An Exercise in Fault Tolerance

Einar Broch Johnsen
Department of Informatics, University of Oslo

Abstract

In this paper fault tolerance issues are addressed in system development using the specification language OUN [8]. We illustrate through an example how exception handling can be introduced into the design. This enables us to start with a perfect design, introducing faults gradually. We aim at a precise, formal understanding of this development, expressed as a refinement relation, that is lacking in less formal specification languages such as UML [2] and OCL [10].

1 Introduction

The purpose of constructing fault tolerant systems is to improve system dependability by giving the system an ability to operate in the presence of (some) faults. Specifying fault tolerant systems usually consists of recognizing certain exceptions prior to designing the system, as in [6]. More recently [1] has proposed a methodology for a gradual introduction of fault handling. The idea is to wrap the original system in layers that detect and handle exceptional behavior, thus incrementally enhancing system performance for each layer.

At the level of specification languages, it is assumed that we know how to detect faults. Hence there is no conceptual distinction between hardware failure and software error. In this paper aspects of this stepwise approach to fault tolerance are illustrated in the OUN specification language, with an example where a teller machine system is gradually developed. We generally use abstract notation for readability and modify the exact definitions of the abstract events when needed, to facilitate specification reuse. The OUN specification language can be seen as a complement to UML and system descriptions in UML can in fact be translated into the language in a semiautomatic manner [9].

Within OUN, the first development steps of a fault tolerant system is explored, focusing on fault handling as an interaction refinement relation in the system development. This formal approach seems to be lacking in the literature [2], [10], [3]. In our example, we introduce dispenser breakdown into the teller machines.

1.1 The Specification Language OUN

The specifications are made in OUN, a formal specification language relying on trace semantics similar to CSP [4] without refusals and divergence, but extended to deal with object identity. The language describes objects typed by interfaces, but unlike for instance Java interfaces an interface is a behavioral description of a particular aspect of an object.

There are independent inheritance hierarchies for classes and interfaces, reflecting the difference between code reuse and behavioral refinement. In this paper we are concerned with an aspect of the behavioral refinement, and we only work with interfaces. Object behavior is described through interfaces and contracts defining observable behavior.

An interface consists of formal parameters, method declarations, inheritance, a WITH-clause (cointerface declaration) and an assumption and an invariant. The formal parameters, method declarations, inherited interfaces and the WITH-clause together define an alphabet of communication events visible to the interface. In the alphabet, method calls are described by initiation and termination events. The assumption and invariant predicates range over finite sequences over such (interface) alphabets, called traces, using the keywords **asm** and **inv** respectively. The assumption is a predicate describing the minimal requirements to behavior of the environment, i.e. the obligation of every object communicating with the interface, whereas the invariant is guaranteed to hold for the entire trace until the assumption is broken. These predicates define a set of traces describing legal "runs" of objects implementing the interface, restricted to the relevant alphabet. The specification language OUN is discussed in [7] and [8]. Necessary details of the specification language will be explained as we proceed.

Our specifications are developed as follows. First we specify the alphabets of the interfaces, then we introduce a contract describing their interaction. We use the keyword **respects** to indicate that an interface must not violate a contract. As no assumption or invariant is given for the interface, it is understood that the interface inherits the invariant of the contract, restricted to the relevant alphabet. By contrast, the keyword **refines** states that this relationship must be proved.

A contract represents a behavioral commitment for objects of the interfaces involved and generates proof obligations within the formal development of the system. When the obligations are fulfilled, the composition of the objects of the given interfaces involved satisfies the contract. In the following example, the proof obligations are generally quite straightforward and will not be discussed further. Abstract syntax is used to make our specifications clear and easy to read. Auxiliary functions are introduced by the keyword **where**, used within the specifications only. We will often use these functions to declare the scope of variables or to make a logically separate part of a specification intuitive and distinct, so they can be reused through the keywords **respects** and **refines**, and we avoid redefining similar abstract notions. As with class inheritance in object oriented languages, auxiliary functions are presumed to be the same unless redefined. We will often, however, need to perform minor changes following alphabet expansion.

