times. If we would have a historical database, the different occurrences of one *LOAN* instance would be distinguished by their *borrowing_date*.

Components:	
document	The document being borrowed.
member	The member who borrows the document.

Attributes:	
date_borrowed	The date at which the borrow event takes place.
return_before	The document must have been returned when this date passes. This defines a temporal event.

Events:	
create	A *LOAN* instance is created.
exclude	The member *member*(*b*) is excluded from further use of the library and a fine is created for *member*(*b*). The member can only be included when all documents are returned and all fines paid.
lose	The member loses the borrowed document. The *LOAN* instance ceases to exist and a fine is created.
overdue	The member is overdue with returning the document or renewing the *LOAN* period.
remind	The member of the loan is sent a reminder.
renew	The *LOAN* period is extended.
return	The borrowed document is returned. The *LOAN* instance ceases to exist.

Business rules:
• A document can only be borrowed for a period that is allowed to the member. The permissible borrowing periods are represented by the attribute *borrowing_periods* of *MEMBER_CLASS* (axioms 2 and 3).

4.4.7 MEMBER

Class name: *MEMBER*

Class definition:

A member is a person, a group of persons, or an institution who or which has permission to use the library. If the member is a person, it is either a staff member, a student, or external.

Identity:

- Members are identified by their current pass.
- A person who leaves the library and thereafter becomes member again is two different members.
- A person who is a member but is also a member of a group that is a member, is still one member; he or she is a member as a person and identified as such.

Attributes:	
address	The street name and number where the member may be reached.
alternative_id	Passport or drivers license number.
city	City of the address where the member can be reached.
class	Class of the member. Used to get the allowed borrowing periods for members of this class.
department	Department where the member is registered. *null* when not applicable.
message	Message to be given to the member when he or she next comes at the circulation desk. For example "You forgot your umbrella."
name	Name of the member.
nr_documents_borrowed	Number of documents currently borrowed by the member.
pass	Current pass of the member.
telephone	Telephone number where the member can be reached.
zip	Zip code of the address of the member.

Events:	
borrow	A member borrows a document.
d_res_borrow	A member borrows a document that he or she reserved.
exclude	A member is excluded from further use of the library until he or she returns all documents and pays all fines.
get_pass	A member gets a pass.
include	An excluded member is again allowed to use the library.
leave	A member leaves the library, but may still return documents.
lose_document	A member loses a document borrowed by him or her.
renew	A member extends the borrowing period of a document borrowed by him or her.
return	A member returns a document borrowed by him or her.
terminate	A member terminates his or here membership.
t_res_borrow	A member borrows a document whose title he or she reserved.

Business rules:

- Members have a set of allowed borrowing periods that is determined by their class. (Modeled by the *class* attribute.)
- An excluded member cannot borrow a document or extend a borrowing period (life cycle definition).
- A member can only be included if he or she has no copies borrowed and all his or her fines are paid (axiom 3).
- A member cannot borrow more than the allowance granted to his or her member class (axioms 4–9).
- A member has one and only one active pass (axiom 10).

4.4.8 MEMBER_CLASS

Class name: *MEMBER_CLASS*

Class definition:
Each library member is classified as one of the member classes.

Identity:
• There are exactly four different member classes, External, Group, Staff and Student.

Attributes:	
borrowing_allowance	The maximum number of books that a member of this class can borrow simultaneously.
borrowing_periods	The borrowing periods (number of working days) allowed to members of this class.
name	The name of the member class.

Business rules:

- Staff is allowed to borrow up to 20 copies simultaneously (axiom 2).
- Non-staff is allowed to borrow up to 10 copies simultaneously (axiom 3).
- Staff is allowed a borrowing period of 15 working days or 360 working days (axiom 4).
- Non-staff is allowed a borrowing period of 15 working days (axiom 5).

4.4.9 PART_OF

Class name: *PART_OF*

Class definition:
Relationship between a volume and the periodical of which it is a part. A volume need not be part of a periodical but if it is, it is related through a *PART_OF* relationship.

Identity:
• The identity of a *PART_OF* relationship consists of the identities of the participating volume and periodical.

Components:	
volume	The volume part of a periodical
periodical	The periodical containing the volumes.

4.4.10 PASS

Class name: *PASS*

Class definition:
A pass is a small card carrying the name and identification number of the member.

Identity:
• A pass is identified by its barcode.

Attributes:	
barcode	Unique identification of the pass, issued by the library. Number of 14 digits starting with a 2 and built up in the same way as the *barcode* of a *DOCUMENT*.
member	The member to whom the pass is assigned.

Predicates:	
Lost	*true* when the pass is lost, *false* otherwise.

Events:	
create	A new barcode is generated and a pass created with this barcode. The generated barcode has never been generated before.
destroy	The pass is destroyed.
find	The pass is found.
lose	The pass is lost.

Business rules:
• A pass is issued to new members or to an existing member who has lost his or her pass. • A barcode, once generated, is never reused. • A member can have several passes, but then all but one of these passes in the *Lost* state. This pass is the current identification of the member.

4.4.11 PERIODICAL

Class name: *PERIODICAL*

Class definition:
A title of which different instances are published at regular intervals. Periodicals are either journals or serial works like *Spinger Lecture Notes in Computer Science*.

Identity:
• The identity of a periodical is ascertained by the founders of the periodical, who usually are the first editors. These may declare the periodical a reincarnation of a previously published periodical which went out of print.

Attributes:	
editors	The editors of the periodical.

4.4.12 SERIAL_WORK

Class name: *SERIAL_WORK*

Class definition:

A title consisting of books (volumes), published at regular intervals.

Identity:

- Same as those for *PERIODICAL*.

4.4.13 TITLE

Class name: *TITLE*

Class definition:

A *TITLE* is a unique set of bibliographical data, such as appears in a bibliography at the end of a paper or book. A *TITLE* may have more than one *DOCUMENT* and it may also be a multi-volume work, a *PERIODICAL* or a journal. Each volume in a multi-volume work is itself also a *TITLE*.

Identity:

- Different editions of a book count as different titles.
- Hard- and paperback printings of one title count as one *TITLE*, even if they have different ISBN numbers.
- A multi-volume *TITLE* consisting of n *VOLUME*s counts as $n + 1$ *TITLE*s: n volumes plus the whole series itself.
- A *JOURNAL* counts as one title, even if it disappears and returns later with the suffix "n.s." (new series).
- Etcetera, as determined by the librarian.

Attributes:	
annotation	Any comment.
code	Systematic code.
title	The name of the *TITLE*. Examples:"ACM Transactions on Database Systems", "The Specification of Computer Programs", "An Introduction to Database Systems, Volume 1, Fourth Edition", "Deductive and Object-Oriented Databases. Second International Conference, DOOD'91".
title_key	The first three letters of the name of the first author, followed by the first letter of each of the first four words of the title, followed by two digits, followed by a number that identifies the volume in a series.

4.4.14 T_RESERVATION

Class name: *T_RESERVATION*

Class definition:
A *T_RESERVATION* represents a title reservation. It is a relationship between a member and a title which is reserved.

Identity:
• The identity of a *T_RESERVATION* relationship consists of the identity of the reserved title and the member who reserved the title.

Components:	
volume	The reserved title.
member	The member who reserved the title.

Attributes:	
date_reserved	The date at which the reservation is created.

Events:	
cancel	The reservation is terminated either because the member cancels it or because the deadline before which the reserver could collect a document of the reserved title has passed. Destroys the *T_RESERVATION* relationship.
create	Reserve a title.
res_borrow	The reserver borrows a document of the reserved title. Destroys the *T_RESERVATION* relationship.

Business rules:

- A member can reserve any number of titles and a title can have any number of reservers. The reservers are served in order of reservation date.
- Only titles that are volumes can be reserved.
- A document of a reserved volume that is returned can only be borrowed by a member who has a reservation not younger than any other reservation for that volume (axiom 1).
- A volume can only be reserved is none of its copies is available (axiom 2).
- A member cannot place two or more title reservations for a volume simultaneously (axiom 3).

