Incremental Fault-Tolerant Design in an Object-Oriented Setting *

Einar Broch Johnsen, Olaf Owe, Ellen Munthe-Kaas
Department of informatics
University of Oslo
PO Box 1080 Blindern, N-0316 Oslo, Norway
{einarj, olaf, ellenmk}@ifi.uio.no

Jüri Vain
Inst. of Cybernetics
Tallinn Technical University
Akadeemia tee 21, 12618 Tallinn, Estonia
vain@ioc.ee

Abstract

With the increasing emphasis on dependability in complex, distributed systems, it is essential that system development can be done gradually and at different levels of detail. In this paper we propose an incremental treatment of faults as a refinement process on object-oriented system specifications. An intolerant system specification is a natural abstraction from which a fault-tolerant system can evolve. With each refinement step a fault and its treatment are introduced, so the fault-tolerance of the system increases during the design process.

Different kinds of faults are identified and captured by separate refinement relations according to how the tolerant system relates to abstract properties of the intolerant one in terms of safety and liveness. The specification language utilized is object-oriented and based upon first-order predicates on communication traces. Fault-tolerance refinement relations are formalized within this framework.

Keywords
Fault-tolerance, formal methods, refinement, trace semantics, incremental design, object orientation.

1 Introduction

Fault-tolerant systems are constructed to improve system dependability by giving the system an ability to operate in the presence of (some) faults. Faults can be physical, or related to design or interaction [7]. At an abstract level, abnormal behavior in a part of a computing system is naturally represented by exceptions [2], leaving the actual error detection to the implementation. The verification of fault-tolerant systems usually consists of recognizing possible failures prior to the design of a system and then prove that the anticipated faults and their treatment preserve desired system properties, as in e.g. [6], [9], [10]. While such an approach presupposes a detailed understanding of the system studied, the design process of a system (in a specification language) is often more gradual, using refinement techniques in a stepwise development of specifications, including new aspects of the evolving system in each step.

Recently [3] proposes a component-based methodology for a gradual introduction of fault-tolerance. An intolerant system is extended with layers that detect and handle exceptional behavior by adding new components, thus incrementally enhancing system dependability for each layer. The major advantage of this approach to the design of fault-tolerant systems is that we need not be aware of all possible failures a priori, but rather concentrate on the functionality of the fault-free system in the beginning of the design process. Additional insight is often gained from work with a particular system and fault-tolerance can, with advantage, be included in the design at a later stage.

In this paper we consider a stepwise introduction of fault-tolerance in system design at the requirement specification level. The approach is formalized within a specification language designed to develop distributed object-oriented systems, enabling black-box specifications of the observable behavior of objects or components, by first-order predicates on finite traces of asynchronous communication events.

The approach in this paper is illustrated through a case study where dispenser breakdown is introduced into a system of teller machines. Fault occurrence and system recovery are shown to maintain an abstract system requirement and refine the fault-ignorant specification. The example could be extended with additional faults such as line failure, giving rise to a similar development to the one studied. An extended version of this paper is available [5].

2 Fault-Tolerant Design

We use an incremental specification method to design multi-tolerant distributed systems. Multi-tolerance refers to

*Original appeared in the proceedings of the 2nd Asian Pacific Conference on Quality Software (APAQS 2001), p. 223–230. Published by the IEEE Computer Society Press, December 2001.

how a system tolerates multiple classes of faults, possibly in different ways. Compared to other (typically top-down) fault-tolerance design approaches (see e.g. [6]), the complexity of correctness proofs is reduced. As specifications are extended with new fault-classes and their treatment one-by-one, we need to show that the modified design tolerates a fault class without interfering with the previous design.

A fault-class for a component is a set of (fault) actions involving the component. The fault-classes that an intolerant system is subjected to are considered in some fixed total order, $F_1, ..., F_n$, typically determined by design considerations. First, correction for the fault F_1 is introduced in a manner stated in the design requirements. The resulting design is augmented to tolerate F_2 while tolerance to F_1 is preserved. This process is repeated until all n fault-classes are accounted for. Monotonicity of the reduction of intolerance guarantees that the final system is multi-tolerant w.r.t. all the fault-classes.

At the level of specifications, this process is identified as design refinement, which gives us a stepwise design method. Identification and treatment of faults can happen anytime after the original, intolerant design, facilitating the design process without losing the advantages of a formal development. In the original method of [3] it is demonstrated that faults can be represented uniformly as state perturbations and different kinds of fault-tolerance can be defined using traditional notions of behavioral specifications, divided into safety and liveness properties. Informally, safety properties state that nothing bad may happen, whereas liveness properties claim that something good will happen [1].