1.2 The Example: A Teller Machine System

A teller machine system is specified, consisting of three kinds of objects: users, teller machines and a center. We will also specify an object representing repair actions when we begin to consider faults in the original system. We assume that each teller machine only communicates with one user (at a time) and the center. The center may communicate with several teller machines. There is only one center.

We place the following assumptions on the teller machine system:

- The teller machine never runs out of money.
- The teller machine never dispenses more than 3000 units per request, but may repeat this operation several times in one session.

- The teller machine never dispenses more than the amount on the user's account.

Our approach to the design consists of leaving control with the teller machines. The user and the center are thus regarded as passive objects, when ignoring interactions irrelevant to the teller machine. Objects of these classes are solely reacting to method calls initiated by the teller machine object, which is in the middle of the communication link. We will declare all methods in the user and center objects and all calls are from the teller machine to one of these. Thus the teller machine has no visible methods. We specify one interface for each role of the teller machine and we would use the multiple inheritance property of subinterfacing to obtain a specification of the teller machine class from the partial descriptions. To show that there is a specification of a class that supports a given set of interfaces is an additional difficulty and class implementation as such is beyond the aim of this paper.

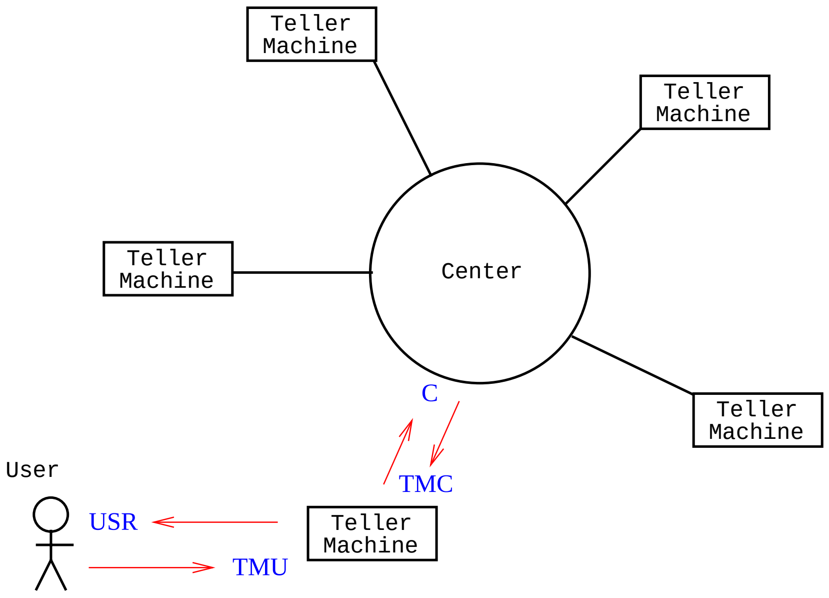

Figure 1: *The teller machine system, illustrated by objects and interfaces. The specification leaves the teller machine in control of the transactions, regarding the user and the center as passive objects.*

2 Patterns

In the specification language, legal traces are described by assumption and invariant predicates. Traces are sequences of events and the predicates are conveniently described by a kind of regular expressions, called patterns. We typically describe traces by prefixes of a pattern. This is an extension to regular expression matching as follows: A trace t is the prefix of a pattern if it can be extended to give an exact match, i.e. t is the prefix of a trace matching the pattern in the usual sense. We use the notation

$$\mathcal{H} \ \mathbf{prp} \ P$$

to denote this predicate, where $\mathcal{H}$ ranges over traces and P is a pattern. A pattern can consist of the following notation:

- Events $a, b, \ldots$ are patterns.
- $a\ b$ is a pattern describing a pattern a followed by a pattern b.
- $a|b$ is a pattern describing a nondeterministic choice between patterns a and b.
- a^+ is a pattern describing 1 or more occurrences of the pattern a.
- a^* is a pattern describing 0 or more occurrences of the pattern a, so $a^* = a\ a^+ = a^+\ a$. We will often decompose patterns of the form a^* to either $a\ a^+$ or $a^+\ a$ when we refine our specifications to introduce faults.
- $a^?$ is a pattern describing zero or one occurrence of the pattern a.