4.4.15 VOLUME

Class name: *VOLUME*

Class definition:
A title that is published in a non-serial way, as single volumes.

Identity:
• Same as those for *TITLE*.

Attributes:	
authors	Author(s) of the volume.

Chapter 5

Formal specification of the UoD model: classes and life cycles

The specification in this chapter and the following is checked by a parser generated (using LLgen) from the context-free grammar of the LCM 3.0 syntax report [4].

5.1 DOCUMENT

```
begin object class DOCUMENT
  attributes
    available_at: DEPARTMENT;
    barcode: NATURAL                                     fixed;
    location_code: STRING                                fixed;
    owner: DEPARTMENT;
    max_borrowing_period: NATURAL;
    penalty: MONEY;
    price:  MONEY;
    volume: VOLUME;
    technology: STRING;
  keys
    location_code;
  identifiers
    barcode;
  predicates
    Available                                            initially True;
  events
    bind_in(DOCUMENT);
    bind_out(DOCUMENT);
    borrow(DOCUMENT);
    destroy(DOCUMENT)                                    deletion;
    d_res_borrow(DOCUMENT);
    find(DOCUMENT);
    lose(DOCUMENT);
    renew(DOCUMENT);
    return(DOCUMENT);
    t_res_borrow(DOCUMENT, VOLUME);
    write_off(DOCUMENT)                                  deletion;
  life cycle
-- ACQUIRE and CATALOGING are two subprocesses not defined here.
    forall d : DOCUMENT, dep : DEPARTMENT, bar : NATURAL, loc : STRING,
           max : NATURAL, own : DEPARTMENT, pen : MONEY, pri : MONEY,
           tec : STRING, vol : VOLUME ::
     DOCUMENT(d, dep, bar, loc, max, own, pen, pri, tec, vol) =
       create(d;dep,bar,loc,max,own,pen,pri,tec,vol) . ACQUISITION(d) .
       CATALOGING(d) . AVAILABLE(d);
    forall d : DOCUMENT, v : VOLUME ::
     AVAILABLE(d,v) =
       bind_out(d) . bind_in(d) . AVAILABLE(d,v) +
       lose(d) .(find(d). AVAILABLE(d) + write_off(d)) +
       (borrow(d) + t_res_borrow(d,v) + d_res_borrow(d)) . LOANED(d) +
       destroy(d);
    forall d : DOCUMENT ::
     LOANED(d) = return(d) . AVAILABLE(d) +
       renew(d) . LOANED(d) +
       lose(d) .(find(d) . LOANED(d) + write_off(d));

  axioms
-- 1, 2, 3, 4. A document is not available after being borrowed or lost,
-- missed, or after going to the binder.
-- Effect axioms.
   forall d: DOCUMENT:: [borrow(d) ] not Available(d);
   forall d: DOCUMENT, v: VOLUME:: [t_res_borrow(d; v) ] not Available(d);
   forall d: DOCUMENT:: [d_res_borrow(d) ] not Available(d);
   forall d: DOCUMENT:: [lose(d) ] not Available(d);
   forall d: DOCUMENT:: [bind_out(d) ] not Available(d);

-- 5, 6, 7. A document is available after being returned, found, or
-- brought back from the binder.
-- Effect axioms.
   forall d: DOCUMENT:: [return(d)] Available(d);
   forall d: DOCUMENT:: [find(d)] Available(d);
   forall d: DOCUMENT:: [bind_in(d)] Available(d);

-- 8, 9, 10. A document can only be borrowed if it is borrowable.
-- Local precondition necessary for success.
   forall d: DOCUMENT ::
     <borrow(d)> True ->  max_borrowing_period(d) > 0;
   forall d: DOCUMENT, v: VOLUME::
     <t_res_borrow(d, v)> True -> max_borrowing_period(d) > 0;
   forall d: DOCUMENT ::
     <d_res_borrow(d)> True -> max_borrowing_period(d) > 0;

-- 11, 12. A normal borrow event is only possible if there are no
-- title reservations for the volume of which an instance is borrowed and
-- if there are no document reservations for the document.
-- Global precondition necessary for success.
   forall d: DOCUMENT, r: T_RESERVATIONS::
     <borrow(d)> True -> not volume(r) = volume(d);
   forall d: DOCUMENT, r: D_RESERVATIONS::
     <borrow(d)> True -> not document(r) = d;

-- 13. t_res_borrow  only occurs when there is no document reservation
-- for the borrowed document. This gives precedence to document
-- reservations.
   forall d: DOCUMENT, v : VOLUME, r: D_RESERVATIONS::
     <t_res_borrow(d; v)> True -> not document(r) = d;

-- 14. A document is only borrowed by t_res_borrow if it is an instance
-- of the reserved volume. (Ensuring that the borrower has the right
-- identity is taken care of by the transaction.)
-- Local precondition necessary for success.
   forall d: DOCUMENT,  r: T_RESERVATIONS::
     <t_res_borrow(d; v)> True -> volume(d) = v;

-- 15, 16. Renewal is only possible if there are no title reservations for
-- the volume of which a document borrowed and if there are no document
-- reservations for the document.
-- Global precondition necessary for success.
  forall  d: DOCUMENT, t: DATE, r: T_RESERVATIONS::
    <renew(d; t)> True -> not volume(r) = volume(d);
  forall  d: DOCUMENT, t: DATE, r: D_RESERVATIONS::
    <renew(d; t)> True -> not document(r) = d;
end object class DOCUMENT;
```

Remarks:

- `lose` occurs in two transactions, called `miss` and `lose` (see subsection 6.1.4). As part of the *miss* transaction, it has no communication partners. As part of the *lose* transaction, `DOCUMENT.lose` communicates with `FINE.lose`, `LOAN.lose` and `MEMBER.lose_document`. This means that it is synchronized with the creation of a fine for the member who lost the document, that the loan during which this loss occurred is terminated, and tat the number of documents borrowed by the member is decreased by one. Thus, the `lose` transaction is an **asymmetric communication**: `DOCUMENT.lose` can occur on its own, but the other transaction components cannot.
- Looking at the life cycle for `DOCUMENT` instances, we see that the `lose` event starts an **interrupt** that may lead to a termination (via `write_off` or may lead to a return to the state in which the interrupt occurred. `LCM` has currently no facility for defining interrupts.
- `ACQUISITION` and `CATALOGING` are subprocesses of `DOCUMENT` that are not specified here. `ACQUISITION` includes reception of the document, paying for it, sending it back if it is damaged,dealing with copies ordered but received too late, etc. `CATALOGING` deals with classification, updating the user catalog, making the updated catalog available and putting the document in a location in the store room. `ACQUISITION` and `CATALOGING` are the **views** that the finance and acquisition departments have of a `DOCUMENT`. `LCM` does not currently have the ability to specify views. When adding the ability to define views of classes to CMSL, we should also incorporate the ability to define views on processes.

