UML-based Fusion Analysis[1]

Shane Sendall and Alfred Strohmeier

Swiss Federal Institute of Technology
Department of Computer Science
Software Engineering Laboratory
1015 Lausanne-EPFL
Switzerland

email: {Shane.Sendall, Alfred.Strohmeier}@epfl.ch

ABSTRACT In recent times, there has been an increased requirement for software to be distributed. The well-known Fusion development method, however, can only be used to develop sequential reactive systems, and certain restricted kinds of concurrent systems. In contrast, the Unified Modeling Language (UML) provides a rich set of notations that can be used to model systems that are distributed. In addition, UML provides the ability to introduce rigor into diagrams through its constraint language OCL. In this paper, we present a UML-based Fusion analysis phase by way of a simple bank case study, and we discuss some enhancements that were made in addition to a mapping of notations; our proposal is the first step towards providing a Fusion-based analysis phase which supports high-level modeling of distributed systems.

KEYWORDS Fusion, UML, Analysis, Object-Oriented Software Development.

1 Introduction

The Fusion object-oriented software development method [4] is an integration of a variety of OO methods, including OMT [13], Booch [2] and CRC [18], in a single method framework. It provides a systematic process that takes the developer from requirements analysis to implementation. A downside of Fusion is that it can only be used to develop sequential reactive systems, and certain restricted kinds of concurrent systems.

The Unified Modeling Language (UML) [3] is a rich set of notations, brought about to unify the large number of notations that exist in the object-oriented development method community. It does not prescribe how its notations should be used, and thus methods are free to use it in any way that they deem fit; nevertheless, the target method must adhere to the well-defined semantics of UML.

The UML semantics is defined by a meta-model, which relies heavily on natural language and the Object Constraint Language OCL [17] to define well-formedness. UML's adoption by OMG in November 1997 has seen it since become the defacto standard for industry and academia alike.

The primary purpose for introducing UML into Fusion is to support the modeling of distributed systems, while retaining the systematic process, and separation of concerns of Fusion. Also, UML provides a vehicle to remove some of the known problems from Fusion models, such as unresolved interpretations of the object model [10], the inability of the life cycle model to express certain operation sequences [8], and a general difficulty in using regular expressions for modeling the system life cycle [5].

In this paper, we present a UML-based Fusion analysis phase by way of a simple banking case study and discuss some enhancements that were made in addition to the translation. In essence, we make four contributions to Fusion analysis above and beyond the result of a translation from Fusion to UML notations:

- establishment of precise definitions for actors and system entities and the communication between the two.

1. A shorter version of this paper will be published in "The Second International Conference on The Unified Modeling Language—UML '99, Fort Collins, Colorado, USA, October, 1999".

- formalization of the style and language for writing operation schemas via the introduction of OCL and usage rules.
- the ability to model concurrent interactions with multiple users.
- the ability to model multiple users as instances of actor classes and to identify such instances in order to interact with them (a known problem of Fusion in describing multi-user systems [15]).

This paper is organized in the following way: section 2 presents an overview of the Fusion analysis phase and the proposed mapping between diagrams; section 3 presents and discusses the UML enhanced analysis models for the Bank case study; section 4 is a discussion of some of the issues that arise from this work and compares it with work by other authors; finally section 5 sums up the main results presented in the paper.

2 Fusion analysis and UML

2.1 Fusion analysis

The analysis phase of Fusion abstracts from the internal working of the system, placing a particular emphasis on defining the system interface and the concepts related to the system domain. The dependencies between the models of the Fusion analysis phase (from the static point of view) is shown in figure 1. It is noteworthy that the flow of the elaboration process goes generally in the opposite direction of the dependencies. The concepts of the problem domain are described by the object model. The object model later evolves into the system object model. Neither the classes inside the object model nor inside the system object model have methods assigned to them. The procedure of attaching system operations to classes is deferred until the design phase. Thus, during analysis, the system can be viewed as a functional black-box.

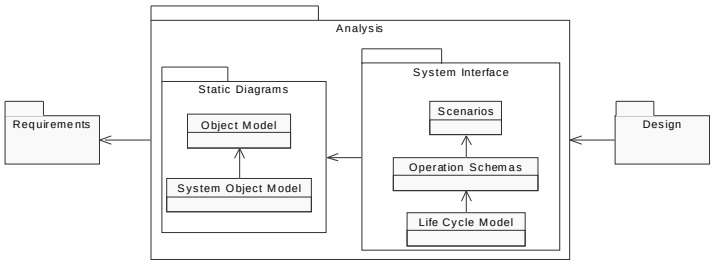


Fig. 1. Fusion Analysis Process

During analysis, Fusion elaborates a comprehensive set of specifications for the system level operations, described by operation schemas. An operation schema specifies the changes in system state and the production of output events by pre- and postconditions. Fusion also specifies temporal sequencing between system operations by regular expressions in the life cycle model. The operation schemas and the life cycle model complement each other well. Scenarios of system usage are used to find the right granularity for system operations and to aid the discovery of the life cycle model. Also, the use of a data dictionary is part of Fusion and it is used throughout the development life cycle.

2.2 Mapping to UML

UML provides a rich set of graphical notations which reflects the wide background and motivations of the UML Partners Consortium. Hewlett Packard's inclusion in the consortium and Fusion's roots in well-known methods has seen the incorporation of most of Fusion's models in UML. Thus, the high-level mapping for the diagrams in common is straightforward. Figure 2 displays the mapping, where the 'Model Name' column indicates the name that we have given to the resulting model.