Classification of fault-tolerances is based on the extent to which the object satisfies its specification in the presence of faults. Four kinds of fault-tolerance are distinguished, namely fatal, masking, nonmasking and fail-safe. *Masking tolerance* is the strictest category: in the presence of some faults, an object always satisfies its safety specification, and after an occurrence of faults it yields a trace that also satisfies both safety and liveness properties. *Nonmasking* is less strict than masking: in the presence of faults, the object need not satisfy its safety specification but, when faults stop occurring, it eventually resumes satisfying both its liveness and safety specification. *Fail-safe* is even less strict: in the presence of faults, a program will satisfy its safety specification but when the faults stop occurring, the program need not resume satisfying its liveness specification, although the safety requirements remain valid. *Fatal* is the weakest category where both safety and liveness properties are lost after the occurrence of faults.

3 Trace-based Specifications

In order to capture the main aspects of object-oriented and distributed systems, we consider objects that run in parallel and interact by (remote) method calls. Object identities may be communicated, thus allowing dynamic communication patterns. Objects are not static parts of a system, but evolve through interaction with objects in the environment. One may think of the current "state" of an object as the result of its past interactions with the environment. A trace reflects the communication history, i.e. the sequence of all interactions between the object and its environment. The observable behavior of objects up to a given time is described by predicates on finite traces. An object at one level of abstraction may represent several objects at a lower level.

In order to allow asynchronous communication, a method invocation is represented by two separate events, reflecting its initiation and its completion, respectively. If an object o_1 calls a method m in o_2 with a parameter(list) p and obtains a response, the events $o_1 \rightarrow o_2.\mathrm{m}(p)$ and $o_1 \leftarrow o_2.\mathrm{m}(p; r)$ appear in the trace, representing the initiation and termination of the call. Initiation events include values for the in-parameters only, whereas completion events also include values for the out-parameters (i.e. r in the case above).

An object o has a set of methods available to the environment. The alphabet of o consists of events reflecting invocations of the methods of o by other objects and events reflecting invocations by o of methods of other objects.

Definition 1 *The alphabet $\alpha(o)$ of an object o is defined as*

$$\alpha(o) \triangleq \{o' \rightarrow o.m(\ldots)\} \cup \{o' \leftarrow o.m(\ldots)\} \\ \cup \{o \rightarrow o'.n(\ldots)\} \cup \{o \leftarrow o'.n(\ldots)\},$$

where m is a method of o, o' ranges over other objects, and n is a method of o' available to o.

By means of interfaces and inheritance, we later consider various subsets of the alphabet of an object.

Let $\alpha_o^{\mathrm{output}}$ denote the *output events* of o, i.e. completions of calls to methods of o or initiations of calls to methods of other objects, known to o. Input events $\alpha_o^{\mathrm{input}}$ are complementary to outputs, so that $\alpha_o^{\mathrm{input}} \cup \alpha_o^{\mathrm{output}} = \alpha(o)$. We assume that an object cannot deny input events.

We let $\mathrm{Seq}[\alpha]$ denote the set of finite traces over a set of events α, and let ε, $\vdash$, $\dashv$ and $\vdash\!\dashv$ denote the empty trace, right append, left append, and concatenation, respectively. For a trace h, we let $\sharp(h)$ denote its length and h/s the restriction to events from a set s. By extension, $h/o \leftarrow$ and $h/o \rightarrow$ denote restrictions to events that initiate and terminate methods of an object o, respectively. We let $h' < h$ denote that h' is a finite prefix of h.

A *safety specification* of an object o is given by an invariant predicate P on its (finite) communication traces. A specification P defines a set of traces $\mathcal{T}_P$, which is the largest *prefix-closed* subset of

$$\{h \in \mathrm{Seq}[\alpha(o)] \mid P(h)\}.$$

If I and J are two specifications with the same alphabet, I refines J, written $I \sqsubseteq J$, if $\mathcal{T}_I \subseteq \mathcal{T}_J$.

Prefix-closed sets of traces may express safety properties, but in the presence of nondeterminism, deadlocks cannot always be distinguished [4]. These sets are generally not sufficient to express liveness properties. A simple extension to liveness is introduced below.

3.1 Semantics

The full semantics of an object o is here given by a set of finite traces describing its terminated executions, denoted $\mathcal{T}_o^*$, and a set of infinite traces describing its nonterminating executions, denoted $\mathcal{T}_o^\infty$. The set of all possible executions, $\mathcal{T}_o^\omega$, is defined as

$$\mathcal{T}_o^\omega \triangleq \mathcal{T}_o^* \cup \mathcal{T}_o^\infty.$$

The finite trace set $\mathcal{T}_o$ denoting executions of o up to a time t, can be obtained from $\mathcal{T}_o^\omega$ by the (finite) prefix-closure,

$$\mathcal{T}_o \triangleq \{h \in \text{Seq}[\alpha(o)] \mid \exists h' \in \mathcal{T}_o^\omega : h < h'\} \cup \mathcal{T}_o^*.$$

A specification I of o describes some, but not necessarily all, aspects of o. Hence I is said to be *semantically correct* if $\mathcal{T}_o \subseteq \mathcal{T}_I$ (assuming a fixed alphabet). $\mathcal{T}_I$ is defined by a predicate on traces, as above.