Patterns have (formal) parameters, declared by scoping rules. We use the scoping mechanism to declare the type and the scope of variables. If we want to describe a loop, where the events take different argument values for each iteration, we do this by

$$(a(x) \bullet x : T)^*$$

for some regular expression a depending on a variable x of type T. The idea is that if the scope of the variable is the entire trace, it is denoted

$$(a(x))^* \bullet x : T,$$

so we get a notation that specifies the scope we want to express. In this way we obtain a pattern recognition mechanism on the arguments of the events we allow. The trace

$$a(1)\ b(1)\ a(2)\ b(2)$$

matches the pattern $(a(x)\ b(x) \bullet x : \text{Nat})^*$, but not $(a(x)\ b(x))^*\ x : \text{Nat}$. The arguments of the events are either left unspecified by underscore and ignored or they are used in the process of pattern recognition. We define subpatterns by means of auxiliary functions. Such auxiliary definitions are inherited in subinterfaces and may be reused there for specification purposes.

3 Intolerant Design

Following the development strategy described, we first consider the ideal situation, where no errors may occur. Errors and their handling are introduced later.

3.1 Interaction between the Teller Machine and the User

Let the interaction between the teller machine and the user consist of the following operations:

- dispense: The action of dispensing a sum to the user.
- returnCard: The action of returning the card to the user.
- insertCard: The action of inserting the card into the teller machine.
- giveCode: The action of authenticating the user.
- withdraw: The user informs the teller machine of the amount he/she wishes to withdraw.
- query: The menu of the teller machine, where the user chooses what he wishes to do. For this example, there are two possibilities: “wd” (withdraw) and “end” (end session).

One might consider a display-method, giving text messages to the user (between each of the other events). This way events will be accompanied by appropriate text messages on the screen. As they do not influence our reasoning, we will ignore them in the development of the abstract specification.

Let TMU be an interface of the teller machine describing it's interaction with users and let USR be the interface describing the methods of a user. In our approach, as all methods are implemented by the user, input to the teller machine occurs as out-parameters of the methods called in the user (reflected in the termination events) and output from the teller machine occurs as in-parameters to the methods of the user. To achieve this, we place all the operations in the USR interface. The interface of the user becomes

```
interface USR respects USR↔TMU
begin
  with x: TMU
    opr dispense(in sum : nat)
    opr returnCard()
    opr insertCard(out card : Card)
    opr giveCode(out code : String)
    opr withdraw(out sum : nat)
    opr query(out choice : String)
end
```

The interface of the teller machine becomes

```
interface TMU (x: USR) respects USR↔TMU
begin
end
```

The methodless interface gives us a nonempty alphabet because of the formal parameter, so it still has a function as it enables us to specify requirements. We use the keyword **respects** to indicate that the interface must respect the requirement of a contract, TMU↔USR. Appropriate assumptions and invariants for the two interfaces can then be derived from the contract. Let us identify some interaction scenarios.

1. The user attempts to give the correct code for the card but fails. The card is returned to the user and the session terminates.
2. The user succeeds in giving the correct code to the teller machine. The session continues by a query driven menu which ends when the user wishes to end the session. The card is returned and the session terminates.

These scenarios lead to a contract between the interfaces, specified as follows:

```
contract USR↔TMU
begin
  with m: TMU, u: USR
    inv $\mathcal{H}$ prp $(\text{start}\ (\text{withdrawSession}^*\ \text{quit} | m{\leftrightarrow}u.\text{returnCard}))^*$
  where
    start == $m{\leftrightarrow}u.\text{insertCard}(\_)\ m{\leftrightarrow}u.\text{giveCode}(\_)$
    Dispense == $m{\leftrightarrow}u.\text{withdraw}(sum)\ m{\leftrightarrow}u.\text{dispense}(sum) \bullet sum : \text{nat}$
    quit == $m{\leftrightarrow}u.\text{query}(;\text{“end”})\ m{\leftrightarrow}u.\text{returnCard}()$
    withdrawSession == $m{\leftrightarrow}u.\text{query}(;\text{“wd”})\ m{\leftrightarrow}u.\text{withdraw}(\_)^*$ Dispense
end
```

All events here are of the form ↔ (an initiation event immediately followed by the corresponding termination event) and all events are calls from TMU to USR. Thus

$m{\leftrightarrow}u$.insertCard(_) reflects the initiation of a method call to the method insertCard of object *u* by object *m*, followed by it's termination.