5.2 MEMBER

```
begin object class MEMBER
  attributes
    address: STRING;
    alternative_id: STRING;
    city: STRING;
    class_of_member: MEMBER_CLASS;
    department: DEPARTMENT;
    message: STRING                                      initially "";
    name: STRING;
    nr_documents_borrowed: NATURAL                       initially 0;
    pass: PASS;
    telephone: STRING;
    zip: ZIP;
  predicates
    Excluded                                             initially False;
  events
    borrow(MEMBER);
    d_res_borrow(MEMBER);
    exclude(MEMBER);
    get_pass(MEMBER, PASS);
    include(MEMBER);
    leave(MEMBER);
    lose_document(MEMBER);
    renew(MEMBER);
    return(MEMBER);
    terminate(MEMBER)  deletion;
    t_res_borrow(MEMBER);
  life cycle
    forall m : MEMBER, add : STRING, alt : STRING, cit : STRING,
           cla : MEMBER_CLASS, dep : DEPARTMENT, nam : STRING, pas : PASS,
           tel : STRING, zip : ZIP, p : PASS ::
    MEMBER(m,add, alt, cit, cla, dep, nam, pas, tel, zip, p) =
      create(m;add, alt, cit, cla, dep, nam, pas, tel, zip). LIFE(m,p);
    forall m : MEMBER, p : PASS ::
    LIFE(m,p) = ACTIVE(m,p) . leave(m) . RETURN(m) . terminate(m) +
           exclude(m) . RETURN(m) . include(m) . LIFE(m,p);
    forall m : MEMBER ::
      ACTIVE(m,p) = (borrow(m) + t_res_borrow(m) + d_res_borrow(m) + renew(m) +
                  return(m) + get_pass(m,p)) . ACTIVE(m,p);
    forall m : MEMBER ::
      RETURN(m) = return(m) . RETURN(m);
  axioms
-- 1, 2. Excluded remembers whether a member suffered an exclude without a
-- subsequent include.
-- Effect axioms.
   forall m: MEMBER:: [exclude(m)] Excluded(m);
   forall m: MEMBER:: [include(m)] not Excluded(m);
```

```
-- 3. A member can only be included when all his or her fines are paid and
-- no documents are borrowed by the member.
-- Global precondition necessary for success.
   forall m: MEMBER , f: FINE, l: LOAN::
     <include(m)> True -> member(f) != m and member(l) != m;

-- 4. Members cannot exceed their borrowing allowance.
-- Global static integrity constraint.
   forall m: MEMBER::
     nr_documents_borrowed(m) < borrowing_allowance(class_of_member(m));

-- 5, 6, 7, 8, 9. The number of documents currently borrowed is affected
-- only by borrow, return and lose_document.
-- Effect axioms.
   forall m: MEMBER, n: NATURAL::
     nr_documents_borrowed(m) = n ->
     [borrow(m)] nr_documents_borrowed(m) = n+1;
   forall m: MEMBER, n: NATURAL::
     nr_documents_borrowed(m) = n ->
     [t_res_borrow(m)] nr_documents_borrowed(m) = n+1;
   forall m: MEMBER, n: NATURAL::
     nr_documents_borrowed(m) = n ->
     [d_res_borrow(m)] nr_documents_borrowed(m) = n+1;
   forall m: MEMBER, n: NATURAL::
     nr_documents_borrowed(m) = n ->
     [return(m)] nr_documents_borrowed(m) = n-1;
   forall m: MEMBER, n: NATURAL::
     nr_documents_borrowed(m) = n ->
     [lose_document(m)] nr_documents_borrowed(m) = n-1;

-- 10. Each pass belongs to at most one member.
-- Static global integrity constraint.
   forall m1, m2 : MEMBER:: pass(m1) = pass(m2) -> m1 = m2;

-- 11. A pass knows of which member it is a pass.
-- Static global integrity constraint.
   forall m: MEMBER :: member(pass(m)) = m;

-- 12. Assign a new pass to the member. This is synchronized with
-- PASS.create(p, n, m) so that axiom 13 is respected.
-- Effect axiom.
   forall m: MEMBER, p: PASS:: [get_pass(m; p)] pass(m) = p;
end object class MEMBER;
```

Remarks:

- The `message` attribute may be viewed as a typical database system attribute; members don't walk around in the UoD with a message attribute attached to them. So one may wonder why `message` appears in a model of the UoD and not of the DBS. However, what is here represented is the fact that the library has a commitment to hand a message to the member, such as "you forgot your umbrella" etc. This commitment is a fact that exists in the UoD. To be more precise, it is a relationship between the member and the library (created by the library). Because we don't represent the library, we attach it to MEMBER.

- nr_documents_borrowed is a **derived attribute**. Its derivation is a query on *LOAN*, which may be specified by

  ```
  nr_documents_borrowed(m) = card({ d | exists <document:  d, member:  m> in
  LOAN }).
  ```

  Like relational algebra, we must be manipulate class existence sets as sets. This facility should be added to LCM.

- Axiom 13 says that the PASS.member attribute is the left inverse of the MEMBER.pass attribute. A proof that the DBS transactions preserve this constraint would be one of the tasks of a workbench for LCM.

- The constraint that a member cannot borrow more than indicated by the allowance associated with his or her class is deontic. We regiment it (turn it into a totalitarian constraint) by not representing any exceptions to this. *Regimentation* is a term introduced by Jones and Sergot [8]. We called this the *totalitarian* reading of deontic constraints elsewhere [18].

- Axiom 5 makes crucial use of the fact that nor all variables are bound at the left-hand side of the implication arrow. It is equivalent to

  ```
  forall m : MEMBER::
  <include(m) > True -> forall f : FINE :: (not member(f) = m)
  and
  forall b : LOAN:: (not member(b) = m)
  ```

  This means that include may be viewed as a multicast, for its precondition asks if there is still any FINE or LOAN instance for this member.

5.3 MEMBER_CLASS

```
begin object class MEMBER_CLASS
  attributes
    borrowing_allowance: NATURAL;
    borrowing_periods: SET[NATURAL];
    name: MEMBER_CLASS_NAME                               fixed;
  identifiers
    name;
  axioms
-- 1. In combination with the fact that name is an identifier this axiom
-- says that there are at most four existing MEMBER_CLASS instances.
   forall m: MEMBER_CLASS:: Exists(m) -> name(m) = external
                                      or name(m) = group
                                      or name(m) = staff
                                      or name(m) = student;
```

```
-- 2, 3. Business rules for borrowing allowances.
   forall m: MEMBER_CLASS::
     name(m) = staff -> borrowing_allowance(m) = 20;
   forall m: MEMBER_CLASS::
     not name(m) = staff -> borrowing_allowance(m) = 10;


-- 4, 5. Business rules for borrowing periods.
   forall m: MEMBER_CLASS, v: SET[NATURAL]::
     name(m) = staff and v = insert(insert(empty,15),360) -> borrowing_periods(m) = v;
   forall m: MEMBER_CLASS, v: SET[NATURAL]::
     not name(m) = staff and v = insert(empty,15) -> borrowing_periods(m) = v;
end object class MEMBER_CLASS;
```

Remarks:

- MEMBER is partioned into four subclasses, EXTERNAL, GROUP, STAFF and STUDENT. The information stored about an instance of any of these subclasses is the same as stored about MEMBER, so according to the principle of informativeness of taxonomies [10, 15], these subclasses should not be added to the model. However some information is stored about these subclasses themselves (as opposed to information about instances of these classes), viz. the borrowing period and borrowing allowance granted to instances of these classes. We therefore add the MEMBER_CLASS metaclass. Since we do not represent the classes of which this is a metaclass (we just decided not to represent EXTERNAL, GROUP, STAFF and STUDENT), this metaclass is not related to another class through an instantiation relationship. In general, LCM does not formalize the instantiation relationship. The fact that TITLE is a metaclass with respect to DOCUMENT is not represented in the logic of the language.
- The specification uses the following value type:

  ```
  value type
    MEMBER_CLASS_NAME = {external, group, staff, student}
  ```

  Note that there are models of this specification in which no member class exists.

5.4 T_RESERVATION

```
begin relationship class T_RESERVATION
  components
    member: MEMBER;
    volume: VOLUME;
  attributes
    date_reserved: DATE;
  events
    cancel(T_RESERVATION)                                   deletion;
    res_borrow(T_RESERVATION)                               deletion;
  life cycle
    forall t : T_RESERVATION, dat : DATE ::
```

```
      T_RESERVATION(t) = create(t;dat) . (cancel(t) + res_borrow(t));
 axioms
-- 1. The res_borrow transaction only succeeds if there is no
-- reservation older than the current one. This leaves the choice between
-- reservations of the same age nondeterministic, to be resolved on a
-- first-come first-serve basis.
-- Global precondition necessary for success.
  forall r1, r2: T_RESERVATION::
    <res_borrow(r1)> True ->
         date_reserved(r1) <= date_reserved(r2) or volume(r1) != volume(r2);


-- 2. A volume can only be reserved if none of its copies are available
--(i.e. borrowed.)
-- Global precondition necessary for success.
   forall r: T_RESERVATION,  d: DOCUMENT::
     <create(r)> True -> volume(r) != volume(d) or not Available(d);

-- 3. A volume can only be reserved by a member who did not already reserve
-- that same volume.
-- Global precondition necessary for success.
   forall r1, r2: T_RESERVATION::
     <create(r1)> True ->
     volume(r1) != volume(r2) or  member(r1) != member(r2);
end relationship class T_RESERVATION;
```

Remarks:

- `cancel` and `res_borrow` can be replaced by a single delete event, that would then participate in the `t_res_borrow` and `t_cancel` transactions.
- Axiom 1 is quite complex; a theorem-prover would probably have some difficulties in handling this.