We say that an object is *dead(locked)* (or terminated) if it no longer produces outputs (regardless of inputs). This reflects that even a dead object may not deny its environment to give it inputs. The set $\mathcal{T}_o^*$ of finite executions represents possible deadlocks of o, either caused by o itself or by objects in its environment. However, the object may receive input after it has deadlocked, so deadlock traces need not terminate. For reasoning purposes, we isolate the deadlocks that are the responsibility of o from those that are the responsibility of the environment. Denote by $\mathcal{T}_o^{ND}$ the nonterminating executions of o that have deadlocked,

$$\mathcal{T}_o^{ND} \triangleq \{h \mapsto h' \in \mathcal{T}_o^\infty \mid h'/\alpha_o^{\text{output}} = \varepsilon\}.$$

The set of (possible) deadlocks caused by the object o itself can be defined by

$$\mathcal{D}_o \triangleq \{h \in \mathcal{T}_o^* \cup \mathcal{T}_o^{ND} \mid \#(h/o \rightarrow) = \#(h/o \leftarrow)\},$$

where the environment has completed all calls from o. A subset of the possible deadlocks can be detected from a specification I.

Definition 2 *If there is no extension of a trace h with output events in a specification I, then h is an* absolute deadlock *in $\mathcal{T}_I$:*

$$\mathcal{D}_I \triangleq \{h \in \mathcal{T}_I \mid \forall h' : h \mapsto h' \in \mathcal{T}_I \Longrightarrow h'/\alpha_o^{\text{output}} = \varepsilon\}$$

Definition 3 *A specification I of an object o is* deadlock-deterministic *if all deadlocks of o are absolute, i.e.*

$$\mathcal{D}_o \subseteq \mathcal{D}_I.$$

A deadlock-deterministic object o is predictable with respect to deadlocks, in the sense that a deadlock may not depend on internal nondeterminism of o. Thus, if two executions of the object give the same trace, the object may not cause deadlock in only one of the executions. This property is often desirable. For deadlock-deterministic objects, it is possible to reason about liveness properties from safety specifications. In particular if $\mathcal{D}_I$ is empty, the object may not cause a deadlock. However, for objects that are not deadlock-deterministic, one may not reason about liveness in this manner. For the purpose and examples of this paper, this notion of liveness suffices, and we avoid more general notation for describing liveness.

4 Fault-Tolerant Design with Traces

Incremental reasoning around faults manifests itself at the code level through the use of wrappers, thus allowing code reuse. In specification languages incremental reasoning is typically done through refinement. Refinement relations formalize the addition of detail as a specification evolves. Fault-tolerant refinement departs from an ideal, fault-free specification and incrementally adds robustness.

In a trace model, both hardware and software faults can be represented by replacing standard events by exceptions. A fault-class within this framework is a set of events identifying a fault, i.e. typically an event ranging over parameter values. Possible fault occurrence results in a nondeterministic choice of completion event corresponding to an initiation in the history. Traditional refinement relations, i.e. subset-based refinement on trace sets, do not cover this kind of development, as this nondeterminism augments trace sets. Now, a known input may generate unknown output and the refinement relation must reflect this. Consider the evolution of a design through refinement, starting with a specification I_1 (Figure 1). Our approach to fault-tolerance is related to an abstract specification or invariant P_A of the program.

The specification I_1 is a possible solution to a requirement P_A by the regular refinement relation $\sqsubseteq$, and possibly to other requirements as well. We study the further evolution of this proposed design with respect to the abstract requirement. By assumption, I_1 does not consider faults. We expand I_1 with an ability to handle some fault f. The expanded specification I_2 is then f-tolerant (whereas I_1 is f-intolerant) with respect to the abstract specification I_A. Generally, if a specification I_i is f-intolerant for some fault f, i.e. f breaks the invariant of I_i, and I_{i+1} is f-tolerant, the introduction of f-tolerance may or may not affect I_{i+1}

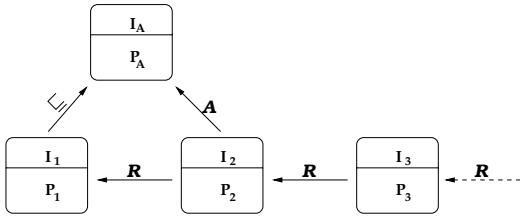

Figure 1. When faults are considered, the specification is refined with respect to abstract requirements.

as a refinement of the abstract specification I_A. Our aim is to study the possible relationship between these specifications, represented by the three relations $\sqsubseteq$, $\mathcal{A}$ and $\mathcal{R}$.