Operation schemas do not have an equivalent UML diagram because all diagrams in UML are graphical. The developers of UML explicitly omitted textual models from the standard because they were

considered the concern of CASE tools. Operation schemas are integrated into UML by attaching them to the corresponding operation declarations

.

Classic Fusion Model	UML Diagram	Model Name
Object model	Class diagram	Domain class diagram
System object model	Class diagram	System class diagram (SCD)
Operation schema	— (use of OCL)	Operation schema
Scenario	Sequence diagram	Scenario
Life cycle model (reg. expressions)	Statechart diagram	System interface protocol (SIP)
System context diagram	Collaboration diagram	System context diagram

Fig. 2. Fusion to UML notation mapping

3 Bank Case Study

In this case study, we provide a brief statement of requirements and proceed by highlighting the effects of the new notation and modifications on each analysis activity. For reasons of size, however, we omit the domain class diagram and data dictionary. We place a particular focus on the primary analysis deliverables: the system class diagram, the operation schemas and the system interface protocol.

3.1 Problem Statement: Banking System

A bank manages accounts. There are two kinds of accounts: checking accounts and savings accounts. An account is owned by exactly one client, who can in turn own a checking account, or a savings account, or both. The bank processes financial transactions and enquiries: get the balance of an account, withdraw cash from an account, deposit cash into an account, and transfer money from one account to another. A transaction can, therefore, involve one or two accounts. The transactions are recorded, because at the end of each month statements are sent to all clients showing all transactions performed on their accounts during the last period.

We will suppose that all transactions and enquiries are performed by the clerks of the bank on behalf of the clients. The clients are, however, informed directly about the outcome of these transactions and enquiries, for instance by a receipt.

3.2 Class Diagrams

As one would imagine, the UML notation for class diagrams is very similar to the one used by Fusion for its object models. Nevertheless, there are slight differences in the definitions, such as associations and aggregations, but their explanation is outside the scope of this paper.

The domain class diagram, replacing the object model, defines the concepts and their associations related to the application and its domain by defining the classes, attributes, and the relationships between the classes. The Bank domain class diagram is not included here, due to its similarity to the system class diagram.

The system class diagram (SCD), replacing the system object model, may filter or extend the domain class diagram to represent the state space of the system. In the SCD we will use stereotypes to indicate specific semantics for some elements. These new stereotypes are system, actor, and id, as described in figure 3. Note that the term actor corresponds roughly to an agent in Fusion. For instance, a human agent could act as a customer or car driver.

Keyword	Applies to element	Well-formedness rules & rules for general use	Meaning
system	classifier	1. There is exactly one element that is stereotyped 'system' for each SCD. 2. All associations with the system (and those that originate from within the system itself) that go outside the system entity are connected to actors. 3. These associations (defined by 2) are not navigable unless stereotyped 'id' (described below). 4. The system element has a single instance.	It defines an element that is equivalent to the system boundary of Fusion's system object model, except that it can have associations of its own, i.e. the association is attached to the system entity itself. From a behavior point of view the system is reactive.
actor	classifier (stick- figure icon)	1. An actor element can only communicate with the system if an association exists between itself and the system, the system entity itself or an entity within the system. 2. If such an association exists (defined by 1), it implies that the actor can only communicate with the system through the system interface, and thus cannot directly access the internals of the system. 3. An actor element has implicitly zero or many instances.	It defines an entity that provides stimuli to the system. It exhibits active behavior. It does not have visibility into the internal structure of the system.
id	association	1. An 'id' association must be between an actor and the system (either the system itself or an entity within the system). 2. This stereotype places an obligation on the implementation to realize a mechanism which determines a unique actor instance given an object which is part of the system and participates in the association.	An 'id' association serves the purpose of allowing the system to *identify* the actor on the other end of the association.

Fig. 3. Stereotypes on UML Elements of the System Class Diagram

3.2.1 Bank System Class Diagram

The Bank SCD, shown in figure 4, defines accounts, customers, transactions and a calendar as entities of the system, and clerks, clients and a clock as actors. The calendar and clock are present, as they represent time in the system and its continual update. Transaction is a generalization of deposit, withdraw and transfer transactions. Account is a generalization of the Checking and Savings accounts. Constraints and invariants can be placed in UML notes and attached to elements of the SCD, including the system.