5.5 D_RESERVATION

```
begin relationship class D_RESERVATION
  components
    document: DOCUMENT;
    member : MEMBER;
  attributes
    date_reserved: DATE;
  events
    cancel(D_RESERVATION)                                  deletion;
    d_res_borrow(D_RESERVATION)                            deletion;
  life cycle
    forall d : D_RESERVATION, dat : DATE ::
      D_RESERVATION(d,dat) = create(d;dat) . (cancel(d) + d_res_borrow(d));
 axioms
-- 1. The d_res_borrow transaction can only succeed if there is no document
-- reservation older than the current one. This leaves the choice between
-- document reservations of the same age nondeterministic, to be resolved on a
-- first-come first-serve basis.
-- Global precondition necessary for success.
   forall r1, r2 : D_RESERVATION ::
     <d_res_borrow(r1)> True ->
     document(r1) != document(r2) or date_reserved(r1) <= date_reserved(r2);


-- 2. A document can only be reserved if it is not available
-- Global precondition necessary for success.
   forall r: D_RESERVATION ::
     <create(r)> True ->  not Available(document(r));


-- 3. A document can only be reserved by a member who did not already reserve
-- that same document.
-- Global precondition necessary for success.
   forall r1, r2: D_RESERVATION::
     <create(r1)> True ->
     document(r1) != document(r2) or member(r1) != member(r2);
end relationship class D_RESERVATION;
```

5.6 LOAN

```
begin relationship class LOAN
  components
    document : DOCUMENT;
    member : MEMBER;
  attributes
    date_borrowed : DATE;
    return_before : DATE;
  events
    exclude(LOAN);
    lose(LOAN)                                          deletion ;
    overdue(LOAN);
    remind(LOAN);
    renew(LOAN,DATE,DATE); -- 2nd arg today, 3rd arg return_before
    return(LOAN)                                        deletion;
  life cycle
    forall l : LOAN, d1,d2,d1a,d2a,dat,ret : DATE ::
      LOAN(l,dat,ret,d1,d2,d1a,d2a) = create(l;dat,ret) . RENEW(l,d1,d2,d1a,d2a);
    forall l : LOAN, d1,d2,d1a,d2a : DATE ::
      RENEW(l,d1,d2,d1a,d2a) =
        return(l) + lose(l) + renew(l) . RENEW(l,d1,d2,d1a,d2a) +
        overdue(l) . remind(l) . OVERDUE1(l,d1,d2,d1a,d2a);
    forall l : LOAN, d1, d2, d1a, d2a : DATE ::
      OVERDUE1(l,d1,d2,d1a,d2a) =
        return(l) + lose(l) + renew(l,d1,d2) . RENEW(l,d1,d2,d1a,d2a) +
        overdue(l) . remind(l) . OVERDUE2(l,d1a,d2a);
    forall l : LOAN, d1, d2, d1a, d2a : DATE ::
      OVERDUE2 = return(l) + lose(l) + renew(l,d1,d2) . RENEW(l,d1,d2,d1a,d2a) +
      exclude(l) . EXCLUDE(l);
    forall l : LOAN ::
      EXCLUDE(l) = return(l) + lose(l);
  axioms
-- 1. renew(l; d) changes the return date of b into d.
-- Effect axiom.
   forall l : LOAN, d1, d2 : DATE:: -- d1 : due, d2 : return_before
     [renew(l;d1, d2)] return_before(l) = d2;

-- 2. A member can only borrow a document for one of the allowed borrowing
-- periods for his or her member class.
-- Global precondition necessary for success.
   forall l: LOAN, d : DOCUMENT, m: MEMBER,
          date_borrowed, return_before : DATE::
     <create(l; d, m, date_borrowed, return_before)> True ->
     date_borrowed <= return_before and
     difference(date_borrowed, return_before)
         In borrowing_periods(class_of_member(m));

-- 3. A member can only extend a borrowing period into one of the allowed
-- borrowing periods for his or her member class.
-- Global precondition necessary for success.
   forall l: LOAN, today, return_before : DATE::
     <renew(l; today , return_before) > True ->
     today <= return_before and
     difference(today, return_before)
     In borrowing_periods(class_of_member(m));
end relationship class LOAN;
```

Remarks:

- This specification includes two **temporal events**, `overdue` and `exclude`; they occur when a certain period of time has elapsed since a reminder. Nothing in the specification indicates that these are temporal events. As it is, what happens is that the DBS user types in the `overdue` transaction and then is in the position to type in the `remind` transaction. This use of the `overdue` transaction is useless; its proper use is to signal the fact that a significant moment in time has occurred. It is useful to the user if the DBS does the signaling, it is not useful to the DBS nor to the user if the user does the signaling.

- The `exclude` event is typed in the DBS to store in the DBS that a user is excluded; this is a useful transaction.

  The ability to specify temporal events should be added to the DBS.

5.7 PASS

```
begin object class PASS
  attributes
```

```
    barcode: NATURAL                                        fixed;
    member: MEMBER;
  identifiers
    barcode;
  predicates
    Lost                                                    initially False;
  events
    create(PASS;NATURAL,MEMBER)                             creation;
    destroy(PASS)                                           deletion;
    find(PASS;DATE);
    lose(PASS;DATE);
  life cycle
    forall p : PASS, n : NATURAL, m : MEMBER, dl, df : DATE ::
      PASS(p,nm,dl,df) = create(p;n,m) .
        (destroy(p) + lose(p;dl) . find(p;df) . destroy(p));
  axioms
-- 1. A pass can only be created when the member for which it is created
-- exists and when the pass does not exist.
-- Global precondition sufficient for failure.
   forall p :PASS, n: NUMBER, m: MEMBER, d: DATE::
     <create(p; n, m, d)> True -> Exists(p) and not Exists(m);

-- 2. Effect axiom.
   forall p :PASS, n: NATURAL, m: MEMBER::
   [create(p; n, m)] Exists(p)
                     and barcode(p) = n
                     and member(p) = m
                     and not Lost(p);

-- 3. The Lost predicate remembers if a pass was ever lost. It remains
-- True even when the pass is found.
-- Effect axiom.
   forall p: PASS, d: DATE:: [lose(p, d)] Lost(p);

-- 4. It is possible that there are two passes for the same member.
-- However, in that case all passes that are not pointed at by the
-- pass attribute of the member are Lost.
-- Global static constraint. Generates global preconditions.
   forall p: PASS :: not pass(member(p)) = p -> Lost(p);

-- 5. As long as it is assigned to a member, a pass cannot be destroyed.
-- Global precondition sufficient for failure
   forall p: PASS:: pass(member(p)) = p -> [destroy(p)] False;
end object class PASS;
```

Remarks:

- Note that if a member is excluded, he or she can lose a pass (pass loss is not part of the member life cycle, so as far as the member life cycle is concerned, there are no restrictions on the moment that this event happens). The excluded member must however wait till he or she is included before he or she can get a new pass. Thus, in this model, exclusion blocks the member, not the pass.