The relation $\mathcal{A}$ depends on the kind of fault-tolerance we want, i.e. $\mathcal{A}$ describes how the treatment of the fault affects the relationship between I_{i+1} and I_A. $\mathcal{R}$ is a specification development relation, which tells us how the introduction and treatment of the fault in I_{i+1} modifies the intolerant specification I_i. This relation will also necessarily vary according to the fault-tolerance category required for the fault considered in the refinement step.

4.1 Different Kinds of Fault-Tolerance

Different kinds of fault-tolerance are now interpreted in the setting of trace sets. These depend on the inheritance of both safety and liveness properties in the design process (Figure 2). There is a liveness notion in the deadlock determinism of specifications, so inheritance of liveness properties is through conservation of this property. Trace specifications are black-box, so a fault is represented by the irregular completion of a method call, i.e. by an exceptional completion event in the history. Thus, fault-detection at the current level of abstraction is explicit in the language.

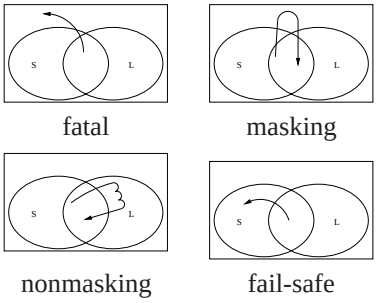

Figure 2. Fault-tolerance as conservation of safety and liveness properties.

Any trace of I_{i+1} is expected to behave like a trace of I_i until a fault occurs, the behavior changes only after occurrence of the (first) fault. To express this, the longest left subtrace restricted to the alphabet of the intolerant specification is used.

Definition 4 *Denote by $lmp(h, \alpha)$ the leftmost maximal prefix of a trace h restricted to an alphabet α.*

The ability to predict, to a certain extent, the future lies implicit in the idea of recovery after the occurrence of a fault. The possible futures of a trace are defined.

Definition 5 *Let $\mathcal{T}$ be a trace set and let h be a trace in the set, $h \in \mathcal{T}$. The extensions to h in $\mathcal{T}$ are defined as*

$$\text{extensions}(h, \mathcal{T}) \triangleq \{t \mid h \mathbin{\vdash} t \in \mathcal{T}\}.$$

From this definition an equivalence notion for traces (with respect to a trace set $\mathcal{T}$) is obtained, which corresponds to fusion closure as discussed in [3]. Recovery after fault-occurrence implies that a normal situation is regained in the future. The possible fault-free extensions to the trace should then be comparable to those of some trace in the intolerant specification. The combination of the above definitions yield a restricted form of extension.

Definition 6 *Let α and α' be alphabets such that $\alpha \subseteq \alpha'$, $\mathcal{T} \subseteq Seq[\alpha']$, and let h be a trace over α'. Define the α-extensions of h in $\mathcal{T}$ as*

$$\alpha\text{-extensions}(h, \mathcal{T}) \triangleq \{t \in Seq[\alpha] \mid h \mathbin{\vdash} t \in \mathcal{T}\}.$$

Fault-tolerance relates to an abstract specification I_A. We will assume that the specifications I_A, I_1 and I_2 of Figure 1 describe the same given object. The specification I_2 is said to be fatal, masking, nonmasking or fail-safe fault-tolerant for I_1 with respect to I_A. The different kinds of fault-tolerance are defined by the relation $\mathcal{A}$ between the tolerant design I_2 and the abstract specification I_A. In what follows, the function $f : \text{Seq}[\alpha(I_2)] \to \text{Seq}[\alpha(I_1)]$ computes the intolerant equivalent of tolerant traces h after "repair", so $\alpha(I_1)$-extensions$(h, \mathcal{T}_{I_2}) \subseteq$ extensions$(f(h), \mathcal{T}_{I_1})$. If I_A is deadlock-deterministic with invariant P_A, then I_2 is

1. Fatal fault-tolerant when
   $\forall h \in \mathcal{T}_{I_2} \colon P_A(lmp(h, \alpha(I_1))/\alpha(I_A))$,
2. Masking fault-tolerant when I_2 is deadlock-deterministic and $\forall h \in \mathcal{T}_{I_2} \colon P_A(f(h)/\alpha(I_A))$,
3. Nonmasking fault-tolerant when I_2 is deadlock-deterministic and

$$\begin{aligned} &\forall h \in \mathcal{T}_{I_2} \colon P_A(lmp(h, \alpha(I_1))/\alpha(I_A)) \\ &\wedge (\neg P_A(h/\alpha(I_A)) \\ &\quad \Longrightarrow \forall h' \in \mathcal{T}^{\omega}_{I_2}\ h < h' \Rightarrow \exists h_r \in \mathcal{T}_{I_2} \colon \\ &\qquad h < h_r < h' \wedge P_A(f(h_r)/\alpha(I_A)), \end{aligned}$$