If we wish to limit the maximum amount dispensed per request to 3000 units as stated in the informal system description, we can do this in the contract by

$$\text{Dispense} == m{\leftrightarrow}u.\text{withdraw}(sum)\ m{\leftrightarrow}u.\text{dispense}(sum) \bullet sum \leq 3000.$$

We cannot include the limitation on the amount left on the user's account here as this is not visible in the interaction between the teller machine and the user.

3.2 Interaction between the Teller Machine and the Center

Let the interaction between the teller machine and the center C consist of the following operations:

- authorize: Check that a code and a card correspond.
- balanceCheck: Check that an amount can be withdrawn from an account. If the transaction is accepted, withdraw the amount and return true. If not, return false.

We specify the fault intolerant design of this contract following the same design principles, i.e. we leave control with the teller machine. Hence all methods are placed in the center, enforcing the teller machine to be the active object.

```
interface C respects C↔TMC
begin
  with x: TMC
    opr  authorize(in card : String, code : String; out ok : bool)
    opr  balanceCheck(in sum : nat, card : String, code : nat; out permit : bool)
end
```

The interface of the teller machine is empty as expected.

```
interface TMC (x: C) respects C↔TMC
begin
end
```

We want the communication between a teller machine and the center to follow the contract C↔TMC. We assume that the authorization request must be validated before transactions can occur. As required, any number of transactions are allowed and by assumption, although not formalized, the amount *sum* is always less than the amount on the account of the user.

```
contract C↔TMC
begin
  with m: TMC, c: C
    inv $\mathcal{H}$ prp (badCode|goodCodeSession)$^*$
  where
    badCode == $m{\leftrightarrow}c.\text{authorize}(\_,\_;false)$
    goodCodeSession == $m{\leftrightarrow}c.\text{authorize}(card,code;true)$
                           $m{\leftrightarrow}c.\text{balanceCheck}(\_,card,code,\_)^* \bullet card,code : \text{String}$
end
```

3.3 Intolerant System Description

To obtain a description of the entire system, we combine the interactions of the previous contracts. Keep in mind that the contract TMSystem must not invalidate either of the previous contracts when restricted to the appropriate alphabets.

```
contract TMSystem
  refines C↔TMC, USR↔TMU
begin
  with u: USR, m: TMU, TMC, c: C
    inv $\mathcal{H}$ prp $(\text{start}(x,y)$
              $(\text{withdrawSession}(x,y) | \text{badCode}(x,y)) \bullet x : \text{String}, y : \text{nat})^*$
  where
    start $(x,y) == m{\leftrightarrow}u.\text{insertCard}(x)\ m{\leftrightarrow}u.\text{giveCode}(y)$
    goodCode$(x,y) == m{\leftrightarrow}c.\text{authorize}(x,y;true)$
    badCode $(x,y) == m{\leftrightarrow}c.\text{authorize}(x,y;false)\ m{\leftrightarrow}u.\text{returnCard}()$
    goodWithdraw$(x,y) == m{\leftrightarrow}u.\text{query}(;\text{“wd”})\ \text{wrongAmount}(x,y)^*\ \text{Dispense}(x,y)$
    wrongAmount$(x,y) == m{\leftrightarrow}u.\text{withdraw}(sum)\ m{\leftrightarrow}c.\text{balanceCheck}(sum,x,y;false)$
    Dispense$(x,y) == \text{okAmount}(x,y,sum)\ m{\leftrightarrow}u.\text{dispense}(sum) \bullet sum : \text{nat}$
    okAmount$(x,y,sum) == m{\leftrightarrow}u.\text{withdraw}(sum)\ m{\leftrightarrow}c.\text{balanceCheck}(sum,x,y;true)$
    withdrawSession$(x,y) == \text{goodCode}(x,y)\ \text{goodWithdraw}(x,y)^*\ \text{quit}$
    quit $== m{\leftrightarrow}u.\text{query}(;\text{“end”})\ m{\leftrightarrow}u.\text{returnCard}()$
end
```

The keyword **refines** gives us a proof obligation: The new contract refines the old one in the OUN sense of refinement [8]; i.e. a trace that fulfills the new contract must also fulfill the old ones when restricted to their respective alphabets. This relationship must be proved.