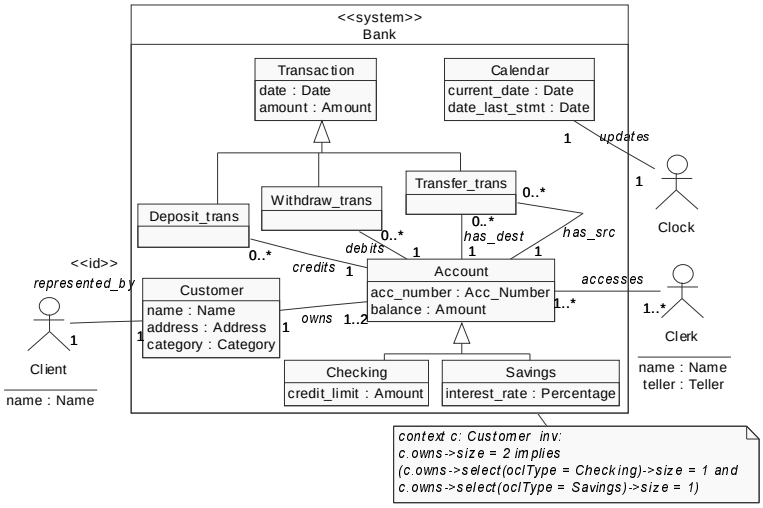


Fig. 4. Bank System Class Diagram

The Bank SCD displays a system invariant, written in OCL in a UML note; it states that any customer who owns two accounts must have one checking account and one savings account. In the OCL expression, the dot operator stands for navigation, for instance from an individual object to a set, e.g., if c is a customer then c.owns is the set of accounts owned by customer c. The select operator is a predefined collection operator of OCL that designates all the elements satisfying a given expression, e.g. select(oclType = Checking). Postfix notation is used for denoting operations on collections, e.g., ->size is predefined and yields the size of the collection which precedes the arrow.

3.3 System Interface

The system interface is defined by the system context diagram, the collection of operation schemas and the life cycle model. Scenarios of system use are constructed in all but the most simple applications; they aid the discovery of the principal models.

3.3.1 Bank System Context Diagram

The system context diagram is a model that has found its way into the method via the recommendation of several authors [16, 12]. It provides a clear view of the system interface, by showing all events exchanged at the system interface, input and output. The Bank system context diagram is shown in figure 5, showing the input and output events at the Bank system interface. We model it in UML using a descriptor-level collaboration diagram, where the half-arrow lines indicate asynchronous events exchanged between the system entity and actor entities. In this diagram, actors represent classes rather than individual instances. For the purpose of conciseness, event parameters are omitted from this diagram.

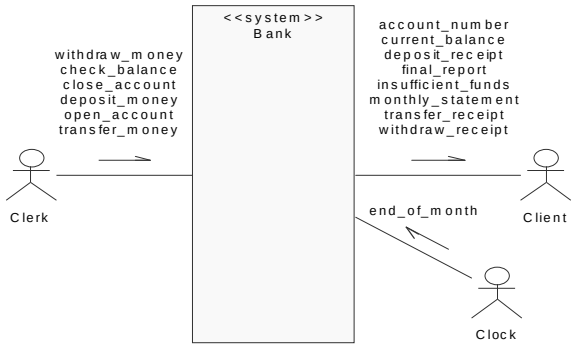


Fig. 5. Bank System Context Diagram

3.3.2 Bank Scenario

Scenarios are used to discover and check the system interface. They are indispensable when the sequencing of the system operations is not intuitive or the system interface contains complex interactions. Also, they provide a tremendous aid in finding the right granularity for the system operations. The scenario in figure 6 illustrates a possible series of interactions between the system, SwissBank, and two clients, Josh and Sophia, speaking to their respective clerks. The clients ask to open accounts and perform transactions on their respective accounts, and the system responds to these requests. At the end of the month, the system sends a statement to the two clients. Normally, many scenarios will be produced, but we show only one here.

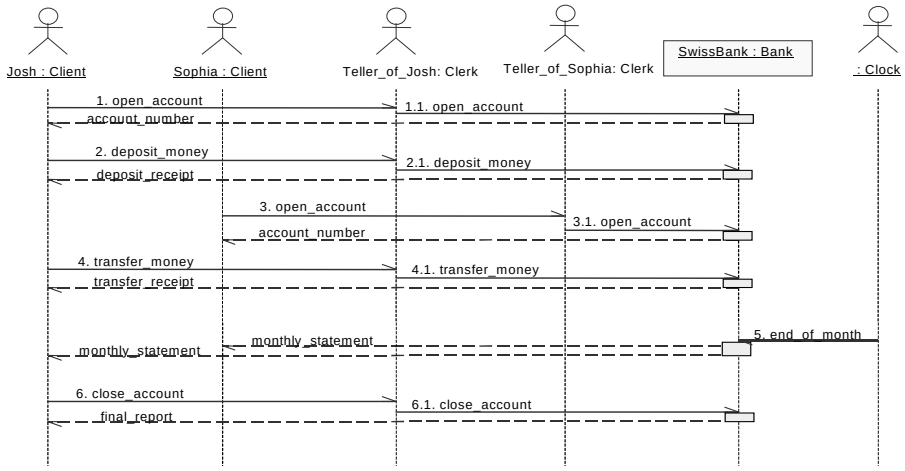


Fig. 6. Bank Scenario

3.3.3 Bank Operation Schemas

An operation schema defines the functionality of a system operation. A schema is written in a declarative style, which details all objects and associations that are read or updated by the operation, events generated by the operation, and the pre- and postconditions ruling the operation, but without showing internal messaging between objects required to realize the operation. The language of operation schemas is left unspecified by Fusion, although they are generally written in natural language.

UML provides the (semi-)formal language OCL that can be used to specify operation schemas. OCL allows the precise description of constraints on the model (invariants), the mention of specific elements (reads and changes clauses) through navigation expressions, and the precise definition of pre- and postconditions (assumes and results clauses).