   where h_r denotes the trace at recovery,

4. Fail-safe fault-tolerant when
   $\forall h \in \mathcal{T}_{I_2} : P_A(f(h)/\alpha(I_A))$.

When it is convenient to distinguish these relations, we call them $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ and $\mathcal{A}_4$, respectively. The relation $\mathcal{R}$ of Figure 1 expresses that the transition from a specification to the next has been done correctly, relative to the desired category of fault-tolerance. When we examine this relation in the light of $\mathcal{A}$, it becomes clear that $\mathcal{R}$ can be described in terms of the given function f and the (possible) conservation of deadlock-determinism. In fact, $\mathcal{A}_i = \mathcal{R}_i \circ \sqsubseteq$. If $\alpha(I) \subseteq \alpha(J)$ for specifications I and J, then refinement is after projection. In fact, $I \sqsubseteq J$ is defined by $\forall h \in \mathcal{T}_I \Rightarrow h/\alpha(J) \in \mathcal{T}_J$, corresponding to fail-safe without "repair".

5 Case Study: Design of a Bank System

A bank system is specified, with three kinds of objects: users, teller machines and a center. 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.

Our approach leaves control with the teller machines. The user and the center are regarded as passive objects, solely reacting to method calls initiated by the teller machine object, which is in the middle of the communication link. Consequently, all methods are declared in the user and center objects, and calls are made from the teller machine to one of these. The teller machine has no visible methods.

5.1 A Specification Language

A simple syntax for trace specifications is introduced through interfaces with behavioral constraints, using a subset of OUN [8]. Objects support interfaces, which have the general form

interface name (typed parameterlist)
  inherits <list of superinterfaces>
begin
  with cointerface
    list of visible operations
    deadlock-deterministic
    invariant
  where-clause
end

The syntax $o{:}\ I$ denotes that the object o supports the interface I. Any object o' supporting the cointerface may use the operations of I, and o may use methods of o' specified in the cointerface. The alphabet thus defined is referred to by $\alpha(o{:}\ I)$. The typed parameter list defines the initial objects and values which an object supporting the interface depends on, for instance providing permanent links to external objects. Multiple inheritance is expressed by giving a list of several superinterfaces. Interface parameters, local operations, cointerfaces, and invariants are inherited by subinterfaces when applying relevant projections. Multiple inheritance is formed by union of operations and concatenation of the interface parameter lists. An operation has the syntax **opr** name (<typed in-parameter list> **out** <typed out-parameter list>). If there are no out-parameters, the keyword **out** is omitted. The **where**-clause is used to define auxiliary functions.

An invariant is used to state requirements on the communication history of o, restricted to the alphabet of the current interface, defining a trace set as in Section 3. The optional keyword **deadlock-deterministic** states that the trace set of the interface is deadlock-deterministic.

5.2 Syntax of the Intolerant Interfaces

We specify one interface for each role of the teller machine (Figure 3) and use the multiple inheritance property

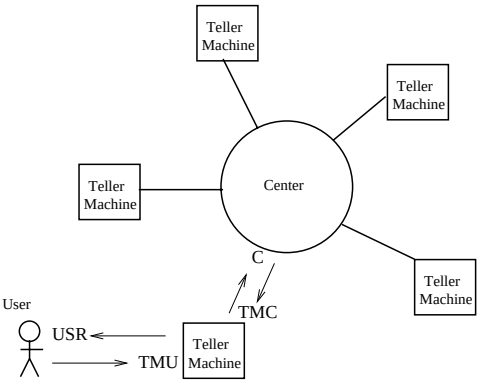

Figure 3. The bank system, illustrated by objects and interfaces (upper case).

of subinterfacing to obtain a (more complete) specification of the teller machine from the different role descriptions. We will first describe the methods and later consider the behavior of the system. The interaction between the teller machine and the user consists of these operations:

- insert: The card is inserted into the teller machine.
- giveCode: The user attempts to validate the card.
- query: The user chooses activity. There are two possibilities:
  "wd" (withdraw) and "end" (end session).
- withdraw: The teller machine is informed of the amount desired by the user.

- dispense: The action of dispensing a sum to the user.
- return: The card is returned to the user.