4 Dispenser Mechanism Halting Failure (DF)

Faults are simulated through nondeterminism. In this example we consider possible failures in the dispenser mechanism of the teller machine. We modify the safety requirements of the system accordingly. Our new requirement is:

“Neither the bank nor the user can lose money due to failures.”

Assume that occurrences of DF failures can be noticed. Without knowing why (or how) the dispense error has occurred, we know that some remainder of the sum demanded has not been dispensed. The teller machine must then enter a routine for correcting the effects of such a fault. When an error occurs, we expect the teller machine to

- kick out the card (and display “out of service”)
- block the card reader
- send a credit call to the center to return the remainder to the user’s account
- hibernate until repair
- after repair, unblock the card reader and wait for a new card

and the center to

- correct the amount on the account according to the credit operation
- acknowledge the credit method call

The difficulty here is that these problems do not occur within the specification of any of the interfaces. The natural place to specify these requirements are in the TMSystem contract.

4.1 DF Fault Tolerant USR$\leftrightarrow$TMU Contract

We revise the contract USR$\leftrightarrow$TMU so the sum dispensed is less than or equal to the sum demanded by the user. In the description of the contract we introduce a new abstract event, Dispense_{DF}, such that

$$\text{Dispense}_{DF} == m{\leftrightarrow}u.\text{withdraw}(sum)\ m{\leftrightarrow}u.\text{dispense}(sum') \quad \bullet\ sum' \leq sum \leq 3000.$$

At the user side of the teller machine, observe that we already have all the events we need. Since control is placed with the teller machine, blocking the card reader is the same as not calling the `insertCard` method of USR, i.e. the card reader is blocked until start function recommences with the `insertCard` method call. This action is not visible, since it is in fact only a time delay before the loop described by the regular expression recommences. We modify the contract between the USR and TMU interfaces.

```
contract USR↔TMU-DF
begin
  with m: TMU, u: USR
    inv $\mathcal{H}$ prp (start [goodWithdraw$^*$ (badWithdraw | quit)|returnCard()])$^*$
  where
    start == $m{\leftrightarrow}u$.insertCard(_) $m{\leftrightarrow}u$.giveCode(_)
    goodWithdraw == $m{\leftrightarrow}u$.query(;“wd”) $m{\leftrightarrow}u$.withdraw(_)$^*$ Dispense
    Dispense == $m{\leftrightarrow}u$.withdraw($sum$) $m{\leftrightarrow}u$.dispense($sum$) $\bullet\ sum$ : nat
    Dispense$_{DF}$ == $m{\leftrightarrow}u$.withdraw($sum$) $m{\leftrightarrow}u$.dispense($sum'$)
                                                      $\bullet\ sum' \leq sum \leq 3000$
    badWithdraw == $m{\leftrightarrow}u$.query(;“wd”) $m{\leftrightarrow}u$.withdraw(_)$^*$
                                                 Dispense$_{DF}$ $m{\leftrightarrow}u$.returnCard()
    quit == $m{\leftrightarrow}u$.query(;“end”) $m{\leftrightarrow}u$.returnCard()
end
```

4.2 DF Fault Tolerant C$\leftrightarrow$TMC Contract

An additional method is needed in the interface C:

- credit: Credits an account with a sum.