OCL is a declarative language, where constraints have no side effects and are evaluated atomically. When used as a postcondition, an OCL expression can not change the state of the system and is evaluated on a consistent system state, i.e., no system changes are possible while the expression is evaluated. OCL is also a typed language where types are any kind of Classifier, the basic building block of UML. OCL also provides the benefits of precision and unambiguity associated with a formal language.

For the format of operation schemas we will keep the one used for Fusion. For their contents, however, we will use OCL together with some additional keywords and operators. The clauses of an operation schema have the following meanings and contents:

Operation: displays the operation name. The *triggered_by* keyword is used when the input event that triggers the operation is of a different name—not possible in classic Fusion. If this keyword is not supplied, the triggering event is assumed to have the same name as the operation. The *triggered_by* keyword is followed by a simple or-expression indicating that the operation can be triggered by different events on different executions. Note that any execution of the operation is only triggered by a single event. Also, note that it is perceivable that a single event could trigger two or more events, e.g., end_of_month could trigger account_statements and interest_payment_update. Also, this convention requires all triggering events to have the same parameter signature.

Description: not affected (still written in natural language). It provides a concise description of the operation.

Reads: all objects and object attributes which are read by the operation are declared in this clause. This clause may contain three keywords: *supplied*, *local,* and *identifier*:

supplied indicates an input parameter (comes straight from the classic operation schemas).

local indicates a local variable with scope only within the current operation schema.

identifier indicates an input parameter identifying the sending actor.

Changes: all objects, object attributes, and associations which may be changed by the operation are declared in this clause. The change clause may contain the keyword *new*, indicating that this is a newly created object of the given class. There is no explicit destroy keyword. Destruction is achieved instead by removing the element from the set of instances of its class.

Sends: all output events are specified in the form:

actor_instance(s) '{' output_event(s) '}' ':' (actor_type | 'set(' actor_type ')')
['*such_that*' ocl_constraint_specifying_actor_instance(s)]

The optional OCL constraint defines the individual instance or set of instances of interest. Note that in classic Fusion it is not possible to send an event to a set of actors.

Assumes: OCL precondition, a boolean expression.

Results: OCL postcondition, a boolean expression including the if-then-else[1] conditional construct (highlighted with bold font) and the keyword *sent*, which indicates that an event has been sent to an actor.

Here are some additional comments applying to operation schemas:

- The additional keyword *such_that* can be used in the Reads, Changes, and Sends clauses; it is used to select the exact instance(s) of interest described by an OCL expression.
- There is an implicit logical *and* between blocks within assumes and results clauses.
- bank.customer designates all instances of the class Customer, which can also be written bank.customer.allInstances. Because all expressions are evaluated in the context of the system, the prefix 'bank' can be omitted, i.e. customer is equivalent to bank.customer. As a consequence, class names should not be used as instance names, even with an initial lowercase letter.
- We allow the use of side-effect free functions defined in the data dictionary. A function description consists of a function signature separated by the '==' symbol, followed by the body, which is an OCL expression returning a value compliant with the signature.

We now present three operation schemas of the Bank system: open_account, transfer and account_statements. All other operation schemas of the Bank system can be found in Appendix A.

The open_account operation schema is shown in figure 7. It describes the action of opening a new account for a client. The schema shows the operation parameters and the objects and associations changed via the reads and changes clauses; the *identifier* keyword indicates that the supplied parameter is the identity of the sending client actor. The sends clause indicates that the client actor is sent the account_number event by this operation. No assumptions are imposed (assumes clause is empty). The results clause defines what needs to be accomplished by the operation: a new customer object is created and initialized (establishing an identification link to the actor through the represented_by association) if it doesn't already exist. A new account object is created and initialized; a link between the account and the customer is made; and the client is sent the new account number. The second to last line of the result clause applies the @pre operator: it specifies the set of accounts in the owns association at the start of the operation (x@pre is equivalent to Fusion's old operator e.g. *old* x).

In implemention terms for distributed systems, the identifier parameter (e.g. first line of reads clause in figure 7) could be realized as callback addressing (RPC/RMI) between two different address spaces; in the case of a local call, it will just be factored out. On a similar line of argument, the results clause (line 9):

```
cust.represented_by = cust.represented_by@pre->including(client)
```

would require the identity of the caller to be stored, e.g., a name registry registers the address. Thus, the <<id>> stereotype places an obligation on the implementation to realize a mechanism to determine the identity of a particular actor instance present in the association.

1. This is in addition to the OCL *if-then-else* which returns a value when evaluated.

Operation: open_account
Description: Opens a new account for a client.

Reads: *identifier* client: Client -- identifier and not the actor instance itself
supplied name : Name
supplied address : Address
supplied category : Category -- indicates the category that the client belongs to—relates to the conditions of the account
supplied type_of_account : Account_type
account

Changes: *new* a: Account *such_that* (type_of_account = #Savings *implies* a.oclType = Savings *and*
type_of_account = #Checking *implies* a.oclType = Checking)
new cust : Customer -- implies that Customer.allInstances is affected (similarly for account)
owns : (Customer, Account) -- association ends are shown for typing and easy reference
represented_by : (Client, Customer)

Sends: client {account_number} : Client -- no navigation expression because the client is supplied as parameter