The interface of the user can be specified as

interface USR
begin
  with TMU
    opr insert(**out** $card$: nat)
    opr giveCode(**out** $code$: nat)
    opr withdraw(**out** sum : nat)
    opr query(**out** $choice$: nat)
    opr dispense(sum : nat)
    opr return()
end

When invariants are absent, we think of them as "true". The teller machine has no visible methods, but an object u with interface USR is included as a formal parameter. The interface of the teller machine towards users becomes

interface TMU (u:USR)
begin
end

This method-less interface gives us a nonempty alphabet because of the formal parameter. For a teller machine x, the interface has an alphabet with events $\{x \leftarrow u.m(\ldots)\} \cup \{x \rightarrow u.m(\ldots)\}$, where m is a method declared in USR. Hence, we can specify requirements on the teller machine.

Interaction between the teller machine and the center consists of these operations:

- authorize: Check that a code and a card correspond.
- debit: If the amount can be withdrawn from an account, do so and return true. Otherwise, return false.
- credit: Augment an account with a given amount.

The methods are placed in the interface of the center, leaving initiative with the teller machine.

interface C
begin
  with TMC
    opr authorize($card$:nat, $code$:nat **out** ok:bool)
    opr debit(sum:nat, $card$:nat, $code$:nat **out** ok:bool)
    opr credit(sum:nat, $card$:nat **out** ok:bool)
end

The interface of the teller machine towards the centre is also without declared methods.

interface TMC (c:C)
begin
end

5.3 The Abstract System

The abstract property against which we will compare our specifications, can informally be described by:

"Neither bank nor user loses money due to failures."

The requirement, when formalized, says that transactions decrease the amount on a user's account with exactly the sum dispensed to the user. However, this is not the case at all times during a transaction, but only after transactions.

The amount received by a user u from the teller machine m is calculated by a function $\mathcal{R}$ on the history. It is sufficient to consider the completion events of insert and dispense. Ignoring other events, a sequence of dispense actions is preceded by an insert action that identifies u, about to receive the dispensed sum. Similarly, the amount withdrawn from the account is calculated by a function $\mathcal{W}$. All changes in the account balance can be identified through the completion events of the debit and credit methods. Cases are considered in the order listed (à la ML). The notation "others" denotes any event. The current object m is always implicit in the events of the example. We therefore write "$\leftarrow u.$insert(_)" instead of "$m \leftarrow u.$insert(_)". Finally, the invariant must hold when the card has been returned to u, i.e. when the history h ends with (the initiation of) the method insert. Let **ew** denote "ends with".

interface TMA(c:C, u:USR)
  inherits TMC(c), TMU(u)
begin
  deadlock-deterministic
  inv $\forall x{:}\, h$ **ew** $\rightarrow u.\text{insert}() \Rightarrow$
    $\mathcal{W}(x, h/\alpha(c{:}\, C)) = \mathcal{R}(x, h/\alpha(u{:}\, USR))$
  where
    $\mathcal{R}$: nat $\times$ Seq$[\alpha(c{:}\, C)] \rightarrow$ nat
    $\mathcal{W}$: nat $\times$ Seq$[\alpha(u{:}\, USR)] \rightarrow$ nat

    $\mathcal{R}(x, \varepsilon) = 0$
    $\mathcal{R}(x, \leftarrow u.\text{insert}(x) \dashv \leftarrow u.\text{dispense}(y) \dashv h)$
      $= \mathcal{R}(x, \leftarrow u.\text{insert}(x) \dashv h) + y$
    $\mathcal{R}(x, \leftarrow u.\text{insert}(\_) \dashv \leftarrow u.\text{insert}(x) \dashv h)$
      $= \mathcal{R}(x, \leftarrow u.\text{insert}(x) \dashv h)$
    $\mathcal{R}(x, \leftarrow u.\text{insert}(\_) \dashv \text{others} \dashv h)$
      $= \mathcal{R}(x, \leftarrow u.\text{insert}(x) \dashv h)$
    $\mathcal{R}(x, \text{others} \dashv h) = \mathcal{R}(x, h).$

    $\mathcal{W}(x, \varepsilon) = 0$
    $\mathcal{W}(x, \leftarrow c.\text{debit}(y, x, \_; true) \dashv h) = \mathcal{W}(x, h) + y$
    $\mathcal{W}(x, \leftarrow c.\text{credit}(y, x; true) \dashv h) = \mathcal{W}(x, h) - y$
    $\mathcal{W}(x, \text{others} \dashv h) = \mathcal{W}(x, h).$
end

5.4 Specification of the Teller Machine

The desired behavior of the teller machine emerges from the interleaving of events from its two interfaces, so for brevity we look at the composition directly, through a new interface TM. At the fault-free level credit operations are not used. Some interaction scenarios are identified:

1. The user gives an incorrect code for the card, so authorization fails. The card is returned to the user and the session terminates.
2. The user gives the correct code to the teller machine. The session continues by a query-driven menu where the user requests an amount, this request is accepted by the center and the amount is dispensed. At the end of the session, the card is returned.