5.8 DEPARTMENT

```
begin object class DEPARTMENT
  attributes
    name : STRING;
end object class DEPARTMENT;
```

5.9 FINE

```
begin object class FINE
  attributes
    amount: MONEY;
    loan: LOAN;
    outstanding: MONEY;
    paid: MONEY                                        initially 0;
    pay_before: DATE                                   initially null_DATE;
    waived: MONEY                                      initially 0;
  events
    clear(FINE)                                        deletion;
    exclude(FINE,LOAN,DATE)                            creation;
    lose(FINE, LOAN, DATE)                             creation;
    pay(FINE, MONEY);
    waive(FINE, MONEY);
  life cycle
    forall f : FINE, ll, le : LOAN, dl, de : DATE, mp, mw : MONEY ::
      FINE(f,ll,le) =(lose(f;ll,dl) + exclude(f;le,de)) . BODY(f,mp,mw);
    forall f : FINE, mp, mw : MONEY ::
      BODY(f,mp,mw) =(pay(f,mp) + waive(f,mw)) . BODY(f,mp,mw) + clear(f);
  axioms
-- 1. A FINE instance may be created in an exclude transaction. The
-- member is charged 70 (in whatever currency unit this is agreed to
-- represent).
-- Effect axiom.
   forall f : FINE, l : LOAN, d : DATE::
     [exclude(f;l,d)] Exists(f) and loan(f) = l
                                and amount(f) = 70
                                and pay_before(d) = d
                                and outstanding(f) = 70
                                and paid(f) = 0
                                and waived(f) = 0;

-- 2. Exclude is a creation event of FINE.
-- Global precondition sufficient for success.
   forall f : FINE, loan: LOAN,  pay_before : DATE::
     not Exists(f) and Exists(b) ->
     <exclude(f;loan,pay_before)> True;

-- 3. A FINE instance may be created in a lose transaction. In this case, the
-- member who loses the document is charged with the penalty of the document.
   forall  f : FINE , l: LOAN::
    [lose(f; l, d)] Exists(f)
                    and loan(f) = l
                    and pay_before(f) = d
                    and amount(f) = penalty(document(b))
                    and outstanding(f) = amount(f)
                    and paid(f) = 0
                    and waived(f) = 0;

-- 4. Global precondition sufficient for success.
   forall  f : FINE , l: LOAN::
    not Exists(f) and Exists(l) -> <lose(f; l)> True;

-- 5. The total amount equals the amount outstanding plus the amount
-- paid plus the amount waived.
-- Static integrity constraint.
   forall f: FINE:: amount(f) = outstanding(f) + paid(f) + waived(f);

-- 6. A fine cannot be deleted when there is still an amount outstanding.
-- Local precondition necessary and sufficient for success.
  forall f : FINE:: outstanding(f) = 0 -> < clear(f) > True;
  forall f : FINE:: < clear(f) > True -> outstanding(f) = 0;
end object class FINE;
```

Remarks:

- A fine can be created in two ways, by losing a document or by returning a document too late. If a document is lost after being overdue, then a fine is paid for both overdue and lost.
- `outstanding` is another example of a derived attribute. This time, derivation can be done by a simple computation rule like

  `outstanding(f) = amount(f) - paid(f) - waived(f)`.

  Axiom 5 is thus really a derivation rule.
- `paid` and `waived` are derivable attributes, because they just summarize the past `pay` and `waive` events in a `FINE`.

5.10 PART_OF

```
begin relationship class PART_OF
  components
    periodical: PERIODICAL                         surjection;
    volume: VOLUME;
  axioms
-- A volume cannot be part of more than one periodical.
-- Global static constraint(cardinality axiom).
   forall p1, p2: PART_OF::volume(p1) = volume(p2) -> p1 = p2;
end relationship class PART_OF;
```

5.11 PERIODICAL

```
begin object class PERIODICAL
  specialization of TITLE;
  partitioned by
    JOURNAL, SERIAL_WORK;
  attributes
    editors: LIST[STRING] ;
end object class PERIODICAL;
```

5.12 JOURNAL

```
begin object class JOURNAL
  specialization of PERIODICAL;
  attributes
   nr_of_issues: NATURAL;
end object class JOURNAL;
```

5.13 SERIAL_WORK

```
begin object class SERIAL_WORK
  specialization of PERIODICAL;
end object class SERIAL_WORK;
```

5.14 TITLE

```
begin object class TITLE
  partitioned by
    VOLUME, PERIODICAL;
  attributes
    annotation: STRING                                   initially "";
    code: STRING;
    title: STRING;
    title_key:  STRING;
  keys
    title_key;
end object class TITLE;
```

5.15 VOLUME

```
begin object class VOLUME
  specialization of TITLE;
  attributes
```

```
    authors: LIST[STRING];
end object class VOLUME;
```

Chapter 6

Formal specification of the DBS boundary and communication model

6.1 Circulation activities

6.1.1 Borrowing

```
begin service Borrowing
 transactions
   borrow(LOAN, -- loan
          MEMBER, -- borrower
          DOCUMENT, -- borrowed_document
          DATE, -- today
          DATE); -- return_before
   overdue(LOAN);
   return(LOAN);
   remind(LOAN);
   renew(LOAN,
         DATE, -- today
         DATE); -- return_before
decompositions
-- LOAN.create creates a LOAN instance b and initializes it so
-- that its components are c and m, and the return date is set to
-- d2. The DOCUMENT.borrow(d, d1) event sets the Available flag
-- of c to False. MEMBER.borrow increases the number of documents
-- currently borrowed by the member by one and also checks whether the
-- member is not excluded.
   forall  loan: LOAN, borrowed_document: DOCUMENT, borrower: MEMBER,  today,
           return_before : DATE::
     borrow(loan, borrower, borrowed_document,
            today, return_before) =
       LOAN.create(l; borrowed_document, borrower,
                   today, return_before) &
       DOCUMENT.borrow(borrowed_document) &
       MEMBER.borrow(borrower);

-- LOAN.return kills the LOAN instance, DOCUMENT.return sets
-- the Available flag of the returned document to True and
-- MEMBER.return decreases the number of documents currently borrowed
-- by m by one.
   forall d: DOCUMENT, m: MEMBER::
     return(LOAN_id(d, m))=
     LOAN.return(LOAN_id(d, m)) &
     DOCUMENT.return(d) &
     MEMBER.return(m);

-- LOAN.renew sets the return date for l = <m, d> to d2 and if the
-- difference between today and d2 is in the allowed borrowing
-- periods for the member.
-- DOCUMENT.renew merely adds the renewal event to the history of the
-- borrowed document, where it is then available for future queries.
-- MEMBER.renew checks whether the member is not excluded.
   forall d: DOCUMENT, m: MEMBER, d1, d2:DATE::
     renew(d, m, d1, d2)=
     LOAN.renew(LOAN_id(d, m),
                d1, d2) &
     DOCUMENT.renew(d) &
     MEMBER.renew(m);

   forall m : MEMBER, d : DOCUMENT::
     overdue(LOAN_id(d, m)) = MEMBER.overdue(m);

   forall m : MEMBER, d : DOCUMENT::
     remind(LOAN_id(d, m)) = MEMBER.remind(m);
end service Borrowing;
```

6.1.2 Title Reservations

```
begin service TitleReservations
 transactions
   t_reserve(VOLUME,
             T_RESERVATION,
     MEMBER,
             DATE); -- today
   t_cancel(T_RESERVATION);
   t_res_borrow(VOLUME,
                LOAN,
                T_RESERVATION,
                MEMBER,
                DOCUMENT,
                DATE, -- today
             DATE); -- return_before
 decompositions
```

```
-- Create reservation relation r between volume v and member m.
-- After creation, volume(r) = v and member(r) = m.
   forall v: VOLUME, r: T_RESERVATION, m: MEMBER, d: DATE::
     t_reserve(v, r, m, d) =
     T_RESERVATION.create(r; v, m, d);

-- Destroy a reservation.
   forall r: T_RESERVATION::
     t_cancel(r) = T_RESERVATION.cancel(r);

-- Create a LOAN relationship l between document c and member m,
-- destroy reservation r between m and the volume of c and add the
-- t_res_borrow event to the history of c. This event also flags c
-- as being not available. The LOAN.create event ensures that
-- l = <document: d, member: m >
   forall v: VOLUME, l: LOAN,  m: MEMBER, d: DOCUMENT, d1, d2: DATE::
     t_res_borrow(v, l, m, c, d1, d2) =
     LOAN.create(l; c, m, d1, d2) &
     T_RESERVATION.res_borrow(T_RESERVATION_id(v, m)) &
     DOCUMENT.t_res_borrow(d; v) &
     MEMBER.t_res_borrow(m);
end service TitleReservations;
```

