University of Oslo
Department of Informatics

Incremental Design of Dependable Systems in OUN

Einar B. Johnsen,
Olaf Owe,
Ellen Munthe-Kaas,
Jüri Vain

Research report 293
ISBN 82-7368-242-0

December 2000

Incremental Design of Dependable Systems in OUN

Einar Broch Johnsen* Olaf Owe*
Ellen Munthe-Kaas* Jüri Vain†

*Department of informatics, University of Oslo, Norway,
email: {einarj, olaf, ellenmk}@ifi.uio.no
†Institute of Cybernetics, Tallinn Technical University, Estonia, email: vain@ioc.ee

Abstract

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

Different types of faults are identified and given separate refinement relations depending on how the tolerant system relates to abstract properties of the intolerant one in terms of safety and liveness. The specification language OUN utilized is based upon communication sequences (traces) between objects and first-order predicates on these traces. Criteria are defined for reasoning about liveness with sets of finite traces. Fault-tolerance refinement relations are precisely defined within this framework.

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 [10]. At an abstract level, abnormal behavior in a part of a computing system is naturally represented by exceptions [2], thus 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 proving that the anticipated faults and their treatment preserve desired system properties, as in a.o. [9], [14], [15]. While such an approach presupposes a detailed understanding of the system studied, the design process of a system (in a specification language) will often have a more dynamic character, utilizing refinement techniques to open for a stepwise development of specifications, gradually including new aspects of the future system into the design.

Recently [3] has proposed a component based methodology for a gradual introduction of fault-tolerance. The idea is to wrap an intolerant system in

layers that detect and handle exceptional behavior by adding new components, thus incrementally enhancing system performance for each layer. The major advantage of this approach for 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 at a later stage in the design.

In this paper we consider a stepwise introduction of fault-tolerance in system design within the OUN specification language. The OUN language is designed to enable development of object oriented, distributed and open systems, allowing formal specification and reasoning control. The specification language deals with interfaces, describing various aspects of objects, and contracts, for specifying subsystems. The design language includes an imperative class construct as well, but in this paper we concentrate on specifications. The specification language is based on finite trace semantics (known from a.o. CSP [7]) and the following discussion of fault-tolerance issues will be related to this framework. The OUN specification language can be seen as an extension of and a complement to UML and system descriptions in UML can in fact be translated into the language in a semi-automatic manner [18] by the addition of behavioral predicates. In the literature, notions of fault-tolerant development are not treated in detail for specification languages like UML [4] or OCL [19], although the importance of a formalization of these issues are discussed in [6]. With our framework, it is possible to follow a development strategy departing from UML and gradually introducing a formal framework for fault-tolerant development, as proposed by [6].

Within OUN, the first development steps of a fault-tolerant system is studied, with an emphasis on understanding fault-treatment as refinement relations on the sequences of interactions between objects. The relations will vary according to how the fault-tolerance preserves the safety and liveness specification of the original design. The development of fault-tolerant specifications is done in a stepwise manner. Faults are gradually introduced into the specifications and the correctness of a system development process can be verified through precise relations between the intolerant and the tolerant descriptions.

Some work has been done on transforming an intolerant specification into a tolerant one. The use of a traditional refinement relation for this transformation gives rise to complications. A traditional refinement relation, in the sense of implication (or subset for trace-based formalisms) restricts the set of states (or traces) whereas fault-tolerance typically expands such a set. A solution has been to design a system where faults are recognized but not dealt with, and then refine this system to obtain fault-tolerance with regard to an abstract specification [11]. Another solution is to weaken the requirement of the intolerant specification so that faulty behavior is accepted by the modified requirement through an ad hoc failure hypothesis, as in the trace-based approach of [16].

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.

2 Fault-Tolerant Design

We employ an incremental method for designing multitolerant distributed systems, following [3]. Multitolerance refers to the ability of a system to tolerate multiple classes of faults, each in a possibly different way. Typically, the objective of fault-tolerance design in the context of safety critical systems is to localize the effects of faults in such a way that the overall system performance is not affected by minor component failures and that the system reacts safely in the event of a major component failure. Identification of failure mechanisms and systematic design of their toleration mechanisms forms the key to the FT design method.

The component based method supports incremental design adding fault tolerance to an intolerant system in a stepwise fashion. Compared to other (typically top-down) fault-tolerance design approaches (see e.g. [9]) it reduces complexity of the design correctness proofs arising due to potential interferences between multiple types of faults and tolerating their components. Extending design specifications with new fault classes and designing components tolerating them one-by-one, it remains to show that a new component introduced in a design step tolerates all faults of the class and it does not interfere with earlier design.

As a result, the method of [3] provides a design methodology consisting of an intolerant system and a set of additional components (correctors), one for each desired type of tolerance. 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, $F_1, ..., F_n$. The order (priority) of faults is typically determined by design considerations. First, a corrector component is added to the intolerant system so that it tolerates F_1 in a manner stated in the design requirements. The resulting system is then augmented with another component so that it tolerates F_2 and its tolerance to F_1 is preserved. The process of adding a new tolerance and preserving all old tolerances is repeated until all n fault-classes are accounted for. Monotonicity of the reduction of intolerance guarantees that the final system is multitolerant w.r.t. all the fault classes $F_1, \ldots, F_n$.