These scenarios are formalized in the interface TM. By inheritance, we reuse the methods previously declared.

interface TM(c:C, u:USR)
  inherits TMC(c), TMU(u)
begin
  deadlock-deterministic
  inv h **prp** (start(x, y) (withdrawSession(x, y) quit
                |badCode(x, y)) • x, y : nat)*
  where
    start (x, y) =↔u.insert(x) ↔u.giveCode(y)
    okCode(x, y) =↔c.authorize$(x, y; true)$
    badCode (x, y) =↔c.authorize$(x, y; false)$
        ↔u.return()
    okWithdraw(x, y) =↔u.query(; "wd")
        badAmount$(x, y)^*$ Dispense(x, y)
    Dispense(x, y) = okAmount(x, y, sum)
        ↔u.dispense(sum) • sum : nat
    okAmount(x, y, sum) =↔u.withdraw(sum)
        ↔c.debit$(sum, x, y; true)$
    badAmount(x, y) =↔u.withdraw(sum)
        ↔c.debit$(sum, x, y; false)$
    withdrawSession(x, y) = okCode(x, y)
        okWithdraw$(x, y)^*$
    quit =↔u.query(; "end") ↔u.return()
end

The first-order predicate h **prp** P states that the trace h is a prefix of a pattern P. Patterns are regular expressions extended with a binding operator • and pattern matching. A trace is a prefix of a pattern if it has an extension described by the pattern. Parameters in events may be left unspecified by a wildcard "_". Auxiliary functions are used in the predicates for readability, defined by equations.

The symbol ↔ in a pattern represents an initiation succeeded by its completion. Here, all events are on this form, thus ↔u.insert(_) reflects the initiation of a call to the method insert of object u by the current object, followed by its completion. In the parameters to an event, those that succeed the semicolon are output, occurring only in completions.

We need to verify that TM refines the abstract interface TMA of the teller machine. The argument uses induction over the traces of $m{:}\,TM$, and is omitted here.

5.5 Introducing Tolerance

Consider failures in the dispenser mechanism of the teller machine and assume that occurrences of such failures (DF) can be detected. Some remainder r of the sum demanded has not been dispensed and the teller machine should enter a routine for correcting the effects of the fault. The teller machine kicks out the card, blocks the card reader, sends a call to the center to return r to the user's account, and hibernates until repair. After repair, it unblocks the card reader and waits for a new card. To fulfill this requirement, we need to consider the interface TM of the teller machine, as the partial descriptions TMU and TMC lack the alphabet needed to describe the interleaving of events from the interaction scenarios. Actual repair may be represented by an additional object and corresponding interfaces, see [5].

The specification of USR is expanded with an exception for dispenser failure through method overloading in a slightly modified interface:

interface USR-DF
inherits USR
begin
  with TMU-DF
    opr dispense(sum : nat) **throws** dispense$_{DF}$
end

This exception manifests itself in the alphabet by an additional completion event to the method dispense, which represents error detection: $m \leftarrow u.\text{dispense}_{DF}(sum)$. In the example we can then express the necessary actions to regain the abstract requirement of TMA after the occurrence of faults. Exceptions can be extended with debugging information, without change in the reasoning.

The interleaving of the DF-tolerant interfaces is obtained using multiple inheritance, thus allowing reuse of old specifications, including auxiliary functions.

interface TM-DF(u:USR-DF), TMC(c:C)
  inherits TMU-DF(u), TMC(c)
begin
  deadlock-deterministic
  inv h **prp** (start(x, y) [withdrawSession(x, y)
    (badWithdraw(x, y) | quit)|badCode(x, y)] • x, y: nat)*
  where
    Dispense$_{DF}(x, y, rem)$ =↔u.withdraw(s)
        ↔u.dispense$_{DF}(s')$ • $s' < s \wedge rem = s - s'$

badWithdraw(x, y) $=\leftrightarrow u.$query(; "wd")
badAmount$(x, y)^*$ Dispense$_{DF}(x, y, sum)$
$\leftrightarrow c.$credit$(x, sum; true)$ $\leftrightarrow u.$return() $\bullet\ sum$: nat
end

The larger alphabet of this interface 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.

We want to show that the proposed interface TM-DF is correct with respect to the intolerant one originally specified. The treatment of dispenser failure is in the masking/nonmasking category of fault-tolerance, so TM-DF is a masking/nonmasking fault-tolerant refinement of TM. The extension h_r of the trace $h \vdash$ Dispense$_{DF}(x, y, r)$ where the dispenser failure has just occurred, can be identified:

$$h_r = h \vdash \text{Dispense}_{DF}(x, y, r) \vdash m{\leftrightarrow}u.\text{return}() \vdash m{\leftrightarrow}c.\text{credit}(x, r; true).$$