6.1.3 Document reservations

```
begin service DocumentReservations
 transactions
   d_reserve(VOLUME,
             D_RESERVATION,
             MEMBER,
             DATE);  -- today
   d_cancel(D_RESERVATION);
   d_res_borrow(VOLUME,
             LOAN,
                D_RESERVATION,
                MEMBER,
          DOCUMENT,
                DATE, -- today
          DATE); -- return before
 decompositions
-- Create reservation relation r between document c and member m.
-- After creation, document(r) = c and member(r) = m.
   forall d: DOCUMENT, r: D_RESERVATION, m: MEMBER, t: DATE::
     d_reserve(d, r, m, t) =
     D_RESERVATION.create(r; d, m, t);

-- Destroy a reservation.
   forall r: D_RESERVATION::
     d_cancel(r) = D_RESERVATION.cancel(r);
```

```
-- Create a LOAN relationship l between document c and member m,
-- destroy reservation r between m and  c and add the
-- d_res_borrow event to the history of c. This event also flags c
-- as being not available. The LOAN.create event ensures that
-- l = < document: c, member: m >.  d1 is today, d2 is the return before date.
   forall  l: LOAN,  m: MEMBER, d: DOCUMENT, d1, d2: DATE::
     d_res_borrow(l, m, d, d1, d2) =
     LOAN.create(l; c, m, d1, d2) &
     D_RESERVATION.res_borrow(D_RESERVATION_id(d, m)) &
     DOCUMENT.d_res_borrow(d, v) &
     MEMBER.d_res_borrow(m);
end service DocumentReservations;
```

6.1.4 Lost document handling

```
begin service LostDocumentHandling
 transactions
   miss(DOCUMENT);
   lose(LOAN, FINE, DATE);
   find(DOCUMENT);
   write_off(DOCUMENT);
 decompositions
-- DOCUMENT.miss suspends the normal life process of a DOCUMENT instance.
-- After miss, the only two possible events in the life of a DOCUMENT
-- instance are find and write_off.
   forall d: DOCUMENT :: miss(d)= DOCUMENT.lose(d);

-- DOCUMENT.lose suspends the normal life cycle of a DOCUMENT instance. The
-- only two next possible events in the life of a DOCUMENT are
-- DOCUMENT.find and DOCUMENT.write_off.
-- FINE.lose creates a FINE instance and initializes it with the
-- penalty for the lost document. LOAN.lose destroys the LOAN
-- instance during the life of which the document is lost. Thus, the document is
-- not borrowed anymore(it does not occur in any existing LOAN
-- instance) but the Available flag of the document is still set to
-- False. When it is found, then this flag is set to True(see the
-- DOCUMENT specification). MEMBER.lose decreases the number of
-- documents borrowed by the member with 1.
   forall l: LOAN, f: FINE, t : DATE::
     lose(l, f, t) =
       DOCUMENT.lose(document(l)) &
       FINE.lose(f, l, t) &
       LOAN.lose(l) &
       MEMBER.lose_document(member(l));

   forall d: DOCUMENT:: find(c) = DOCUMENT.find(d) ;

   forall d: DOCUMENT :: write_off(d) = DOCUMENT.write_off(d) ;
end service LostDocumentHandling;
```

6.2 Member services

6.2.1 Membership

```
begin service Exclusions
 transactions
   become_member(MEMBER,
           STRING, -- address
                 STRING, -- alternative_id
                 NATURAL, -- barcode
                 STRING, -- city
                 MEMBER_CLASS,
                 DEPARTMENT,
                 STRING, -- name
                 PASS,
                 STRING, -- telephone
                 ZIP);
   exclude(LOAN, FINE, DATE); -- DATE argument is pay_before date.
   include(MEMBER);
   leave(MEMBER);
   terminate(MEMBER);
 decompositions
   forall m : MEMBER, a: STRING, b : NATURAL, id: STRING, c: STRING, mc: MEMBER_CLASS, d:
          DEPARTMENT, ms: MEMBER, n: STRING, p: PASS, t: STRING, z: ZIP::
     become_member(m, a, id, b, c, mc, d, ms, n, p, t, z) =
     MEMBER.create(m; a, id, c, mc, d, ms, n, p, t, z) &
     PASS.create(p; b, m);

-- MEMBER.exclude sets the Excluded flag of m to True so that
-- m cannot borrow or renew any documents. LOAN.exclude puts the
-- LOAN process into a state where it can only suffer a lose or
-- return event, both of which terminate the LOAN process.
-- FINE.exclude creates a fine of 70(in whichever currency is
-- agreed), together with a deadline for payment.
   forall m: MEMBER, c: DOCUMENT, f: FINE, d: DATE::
     exclude(LOAN_id(c, m), f, d)=
     MEMBER.exclude(m) &
     LOAN.exclude(LOAN_id(c, m)) &
     FINE.exclude(f, d, LOAN_id(c,m));

-- MEMBER.include can only occur if the member paid all fines and is
-- not involved in any borrowings.
   forall m: MEMBER:: include(m) = MEMBER.include(m);

   forall m: MEMBER:: leave(m) = MEMBER.leave(m);

   forall m: MEMBER:: terminate(m) = MEMBER.terminate(m);
end service Exclusions;
```

6.2.2 FineHandling

```
begin service FineHandling
 transactions
   pay(FINE, MONEY);
   waive(FINE, MONEY);
   clear(FINE);
 decompositions
-- Decrease the amount to be paid with a.
   forall f: FINE, a: MONEY:: pay(f, a) = FINE.pay(f, a);

-- Decrease the amount to be paid with a.
   forall f: FINE, a: MONEY:: waive(f, a) = FINE.waive(f, a);

-- Destroy the FINE instance. This event can only happen when the
-- amount to be paid is 0.
   forall f: FINE:: clear(f) = FINE.clear(f);
end service FineHandling;
```

6.2.3 PassHandling

```
begin service PassHandling
 transactions
   issue_pass(PASS, NATURAL, MEMBER, DATE);
   lose_pass(PASS, DATE);
   find_pass(PASS, DATE);
   destroy_pass(PASS);
 decompositions
-- PASS.create creates a pass and initializes it so that member(p)
-- points to m.  MEMBER.get_pass sets pass(m) to p.
   forall p: PASS, n: NATURAL, m: MEMBER::
     issue_pass(p, n, m) = PASS.create(p, n, m) &
                           MEMBER.get_pass(m, p);

   forall p: PASS, d: DATE::
     lose_pass(p, d) = PASS.lose(p, d);

   forall p: PASS, d: DATE::  find_pass(p, d) = PASS.find(p, d);

   forall p: PASS:: destroy_pass(p, d) = PASS.destroy(p);
end service PassHandling;
```

Chapter 7

Discussion and conclusions

7.1 Discussion

7.1.1 Event specification

Many events have no axioms specified for them. For example, `leave` and `terminate` etc. in the `MEMBER` specification have no preconditions nor effect axioms. However, each action occurrence has at least two effects which are not specified in axioms:

1. The event occurrence may bring the life cycle in a new state. This state change is not specified in dynamic logic axioms.
2. The event occurrence occurs. This may seem trivial, but much of the data in a DBS consists just of the memory that certain events occurred. This points at the need to define a model for `LCM` specifications in which for each objects, we remember the trace of events that occurred in its past. We return to this when discussing possible temporal logic extensions of `LCM` below.

7.1.2 Precondition specification

In `LCM`, each precondition must be attached to a local event by means of a precondition axiom; transactions have no explicit preconditions. Note that there is a precondition hidden in the decomposition of the transactions in the service specifications: These decompositions may be used to enforce equality of the communicating event parameters. For example, in the decomposition of the `t_res_borrow` transaction, the variable m appears in the `LOAN.create` event as the member who does the borrowing and in the `T_RESERVATION.res_borrow` event as the member who placed the reservation. This is part of the formalization of one of the business rules, as indicated in the dictionary entry for `DOCUMENT`.