We expand the invariant of the C$\leftrightarrow$TMC contract accordingly.

```
contract C↔TMC-DF
begin
  with m: TMC, c: C
    inv $\mathcal{H}$ prp (badCode | goodCodeSession)$^*$
  where
    badCode == $m{\leftrightarrow}c$.authorize(_,_; $false$)
    goodCodeSession == goodCode($card, code$) balance($card, code$)$^*$
                                   $m{\leftrightarrow}c$.credit(_,$card$;$true$)$^?$ $\bullet\ card$ : String, $code$ : nat
    goodCode($card, code$) == $m{\leftrightarrow}c$.authorize($card, code$;$true$)
    balance($card, code$) == $m{\leftrightarrow}c$.balanceCheck(_,$card, code$,_)
end
```

The credit operation should be triggered by dispenser failure, which is invisible to this contract, so we cannot be more precise regarding the occurrences of credit events here. The abstract notation of the contract is used in the modified specification of interface C.

interface C-DF **respects** C↔TMC-DF
begin
  with x: TMC
    opr authorize(**in** *card*: String, *code*: String; **out** *ok*: bool)
    opr balanceCheck(**in** *sum*: nat, *card*: String, *code*: nat; **out** *permit*: bool)
    opr credit(**in** *sum*: nat, *card*: String, ; **out** *ok*: bool)
  asm $\mathcal{H}$ **prp** $(\text{badCode} \mid \text{goodCodeSession})^*$
  inv true
end

We would like to specify precisely the amount credited to the users account, but this cannot be done here as it is outside the scope of the alphabet. We could, however, modify the specifications above to state that the sum credited to an account cannot exceed the sum demanded, i.e.

$$\begin{aligned}\text{goodCodeSession} == {} & \text{goodCode}(card, code)\\ & \text{balance}(card, code)^*\\ & \quad (m{\leftrightarrow}c.\text{balanceCheck}(sum1, card, code, \_)\\ & \qquad m{\leftrightarrow}c.\text{credit}(sum2, card; true))^? \bullet sum2 \le sum1\end{aligned}$$

4.3 Repair of the Broken Dispenser

We need a way of handling the repair of the teller machine. To do this, we will simply assume that we have an interface R and a method

- repair: The atomic action of repairing a broken dispenser.

To follow our specification concept, this method will be implemented by some object with the R interface, and we will give a contract between an appropriate interface TMR of the teller machine and the interface R. The interface of the teller machine is defined in the usual way.

interface TMR (x: R) **respects** R↔TMR
begin
end

However, when we consider the interface R, we have the possibility of distinguishing the pointwise behavior vis-à-vis a teller machine and the possible interleaving of requests for repair from different teller machines. For our purpose we don't need to be very precise about this interleaving, so we propose

interface R **respects** R↔TMR
begin
  with x: TMR
    opr repair()
  asm $\mathcal{H}$ **prp** $x{\leftrightarrow}me.\text{repair}()^*$
  inv true
end

The contract simply states that a broken teller machine must be repaired before it can break down again, but does not impose any interleaving of requests from different teller machines.

```
contract R↔TMR
begin
  with m: TMR, r: R
    inv $\mathcal{H}$ prp $m{\leftrightarrow}r$.repair()$^*$
end
```

4.4 DF Fault Tolerant System Description

Now we want to modify the contract for the entire teller machine system to handle DF errors. We expand the contract TMSystem by the abstract event badWithdraw, which is slightly modified.

```
contract TMSystem-DF
begin
  with u: USR, m: TMU, TMC, TMR, c: C, r: R
    inv $\mathcal{H}$ prp (start$(x,y)$ [goodCode$(x,y)$
                goodWithdraw$(x,y)^*$
                  (badWithdraw$(x,y)$ | quit)|badCode$(x,y)$] $\bullet\, x$ : String, $y$ : nat)$^*$
  where
    start$(x,y)$ == $m{\leftrightarrow}u$.insertCard$(x)$ $m{\leftrightarrow}u$.giveCode$(y)$
    goodCode$(x,y)$ == $m{\leftrightarrow}c$.authorize$(x,y;true)$
    badCode $(x,y)$ == $m{\leftrightarrow}c$.authorize$(x,y;false)$ $m{\leftrightarrow}u$.returnCard()
    goodWithdraw$(x,y)$ == $m{\leftrightarrow}u$.query(;“wd”) wrongAmount$(x,y)^*$ Dispense$(x,y)$
    wrongAmount$(x,y)$ == $m{\leftrightarrow}u$.withdraw$(sum)$
                          $m{\leftrightarrow}c$.balanceCheck$(sum,x,y;false)$ $\bullet\, sum$ : nat
    okAmount$(x,y,sum)$ == $m{\leftrightarrow}u$.withdraw$(sum)$ $m{\leftrightarrow}c$.balanceCheck$(sum,x,y;true)$
    Dispense$(x,y)$ == okAmount$(x,y,sum)$ $m{\leftrightarrow}u$.dispense$(sum)$ $\bullet\, sum$ : nat
    Dispense$_{DF}(x,y,sum-sum')$ == $m{\leftrightarrow}u$.withdraw$(sum)$ $m{\leftrightarrow}u$.dispense$(sum')$
                                                            $\bullet\, sum' < sum \leq 3000$
    quit == $m{\leftrightarrow}u$.query(;“end”) $m{\leftrightarrow}u$.returnCard()
    badWithdraw$(x,y)$ == $m{\leftrightarrow}u$.query(;“wd”) wrongAmount$(x,y)^*$
                          Dispense$_{DF}(x,y,sum)$ $m{\leftrightarrow}u$.returnCard()
                                    $m{\leftrightarrow}c$.credit$(x,sum)$ $m{\leftrightarrow}r$.repair() $\bullet\, sum$ : nat
end
```