Assumes: -- invariants are implied in both assumes and results clauses

Results: **if** customer.represented_by->exists(x | x = client) **then** -- if the customer already exists then ... else
-- also note customer.represented_by is an OCL shorthand for customer->collect(represented_by)
customer->select(x | x.represented_by = client)->includes(cust)
else
cust.oclIsNew -- OCL property which evaluates to true if the object is created during the operation
cust.name = name
cust.address = address
cust.category = category
cust.represented_by = cust.represented_by@pre->including(client) --association stereotyped com_id
endif
a.oclIsNew
has_unique_num(a, account) -- a side-effect free function: see below
a.balance = 0
if a.oclType = Savings **then**
a.interest_rate = interest_rate(category) -- see below
else -- a.oclType = Checking
a.credit_limit = credit_limit(category) -- maps a category to a credit limit
endif
cust.owns = cust.owns@pre->including(a) -- new account is linked to its owning customer through owns association
sent client account_number(a.acc_number)

Data Dictionary (extract)
-- references to rules or procedures are defined as side-effect freefunctions—defined in the Data dictionary; avoids clutter and allows reuse.
function
has_unique_num(new_a: Account, accs: Set(Account)): Boolean ==
accs->forall(a | a.num = new_a.num implies a = new_a)

-- The above is a somewhat weak option, one could equally assert the below (on account@pre). This would, of course, depend on what is required.
has_num_one_greater(new_a: Transaction, accs: Set(Account)): Boolean ==
accs->forall(a | a.num < new_a.num) *and* accs->exists(a | a.num + 1 = new_a.num)

interest_rate(cat: Category): Percentage ==
(*if* cat = #gold *then* 5 *else* (*if* cat = #classic *then* 3 *else* 2))

Fig. 7. open_account operation schema

The transfer_money operation schema is shown in figure 8. It describes the action taken to transfer money from one account to another provided that the source account has sufficient funds. The reads clause contains a local boolean declaration, which is used to improve the readability of the schema. In the results clause, if the source account has sufficient funds then the money is transferred from the source to the destination account, a new transaction is created and linked to the involved accounts, and a receipt is sent to the source account owner; otherwise the source account owner is sent an insufficient funds event.

Operation: transfer_money
Description: Transfer an amount of money from a given account to another, only if the amount of money does not exceed the source account's limit.

Reads: *supplied* src_a: Account
supplied dest_a: Account
supplied amount: Amount
calendar.date : Date
local src_has_suff_funds: Boolean =
(src_a.oclType = Savings *and* src_a.balance >= amount) *or*
(src_a.oclType = Checking *and* src_a.balance + src_a.credit_limit >= amount)

Changes: *new* t : Transfer_trans
src_a.balance, dest_a.balance: Amount
has_src, has_dest: (Transfer_trans, Account)

Sends: client {transfer_receipt, insufficient_funds} : Client *such_that*
src_a.owns.represented_by->includes(client) -- client owner of src account is sent the event

Assumes:

Results: **if** (src_has_suff_funds) **then**
src_a.balance = src_a.balance@pre - amount
dest_a.balance = dest_a.balance@pre + amount
t.oclIsNew -- OCL property which evaluates to true if the object is created during the operation
t.date = calendar.date
t.amount = amount
src_a.has_src = src_a.has_src@pre->including(t) -- the new transaction is linked to its source account
dest_a.has_dest = dest_a.has_dest@pre->including(t) -- the new transaction is linked to its dest account
sent client transfer_receipt(src_a, dest_a, amount)
else -- not src_has_suff_funds
sent client insufficient_funds(src_a, amount)
endif

Fig. 8. Transfer_money operation schema

The account_statements operation schema is shown in figure 9. It describes the action of sending out account statements at the end of the month. The *triggered_by* keyword in the operation clause indicates that the name of the triggering event is end_of_month whereas the name of the operation is account_statements. The first line in the results clause states that a statement is sent out for each account to its owner; the statement details all the transactions since the last statement.

Operation: account_statements
triggered_by end_of_month

Description: Statements are sent to all account owners, detailing their respective transactions over the last period (i.e., since last statement).

Reads: calendar.current_date: Date
account -- all accounts, equivalent to Account.allInstances
transaction -- all transactions, equivalent to Transaction.allInstances

Changes: calendar.date_last_stmt : Date

Sends: clients {monthly_statement} : set(Client) = customer.represented_by -- sent to all clients

Assumes:

Results: account->forall(a | *sent* a.owns.represented_by monthly_statement(a,
last_period_transactions(a, transaction, calendar.date_last_stmt@pre)))
-- a statement (details all transactions of an account since the last period) is sent to the owner of each account
calendar.date_last_stmt = calendar.current_date

-- defined in Data dictionary
function
last_period_transactions(a:Account, trans: Set(Transaction), earliest: Date): set(Transaction) ==
trans->select(t | t.date >= earliest *and* (t.credits = a *or* t.debits = a *or* t.has_src = a *or* t.has_dest = a))

Fig. 9. Account_statement operation schema

3.3.4 Bank Life Cycle Model

The Fusion life cycle model defines the behavior at the system interface. It abstracts all possible scenarios by describing all possible sequences of operations at the system interface. The usefulness of the life cycle model should not be underestimated. The life cycle model becomes very important in systems that exhibit complex interactions, especially when there are too many possible scenarios.

The regular expression format of life cycles is not found in UML. The closest UML equivalent is the statechart diagram used as a protocol specification. We prefer statechart diagrams over activity diagrams, the other possible choice, because the former are more concise when modeling event-level interactions. A protocol specification captures all the external interactions of the system, leaving all internal transitions invisible.