After the fault has been repaired, we can pretend that the user always wanted to withdraw $s - r$ instead of s, so

$$f(h_e) = h \vdash {\leftrightarrow}u.\text{withdraw}(s - r) \vdash {\leftrightarrow}u.\text{dispense}(s - r).$$

So far we have not discussed the relationship between the FT specification and the abstract requirement. Obviously, we expect the abstraction to be by either the masking relation $\mathcal{A}_2$ or the unmasking relation $\mathcal{A}_3$. By construction the fault-free behavior of TM-DF is identical to that of TM, so we need to investigate whether an erroneous session can break the abstract requirement, i.e. the invariant of TMA. Such an erroneous session will be on the form

$$\ldots m{\leftarrow}u.\text{insert}(x)\ m{\leftarrow}c.\text{debit}(s, x, \_, true) \quad m{\leftarrow}u.\text{dispense}_{DF}(s')\ m{\leftarrow}c.\text{credit}(s - s', x; true)\ldots$$

when all irrelevant events are hidden. We need to include the dispenser failure event in the definition of the function calculating the total amount dispensed. Ignoring some detail, we extend the definition of $\mathcal{R}$ with the new event,

$$\mathcal{R}(x, m{\leftarrow}u.\text{insert}(x) \dashv m{\leftarrow}u.\text{dispense}_{DF}(y) \dashv h) = \mathcal{R}(x, m{\leftarrow}u.\text{insert}(x) \dashv h) + y.$$

Now, for traces ending with (the initiation of) insert, an erroneous session will not break the abstract requirement.

Observe that the difference between masking and nonmasking fault-tolerance is a question of how the invariant of TMA is formulated. For this example, were we to demand that the invariant of TMA held when h **ew** $\leftarrow u.$return(), we would get nonmasking fault-tolerance instead.

The case study can be extended with other faults. For instance, line failure between the teller machines and the center gives rise to a new completion event to the methods of the center. When the development is incremental, it is necessary to be aware of every possible completion of method calls utilized in the treatment of the new fault.

Finally, TM-DF is deadlock-deterministic, as a deadlock would be caused by noncompletion of a method call to another object. Introducing time-out completion events into the formalism would remove this requirement on deadlocks and deadlock in the environment could be handled locally.

6 Conclusion

A stepwise development of specifications is suggested where an intolerant specification is refined to include tolerance for faults. The object-oriented formalism allows inheritance and specification reuse. First, the fault-tolerance notions of [3] are reinterpreted within a framework of object-oriented specifications and trace semantics. Different kinds of fault-tolerance are distinguished by the preservation of safety and liveness properties.

Simplicity of the specification formalism has been emphasized. Safety specifications based on finite traces are employed, augmented in a simple way to obtain liveness reasoning for an interesting class of objects.

We need to distinguish between the preservation of an abstract system property and the actual refinement of a specification, due to the different levels of detail encountered. The intolerant specification is presumed to implement the abstract requirement, so the tolerant specification relates to both the abstract and the intolerant specifications.

In the stepwise development of specifications, iteration of the design procedure requires that previously specified faults are included in reasoning about repair sequences for later specifications. The exact refinement relation we use in each step depends upon which fault-tolerance category we want for the fault we consider, i.e. on how abstract properties of the intolerant specification should be preserved by the tolerant specification. Different abstraction and refinement relations are given precise definitions within the formalism. With this distinction between abstraction and refinement, we gain clarity with regard to an incremental introduction of fault-tolerance in the design of critical systems. A platform using the PVS theorem prover [9] to verify refinement steps is currently being developed [11].

A case study is developed to illustrate how such an incremental introduction of fault-tolerance can be done departing from a fault-free specification of a system of teller machines. The case study also illustrates how time-outs, which can be handled with a mechanism similar to the one studied in this paper, provide a natural extension to the work done here, strengthening the notion of liveness.

References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[2] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 1981.

[3] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan. 1998.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[5] E. B. Johnsen, O. Owe, E. Munthe-Kaas, and J. Vain. Incremental design of dependable systems in OUN. Technical Report 293, Dept. of informatics, University of Oslo, 2000.

[6] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *Proceedings of FTRTFT'94*, volume 863 of *Lecture Notes in Computer Science*. Springer, 1994.

[7] J. C. Laprie. Dependability of computer systems: from concepts to limits. In *IFIP international workshop on dependable computing and its applications*, South Africa, 1998.

[8] O. Owe and I. Ryl. A notation for combining formal reasoning, object orientation and openness. Technical Report 278, Dept. of informatics, University of Oslo, 1999.

[9] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[10] J. Peleska. Design and verification of fault tolerant systems with CSP. *Distributed Computing*, 5:95–106, 1991.

[11] I. Traore, D. B. Aredo, and K. Stølen. Formal development of open distributed systems: Towards an integrated framework. In *Proceedings of Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours (OOSDS'99)*, France, 1999.