The larger alphabet of this contract enables us to specify that the amount credited to the account of the user in the case of dispenser failure is the remainder between the sum demanded and the sum dispensed.

The fault tolerant specification we have obtained is a refinement of the ideal specification by the traditional OUN refinement relation (cf. section 3.3): Our traces belong to the intolerant specification when we hide the new events.

5 Conclusion

Fault tolerance issues in the specification language OUN are handled by gradually developing interface specifications. As we have seen in the example, this development involves possible alphabet extensions. Changes in the alphabet are usually considered an unwanted complication with regard to refinement issues, but in the context of distributed, or object oriented systems, the OUN notion of refinement as “considering more details” seems rather intuitive. In our example, this refinement notion covers the idea of introducing faults as well, since the new events come in pairs of initiation and termination events. In this example we have non-masking fault tolerance [1]. A precise, formal refinement relation between specifications is thus suggested which covers this kind of fault tolerance issues.

Other faults can be introduced, for instance line failure can be handled through an additional event representing time-out termination, giving a natural extension to the faults introduced here. In this case, however, the new events lack the duality between initiations and terminations of our example, and a weaker refinement relation must be used, stating that the system returns to a situation where fault-free behavior is possible after the fault has been handled. Such cases and their refinement relations will be explored in a later paper [5].

6 Acknowledgment

I am grateful for discussions in the ADAPT-FT project group. In particular, Olaf Owe, Ellen Munthe-Kaas and Jüri Vain have provided valuable advice.

References

[1] A. Arora and S.S. Kulkarni, *Component Based Design of Multitolerant Systems*, IEEE transactions on Software Engineering, 24(1), pp. 63-78, January 1998.

[2] G. Booch, J. Rumbauch and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[3] G. Dondossola and O. Botti, *System Fault Tolerance Specification: Proposal of a Method Combining Semi-formal and Formal Approaches*, in Proceedings of FASE’00, LNCS 1783, Springer Verlag, 2000.

[4] C.A.R. Hoare, *Communicating Sequential Sequences*, Prentice-Hall, 1985.

[5] E.B. Johnsen, E. Munthe-Kaas, O. Owe and J. Vain, *Fault tolerant design in OUN*, manuscript, 2000.

[6] L. Lamport and S. Merz, *Specifying and Verifying Fault-Tolerant Systems*, in Proceedings of the Third International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863, Springer Verlag, 1994.

[7] O. Owe and I. Ryl, *On Combining Object Orientation and Reliability*, in Proceedings of Norsk Informatikk Konferanse NIK’99, Norway, November 1999.

[8] O. Owe and I. Ryl, *A notation for combining formal reasoning, object orientation and openness*, Research report no. 278, Department of Informatics, University of Oslo, 1999.

[9] I. Traoré and K. Stølen, *Towards the definition of a platform supporting the formal development of open distributed systems*, Research report no. 271, Department of Informatics, University of Oslo, 1999.

[10] J. Warmer and A. Kleppe, *The Object Constraint Language — Precise Modeling with UML*, Addison-Wesley, 1999.