The typical use of state diagrams as low-level behavior descriptions of objects is, therefore, not the intended use here. In short, we will describe the system interface protocol (SIP) with UML statechart diagrams used as protocol specifications. Note that the SIP offers additional expressiveness over regular expressions, which we will demonstrate in what is to follow.

Lets now contrast the old model and notations against the new ones. We illustrate this by the classic Fusion Bank life cycle model (figure 10) and a more expressive SIP that is able to express concurrent account activities (figure 12). The life cycle model (figure 10) displays the sequencing of the bank operations and output events by way of four operators: sequence '.', alternative '|', optional repetition '*', and interleaving '||'. The prefixed '#' symbol indicates an output event. Subexpressions, e.g. Enquiry, are used as a structuring mechanism, so that the life cycle model is more readable and manageable.

```
Bank: ((Initialization | Transaction | Enquiry | Finalization)* || Statements)
Initialization = open_account . #account_number
Transaction = withdraw_money . (#withdraw_receipt | #insufficient_funds)
            | deposit_money . #deposit_receipt
            | transfer_money . #transfer_receipt
Enquiry = check_balance . #current_balance
Statements = account_statements . #monthly_statement*
Finalization = close_account . #final_report
```

Fig. 10. Classic Life Cycle for Bank

Before we display and discuss the SIP model, we show in figure 11 the notations used in UML statechart diagrams, the restrictions we apply on them, and their relations to Fusion terms.

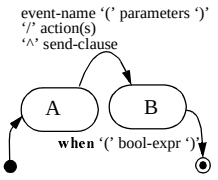
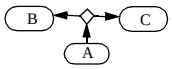
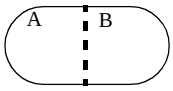
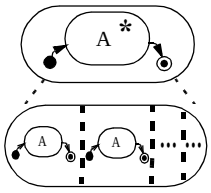
Notation	System interface protocol terms and constraints
	• Input events are mapped to transition triggering events. Parameters are optional, and should generally be omitted for the sake of readability, since they are found in the operation schemas anyway. • System operations are mapped to actions on transitions. *Convention*: no action clause implies that the operation name is the same as the input event name. • Output events are placed in the send-clause. A send-clause is a special action on transitions, indicating the destination of the event. • A simple state (e.g. A) is interpreted as a wait state for the next event. • Initial and Final states are shown with filled circles and bulls-eyes respectively. • A change events (**when**) is triggered when its boolean condition becomes true.
	A diamond symbol represents a branching transition. Branches are strictly used to differentiate possible output events of a system operation. The conditions for the branches are not shown in SIPs to avoid redundancy with the operation schemas.
	Concurrent state machines (so called AND-states) are denoted by a thick dotted line dividing the parent state machine, which means the parent is divided into two or more concurrent submachines whose initial transitions are triggered at the same time as the parent state machine is entered.
	A star, or more generally a multiplicity string, in the upper right hand corner denotes a dynamic state machine. It specifies the number of concurrently executing identical state machines. These concurrent state machines can start execution at different times, however for completion they have to wait for each other. The multiplicity of the dynamic state machine is resolved when entering the state and is equal to the number of argument lists supplied to it. Dynamic state machines are used in UML activity diagrams and are not strictly speaking part of UML statechart diagrams.

Fig. 11. System interface protocol mapping

Figure 12 shows the SIP for the bank system. The system comprises two concurrent state machines, Account_activity and Statement_activity. Account_activity in turn is a dynamic state machine. Thus, many account activities can operate concurrently to each other.

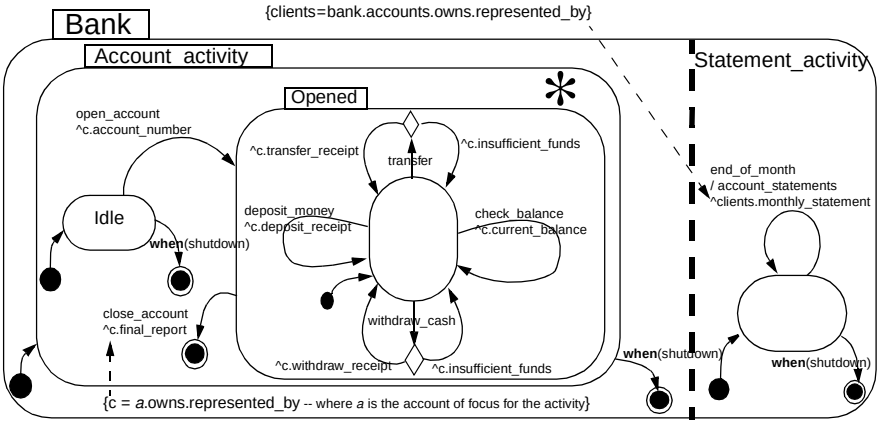


Fig. 12. Bank system interface protocol

According to the definition of a dynamic state machine, the multiplicity of its concurrent state machines must be resolved upon entering it, i.e., the number of state machines that are spawned, when the dynamic state machine is entered, must be known. It is, however, not possible to predict how many accounts will be opened during the lifetime of the bank. A possible, not ideal, solution is to work with a fixed upper limit for the multiplicity and therefore the maximum number of accounts.