Another precondition hidden in transaction specifications is that all subjects of different events in the transactions must be different. This does not exclude local communication (within one object). An event in a transaction may be a local communication.

The precondition of an event itself may be local or global. It seems natural to attach global preconditions to transactions, which are global events, in the same way that local preconditions are attached to local events. There are two reasons for not putting global precondition axioms in the service specifications:

1. First, we have not been able to find examples of such intrinsically global preconditions.

2. Second, in the service specifications, we can only define communications between partners that are known at specification time. A precondition that depends upon an arbitrary number other objects cannot be specified this way.

If the author of a specification wants to avoid global preconditions, then in cases where the objects whose state must be tested are known at specification time, the global precondition can be replaced by a number of local preconditions. To see this, suppose an event e with subject o has a precondition ϕ that involves the state of an object o' different from o. Then ϕ is a global precondition because it reads the state of an object other than o. This can be made into a local precondition by the following trick: Define an event e' for o' that has the relevant part of the state of o' as parameters and make e part of a transaction that consists of e and e'. Then e' shows the relevant part of the state of o' to e through a communication. The global precondition can be dropped and can be replaced by at most two local preconditions, one for e (if necessary) and one for e'.

However, this way of communicating the state of an object to another object increases complexity of the specification. In addition, global preconditions that depend upon the state of a set of objects that cannot be enumerated at specification time cannot be specified this way; as pointed out above, we need a multicast to evaluate these preconditions. Thus, we prefer to allow global preconditions of local events.

7.1.3 The interpretation of transaction equations

The transaction equations in the service specifications are interpreted in the process algebra, which contains the abstract data type as reduct. Because of this semantic structure, it is permitted to use function symbols in the transaction decompositions. This is because function values are defined in the underlying abstract data type and are therefore accessible from the process algebra.

The attributes and predicates of objects, on the other hand, are interpreted in possible worlds. All possible worlds have the abstract data type as domain, but they may differ in the interpretation of attribute names and predicate names. Attribute values can therefore not be used in transaction decompositions. Note that component functions of relationships are *functions*, and can therefore be used in transaction equations. This permissible because the relationships are passed as tuples with labeled components to the transactions.

7.1.4 The form of the axioms

All axioms are closed formulas in prenex normal form and only use universal quantifiers. Their meaning sometimes crucially depends upon the fact that a quantifier can moved inwards into the formula, because it does not bind any variables in the condition; axiom 5 in `MEMBER` is an illustration of this:

```
forall m: MEMBER , f: FINE, l: LOAN::
  <include(m)> true -> (not member(f) = m) and (not member(l) = m);
```

is equivalent to

```
forall m : MEMBER::
<include(m) > true -> forall f : FINE :: (not member(f) = m)
and
forall b : LOAN:: (not member(b) = m)
```

The following forms of axioms have been encountered. (All axiom forms below are implicitly universally quantified.)

Static integrity constraints:

- $a(x) = a(y) \to x = y$ for attribute a and object variables x and y.
- $P(x) \to a(x) = t_1 \vee \cdots \vee a(x) = t_n$ for object variable x.
- $a(x) = t \to b(x) = t'$ for object variable x.
- $a(b(x)) = x$ for object variable x.

The conditions in these axioms can occur in negated form.

Effect axioms:

- $\phi \to [e(x, \ldots)]\psi$ where x is an object variable, all variables at the right-hand side of the implication sign occur at the left-hand side, and ϕ and ψ are conjunctions of the following kinds of atomic formulas:
  - $a(x) = t$ for a an attribute name, x the same object variable as in e, and t an ADT term;
  - $a(x) = b(x)$ for a and b attributes, and x the same object variable as in e;
  - $P(x)$ for P a predicate and x the same object variable as in e;
  - $\neg P(x)$ for P a predicate and x the same object variable as in e.

Preconditions:

- $\langle e(x, y_1, \ldots, y_n)\rangle true \to t_1 = t_2$ where at least one of t_1 and t_2 is an attribute term. These terms may contain variables not occurring among $x, y_1, \ldots, y_n$. The equation may be negated.
- $\langle e(x, y_1, \ldots, y_n)\rangle true \to L(t_1, \ldots, t_m)$ where at least one of t_i for $i = 1, \ldots, m$ is an attribute term. These terms may contain variables not occurring among $x, y_1, \ldots, y_n$. L is a literal, i.e. a predicate or the negation of a predicate.
- $\langle e(x)\rangle true \to (a(x) = a(y) \to L(b(x), b(y)))$ for object variables x and y. L is a literal whole predicate may be the equality sign.

The condition in a conditional effect axiom is not a precondition for success or failure. It merely serves to bind the variables and cannot evaluate to false. We could avoid the conditions in the effect axioms by allowing direct reference of the value of a variable before an event occurs. For example, using the VDM-style *prev_* variables, we could replace

```
forall m: MEMBER, n: NAT::
  nr_documents_borrowed(m) = n ->
  [borrow(m)] nr_documents_borrowed(m) = n+1
```

by

```
forall m: MEMBER, n: NAT::
[borrow(m)] nr_documents_borrowed(m) = prev_nr_documents_borrowed(m)+1.
```

7.1.5 Modularity

Modularity occurs in several ways in `LCM` specifications. First each *class* specification is modular. This modularity is violated in two ways:

- Global preconditions depend for their evaluation on the state of other objects.
- Each event can only be performed in the context of a transaction; i.e. it must usually communicate with other local events.

It turns out that each class specification is easy to understand in isolation but that especially the communication structure is hard to understand; this is where the intelligence of the model resides. This has been called the **Ravioli problem**[1]: the communication graph of the full model usually looks like a bowl of ravioli.

A second important modularity principle in `LCM` is that each transaction is modular. Each transaction involves a limited number of local events and nothing else. Note that this modularity is orthogonal to that of classes. (This orthogonality is visually represented by the transaction decomposition tables.) An attempt has been made to add some modularity at a higher level than that of transactions, by grouping them into services. It is possible to add a higher-level grouping concept like subject areas or domains; it is not yet clear whether these higher-level grouping concepts can be clearly defined and whether this would enhance understanding of the model.

The ravioli problem can be reduced by not specifying derived attributes, predicates or updates. Specifying these explicitly creates interdependencies between different parts of the model and hence adds complexity and decreases modularity. For example, addition of the derived attribute `nr_documents_borrowed` considerable complicates the `MEMBER` specification. Maintaining this attribute requires axioms 7–11 in `MEMBER` as well as participation of `MEMBER` in the `borrow`, `t_res_borrow`, `d_res_borrow`, `return`, and `lose` transactions. The specification would be simpler and easier to understand without this attribute.

Note that we do not have a module concept with an import relation. The current version of `LCM` is an experiment to see if we can do without this; or if we can reduce the need for a module mechanism by making intelligent use of a workbench with a dynamic viewing mechanism of the specification.

7.2 Possible extensions to `LCM`

7.2.1 Derived attributes

The need for derivation mechanisms is apparent from this case study. A simple example of a derived attribute is `outstanding` in `FINE`. This can be derived with a simple computation rule that can be specified in the abstract data type specification of `MONEY`.

The derivation of the attribute `nr_documents_borrowed` in `MEMBER` requires a more complex derivation mechanism, viz. a query facility in which sets of instances in the database can be manipulated. We return to the need for manipulating sets below. Another way to derive the `nr_documents_borrowed` of a `MEMBER` is to add past time temporal logic with the ability to count the number of past occurrences of creation events of `LOAN` in which the member participated minus the number of destruction events of `LOAN` in which that member participated.

The `paid` and `waived` are derivable attributes in `FINE` that can be dropped if we would have a past temporal logic in which we could look at the occurrences of `pay` and `waive` in the past of a `FINE`. We return to this below.

[1]We have not been able to trace the originater of this term.

7.2.2 Null values

There is no theory of null values embedded in LCM. There are at least three different null values, with the meaning

- unknown,
- does not exist,
- not applicable.