As explained in figure 11 we can exit the dynamic state machine Account_activity only when all its sub-machines have completed. This means that we have to take care of the cases where the account is never opened, which are handled by the transition when(finished) exiting from the Idle state.

In Statement_activity, we can see that it is possible to differentiate the triggering event name from the executing operation name; end_of_month/account_statements is such an input event/operation pair. It is also important to note that the send-clause shows that an event of type monthly_statement is sent to each client of the bank, the set of these clients being defined by an OCL expression as the account owners.

4 Related Work and Discussion

4.1 Related Work

Atkinson [1] gives a broad overview of a Fusion-based method that uses UML notations, called FUML. He proposes a split in the analysis phase: conceptualization and detailed analysis. In our approach, we do not propose an explicit split but, as we will discuss below in section 4.3, we make a difference in the level of detail between models within analysis. Also, FUML proposes the use of a design class diagram in the design phase models, a proposal we are in agreement with. A design class diagram allows one to make a clean separation between analysis and design classes. There is no explicit design class diagram in classic Fusion. The method uses the analysis class diagram as the basis for design, allowing only limited additions to the diagram, an approach which does not really scale well to large complex systems[1].

Lano et al. [12] make an interesting comparison between Fusion and Syntropy and roughly outline a method out of the two, which includes a few supplementary models. Although, they do not go into the

1. Our observation is that as software becomes more reusable, flexible, robust etc., supports non-functional requirements, contains features such as dynamic reconfiguration (e.g., plug-and-play), the more the design class structure changes wrt. that of the analysis (SCD).

details of their method, the goal of their proposal relates closely to ours: retaining the systematic approach of Fusion while injecting rigor; they achieve this rigor via Syntropy, which uses an early form of OCL.

Catalysis [9] is a method that has roots in Fusion [8], and advocates rigor through constraints and refinement, and reuse through frameworks.

We claim that the proposals by Atkinson and Lano et al. do not provide the same level of detail as we do in this paper, whereas the Catalysis approach indeed offers a direct alternative to our approach.

4.2 System Interface Protocol

As we have shown, the addition of dynamic state machines is not well suited to model the behavior of multiple concurrent users, such as clients opening, managing and closing their accounts independently of one another. Because of the importance of dynamic actors in distributed systems, this problem is worth to be solved.

As stated we use UML statecharts with limited capabilities because we want only to model the interface protocol of the system. The problems related to the dynamic creation of activities are due to the constraint not to expose the internal working of activities, since without that constraint it would be possible to dynamically create active objects.

In fact, this is the approach taken by Catalysis. Catalysis goes even further by attaching statecharts to types instead of classes[1]. The claimed advantage of attaching statecharts to types is that as design evolves and classes are added and changed the analysis types containing the statecharts are unaffected. However, this approach requires one to assign behavior to types, which in many cases may be premature. Thus, one loses the flexibility that one would have when making the SIP independent of any type or class. Together with Fusion we believe that a clear separation between analysis and design activities promotes cleaner and more effective models. We agree with Fusion which claims that responsibilities should be allocated during design rather than analysis.

We, therefore, believe the most effective approach is to model the SIP independent of analysis classes/types. Thus, a proposal for enhancing UML statecharts with dynamic creation of activities is required. Such a proposal is the subject of our current work.

4.3 Operation Schemas

The approach we propose for describing the contents of operations schemas provides a precise and consistent notation that will allow them to play a greater role in verification and testing. The replacement of natural language with OCL does, however, make operation schemas harder to produce. This can be remedied by the intermediate development of use cases between functional requirements capture and work on operation schemas. This is consistent with the findings of Coleman [5]. An expansion on this idea is presented in [11], and is an area of our current research.

Catalysis promotes a generalized notion of operation schema, named action: a localized action is equivalent to an operation schema and a joint action to a use case. However, we are not sure that these additional concepts make it easier to specify system operations. We claim that the language and guidelines that we use for specifying operations are more precise than the ones used by Catalysis, almost only implicitly defined by examples. Finally, it might be important to know that Catalysis uses some terms and definitions that are not compatible with the UML/OCL standard.

1. In short, a type is an analysis level class that will typically be refined into a design class at design time, but is not obligated to. Catalysis uses type models rather than class models in analysis.

4.4 In Perspective

Since our work is focused on modeling distributed systems during analysis, this paper defines the initial ground work that we will build upon to fulfil our goal of producing a comprehensive set of guidelines and techniques for modeling distributed systems, while endeavoring to provide a solid foundation for supporting the explicit description of software architectures.

5 Conclusion

A UML-based and enhanced Fusion analysis phase has been demonstrated on a simple bank case study. This new approach resolves many of the problems of the original Fusion analysis phase thanks to the introduction of UML notations and changes to the method itself, and it allows one to take advantage of the rich tool support available for UML.

In essence, we have made four contributions to the Fusion analysis phase above and beyond the result of a translation from Fusion to UML notations:

- establishment of precise definitions for actors and system entities and the communication between the two.
- formalization of the style and language for writing operation schemas via the introduction of OCL and usage rules.
- the ability to model concurrent interactions with multiple users.
- the ability to model multiple users as instances of actor classes and to identify such instances in order to interact with them.

Acknowledgments

The authors would like to thank Mohamed Kandé, Didier Buchs, Jörg Kienzle and the anonymous referees for their helpful comments. This work has been supported partially by the Swiss National Science Foundation, grant no. 20-50635.97.

References

[1] C. Atkinson. *Adapting the Fusion Process to Support the UML*. Object Magazine, Sigs Publications, Nov 1997.

[2] G. Booch, *Object-Oriented Analysis and Design with Applications*, second edition, Addison-Wesley, 1994.

[3] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley 1998.

[4] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall 1994.

[5] D. Coleman. *Fusion with Use Cases - Extending Fusion for Requirements Modelling*. OOPSLA Conference Tutorial Slides 1995 (available at http://www.hpl.hp.com/Fusion/me_books.html).

[6] S. Cook and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall 1994

[7] T. Cotton. *Evolutionary Fusion: A Customer-oriented Incremental Life Cycle for Fusion* in Object-Oriented Development at Work: Fusion in the Real World. pp. 179-202, (eds. R. Malan, R. Letsinger and D. Coleman) Prentice-Hall 1996.

[8] D. D'Souza and A. Wills. *Extending Fusion - Practical Rigor and Refinement* in Object-Oriented Development at Work: Fusion in the Real World. pp. 314-359, (eds. R. Malan, R. Letsinger and D. Coleman) Prentice-Hall 1996.

[9] D. D'Souza and A.Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley 1998.

[10] G. Eckert. *Improving the Analysis Stage of the Fusion Method* in Object-Oriented Development at Work: Fusion in the Real World, pp. 276-313, (eds. R. Malan, R. Letsinger and D. Coleman) Prentice-Hall 1996.

[11] Hewlett Packard. *Engineering Process Summary (Fusion 2.0)*, http://www.hpl.hp.com/Fusion/, 1998.

[12] K. Lano, R. France and J-M. Bruel. *A Semantic Comparison of Fusion and Syntropy*. Object-Oriented Systems, 1998; To be published, http://www.univ-pau.fr/~bruel/publications/oos98.ps .

[13] J. Rumbaugh, M. Blaha, W. Prembalini, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1995.

[14] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley 1999.

[15] S. Sendall, N. Guelfi and A. Strohmeier. *Fusion Applied to Distributed Multimedia System Development: the Easy Meeting Case Study*. Technical Report EPFL-DI No 99/304, Swiss Federal Institute of Technology in Lausanne.

[16] A. Strohmeier. *The Fusion Method, with Implementation in Ada 95*; SIGAda'98 Tutorial Notes, Washington, DC, USA, 1998, ACM Press, 1998.

[17] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley 1998.

[18] R. Wirfs-Brock, B. Wilkerson and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

Appendix A: Bank System Operation Schemas

Operation: check_balance
Description: Enquires the balance of a given account. No record transaction is made.

Reads: *supplied* a: Account

Changes:

Sends: client{current_balance} : Client *in* a.owns.represented_by

Assumes:

Results: *sent* client current_balance(a.acc_number, a.balance)

Operation: deposit_money
Description: An amount of money is placed on a given account.

Reads: *supplied* a: Account
supplied amount: Amount
calendar.date : Date

Changes: a.balance: Amount
new t : Deposit_trans
credits : (Deposit_trans, Account)

Sends: client{deposit_receipt} : Client *in* a.owns.represented_by -- send to the owner of the account a

Assumes:

Results: a.balance = a.balance@pre + amount
t.oclIsNew
t.date = calendar.date
t.amount = amount
a.credits = a.credits@pre->including(t)
sent client deposit_receipt(a, amount)

Operation: withdraw_money
Description: Requests an amount of money to be taken from a given account. Cash is given only if account has sufficient funds.

Reads: *supplied* a: Account
supplied request: Amount
calendar.date : Date

Changes: a.balance: Amount
new t : Withdraw_trans
debits : (Withdraw_trans, Account)

Sends: client{insufficient_funds, withdraw_receipt} : Client *in* a.owns.represented_by

Assumes:

Results: **if** (a.balance@pre >= request) **then**
a.balance = a.balance@pre - request
t.oclIsNew
t.date = calendar.date
t.amount = request
a.debits = a.debits@pre-> including(t)
sent client withdraw_receipt(a, request)
else -- a.balance@pre < request
sent client insufficient_funds(a, request)
endif

Operation: close_account
Description: Closes an account for a client. A final report is output detailing all the transactions since the last statement

Reads: *supplied* a: Account
calendar.date_last_stmt : Date
transaction
local last_period_trans: set(Transaction) =
last_period_transactions(a, transaction, calendar.date_last_stmt) -- function defined in Data Dictionary

Changes: cust : Customer = a.owns
account
customer
owns: (Customer, Account) -- cust object disappears from the association
represented_by: (Client, Customer) -- cust object disappears from the association

Sends: client {final_report} : Client in cust.represented_by

Assumes: a.balance = 0

Results: account = account@pre->excluding(a) -- a is taken out of commission
if cust.owns@pre->size = 1 **then** -- if cust has only account a then
customer = customer@pre->excluding(cust)
endif
sent client final_report (a, last_period_trans)

Operation: tick
Description: The current system date is incremented.
Reads:
Changes: calendar.current_date : Date
Sends:
Assumes:
Results: calendar.current_date = increment(calendar.current_date@pre) -- increment defined in Data Dictionary

---