The third one can be eliminated by using taxonomic structures, but this sometimes leads to complicated diagrams and we may have to allow this null value explicitly. The first two kinds cannot be eliminated, but extension of LCM with these kinds of null values is not foreseen in the near future.

For the time being, we use the simple kludge of assuming for every abstract datatype a supertype that contains a *null* value for that type. We require all attributes to be strict, i.e. if $a : s_1 \rightarrow s_2$ then $a(null_{s_1}) = null_{s_2}$. These null values are reminiscent of the semantic element used, in first-order logic with descriptions, to interpret descriptions that refer to nothing.

7.2.3 Manipulating sets

There is a need to be able to manipulate sets in specifications as well as in query-answering. For example, the constraint on the possible borrowing allowances for different classes of library members (axiom 6 in MEMBER) currently has the following form:

```
forall m: MEMBER::
  nr_documents_borrowed(m) < borrowing_allowance(class(m)).
```

This should really be expressed by means of a language that allows us to ask what the cardinality of a solution set to a query is. This could look as follows:

```
forall m: MEMBER, l: LOAN::
card{l: LOAN | document(b) = m} <= borrowing_allowance(m)
```

Because we cannot specify sets in LCM predicatively, we circumvented we introduced the attribute nr_documents_borrowed to count the number of borrowed documents. We already saw that this leads to extra complexity in the specification.

7.2.4 Real time

We encountered the need for real time in such temporal events as overdue and exclude, which trigger certain events in the life of some objects. In the specification, we assume that there is an abstract data type DATE with functions like before and difference as used in the axioms. These can easily be defined in equational logic but this is not sufficient to define real time. We need for example also a facility to specify the condition on time in which a temporal event occurs, to evaluate formulas at a point in time, etc. Real-time temporal logic is planned as an extension to LCM.

It would have been possible to refine the creation axiom of a LOAN instance by defining $d_2 = add\_days(d_1, 21)$ instead of explicitly declaring d_2 to be a parameter. However, this is a simplistic specification, for there is in practice no algorithm to compute d_2 from d_1. This is so because it may depend upon things like public holidays and local agreements between the university and labor representatives about vacations what the value of d_2 must be. These agreements may not even be global for the university but apply to some library departments, and they affect the possible return days and therefore the deadline for returning documents. Currently, at the beginning of each day, each library department computes the return day for documents, stamps a set of paper slips with the return date and hands out this paper slip together with the borrowed document. The entry of the return date is thus connected to business procedures.

7.2.5 Temporal logic

Past temporal logic would allow us to specify the derivation of attributes like `waived` and `paid` in `FINE`. In combination with real time, we could also add an attribute like `date_last_accessed` of `DOCUMENT`. The current specification does not contain this attribute because it would have a null value when a document is created. In addition, it is questionable whether this is an attribute of the UoD entity; the access intended by this attribute is an access to the record in the DBS representing the real-world document. In general, though, past temporal logic can serve as a useful language to specify derived attributes and as a way to avoid some kinds of null values of attributes. Needless to say, it would also be useful as a query language for a historical database system.

7.2.6 Deontic logic

There is a rule that a borrowing period should not exceed the maximum borrowing period for the borrowed document. This is a constraint on the UoD, not on the DBS and it should therefore be stated in deontic logic. If we assume an attribute `borrowing_period` of `LOAN` instances, then a possible specification of this constraint is

```
forall b: LOAN::
OughtToBe(borrowing_period(b) <= max_borrowing_period(document(b)))
```

This is one more reason to add a real-time facility in which we can specify that `borrowing_period` is increased by 1 every day. In addition we need a deontic logic that allows us to specify ought-to-be as well as ought-to-do constraints [1].

7.2.7 Interrupts and premature termination

We also encountered the need for **interrupt handling** in the case study. We saw that `lose` event in the `DOCUMENT` specification is really an interrupt of a `DOCUMENT` process, which may lead to a termination of the `DOCUMENT` process or may cause a return to the state where the interrupt occurred. Interrupts have been dealt with in process algebras as well as temporal logic and the capability to specify them should probably be added when we move to real time applications.

7.2.8 Views

We encountered two kinds of views in the example specification. First, each service specification provides a kind of view on the DBS, consisting of a limited number of transactions. Second, we saw that the `ACQUISITION` and `CATALOGING` subprocesses of `DOCUMENT` are the views that the acquisition and finance departments of the library have of the document life cycle. We can generalize this second kind of view by incorporating the traditional view concept of databases: A view is a window on a DBS that shows some, but not all objects and processes in the DBS. This can be worked out by defining a view as a derived class, which is related by a theory morphism to underlying classes. It is clear that such a mechanism must be added to `LCM`, but it is far from clear what the precise details of such a mechanism must be. The addition of a view specification mechanism to `LCM` is not planned for the near future.

7.3 Functions for a `LCM` workbench

One of the functions of a `LCM` workbench which we sorely missed is a bookkeeping facility that maintained consistency between transaction decomposition tables, the function decomposition tree, service specifications, process graphs, class diagrams, class specifications, and transaction specifications.

A second important function of a workbench, related to the first, is to maintain traceability of the requirements. This means that each informal requirement must be traceable to one or more parts of the specification which formalizes this DBS requirement. The business rules in the dictionary entry for a class attempt to establish this tracability. Ideally, each business rule is formalized at one place in the specification. We have not been able to do this but the example specification comes close to it.

A third major function is of course the ability to prove properties of the specification. We should like to know, for example, where the transactions preserve the static integity constraints. For example, axiom 13 in `MEMBER` (which says that the `PASS.member` attribute is the left inverse of the `MEMBER.pass` attribute) must be respected by the `issue_pass` transaction.

Bibliography

[1] P.d' Altan, J.-J.Ch. Meyer, and R.J. Wieringa. An integrated framework for ought-to-be and ought-to-do constraints. Technical Report IR-342, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[2] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[3] J.F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 User's Manual, version 2.3.* Instituto Superior Técnico, Lisbon, May 1989.

[4] R.B. Feenstra and R.J. Wieringa. LCM 3.0: a language for describing conceptual models. Technical Report IR-344, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[5] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[6] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. `TROLL`–2 report. Informatik-bericht, TU Braunschweig, 1993. *In preparation.*

[7] E.G. IJff. Conceptueel model van de uitleenfuncties van een bibliotheek. Master's thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1991. In Dutch.

[8] A.J.I. Jones and M. Sergot. On the role of deontic logic in the characterization of normative systems. In J.-J.Ch. Meyer and R.J. Wieringa, editors, *Deontic Logic in Computer Science: Normative System Specification*, pages 275–307. Wiley, 1993.

[9] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The `TROLL` Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[10] N. Rescher. *Introduction to Logic.* St. Martin's Press, 1964.

[11] P.A. Spruit. Function symbols in dynamic database logic. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1993.

[12] P.A. Spruit, R.J. Wieringa, and J.-J.Ch. Meyer. Dynamic database logic: The first-order case. In U.W. Lipeck and B. Thalheim, editors, *Modelling Database Dynamics*, pages 103–120. Springer, 1993.

[13] UBVU. Gids voor bezoekers van de bibliotheek vrije universiteit. 9e herziene druk, 1990.

[14] R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566.

[15] R.J. Wieringa. *Information System Development and Conceptual Modeling: A Comparative Study*. Department of Mathematics and Computer Science, Vrije Universiteit, 1993.

[16] R.J. Wieringa. A method for building and evaluating formal specifications of object-oriented conceptual models of database systems (MCM). Technical Report IR-340, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[17] R.J. Wieringa and P.A. Spruit Jonge, W. de. Roles and dynamic subclasses: a modal logic approach. Technical Report IR-341, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[18] R.J. Wieringa, J.-J. Ch. Meyer, and H. Weigand. Specifying dynamic and deontic integrity constraints. *Data and Knowledge Engineering*, 4:157–189, 1989.

[19] R.J. Wieringa and J.-J.Ch. Meyer. Actors, actions, and initiative in normative system specification. *Annals of Mathematics and Artificial Intelligence*, 7:289–346, 1993.