In our approach, we work with specifications and not with code. Instead of building correctional wrappers around intolerant code with components that identify and handle faults, it seems natural to identify the fault-classes as refinement relations between specifications. Instead of adding new components, it is sufficient to refine the existing ones. This opens for 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.

2.1 Formalization of Fault-Tolerance

A formalization of FT design steps using the OUN specification language requires explicit definition of faults and fault-tolerance in trace semantics.

In the original method of [3] it is demonstrated that different types of faults (transient, permanent and intermittent faults) can be represented uniformly as state perturbations and classes of fault-tolerance can be defined using traditional notions of safety and liveness specification. Switching to the event oriented spe-

cification style of the OUN language we translate these notions, at first, into trace semantics. Traditionally, a behavioral specification is divided into a safety specification and a liveness specification. Following [1], a safety predicate states that nothing "bad" will happened, i.e. a predicate that is prefix closed, and a liveness specification states that something "good" will eventually happen. Definitions of safety and liveness are given in terms of trace sets.

Definition 1 *(Safety) A safety specification is represented by a prefix closed set of finite traces.*

Liveness is closely related to the detection of deadlocks. An object may deadlock after a trace α if there is a possibility that no further output is generated.

Definition 2 *(Liveness) A liveness specification for some object is represented by a set of finite traces s.t. for each trace α in the set it can be determined whether the object has the possibility of going into deadlock after α.*

Within finite trace semantics, the latter notion causes difficulty as deadlock need not be detectable. Therefore an ad hoc notion of "deadlock determinism" for trace sets is introduced, such that possible deadlock is detectable for deadlock deterministic trace sets. Deadlock determinism cannot be defined within finite trace semantics, so in order to semantically define the concept, we use infinite trace sets in the underlying semantics for the specifications, from which the finite trace sets are obtained by prefix closure. We may reason about liveness for deadlock deterministic trace sets.

Classification of fault-tolerances is based on the extent to which the object satisfies its specification in the presence of faults. Four types of fault-tolerance can be distinguished, namely fatal, masking, nonmasking and fail-safe. *Fatal* is the weakest type which does not guarantee preservation nor restoration of the required safety and liveness properties after the occurrence of faults. *Masking tolerance* is the strictest type of fault-tolerance: in the presence of 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, i.e., all sufficiently long suffixes of the trace satisfy the object specification. *Fail-safe* is also less strict than masking: 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.

3 The OUN Specification Language

The OUN language is designed to enable development of object oriented, distributed and open systems, allowing formal specification and reasoning control. In order to be of practical use, simplicity of specification and reasoning for large systems have been more important than mathematical expressive power. The traditional OO concepts are supported, in addition to a dynamic class concept. The main concepts of the language are interfaces, classes and contracts (for specification purposes only). The language supports multiple inheritance.

The applicative sublanguage deals with interfaces (with methods but without attributes), and contracts (for specifying subsystems), whereas the design language includes an imperative class construct (with both methods and attributes) as well. A class may implement a number of interfaces. The language is strongly typed, in particular all object variables are typed by interfaces. The language guarantees that errors such as "method not understood" may not occur.

In order to allow distribution, objects are considered to be autonomous, running in parallel and communicating asynchronously. In order to allow certain non-trivial forms of openness, such as dynamic reconfiguration, a dynamic class construct is considered, allowing classes to be redefined and extended, in order to enable added or improved functionality. A class may support a set of interfaces and this set may increase as a result of dynamic changes to the class. An object is assumed to belong to a given class, however, the functionality of the object may be improved and renewed as the class is dynamically updated. An object knows its own identity, and object identities may be communicated, thus communication patterns may change. We will here mainly be concerned with the applicative level of OUN (i.e. not classes) and focus below on the applicative sublanguage.

The language is intended to be used as a complement to UML, offering constructs for precise specification of observable behavior of objects, interfaces, contracts and classes, as well as internal specification of classes. As interfaces do not contain attributes, observable specifications are based on the concept of a communication history (trace). The communication history is the time sequence of all interactions between the object and its environment, up to a given time. The interactions are represented by method call initiations and completions.

An interface is used to specify a certain aspect of the communication capabilities of an object, describing in general both syntactic and semantic information. From a UML class diagram one may first obtain an interface with syntactic information only. One may then add semantic information in a subinterface. However, it is often convenient to define first a semantic contract, describing the interaction of two or more objects (of the same or different interfaces), which corresponds to a message diagram augmented with semantic information.

3.1 Interfaces

An interface consists of a syntactic and a semantic part, where the latter is given by an assumption, an invariant and deadlock determinism. An interface declaration has the general form

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

where the **with**-clause states that only objects supporting the specified cointerface may use the operations listed after the **with**-clause. (The type checking will ensure this.) By giving **any** as cointerface, all kinds of objects may use the operations (since **any** is the universal superinterface). Multiple inheritance is expressed by giving a list of several superinterfaces.

Type parameters may be used to make generalized definitions, by means of formal types, and the typed parameter list defined the initial objects and values which an object supporting the interface depends on, for instance providing permanent links to external objects. As the identity of any object calling the visible operations is accessible to the callee, an object may increase its knowledge about the environment.

Parameters, local operations, assumptions and invariants are inherited by subinterfaces when applying relevant projections. Multiple inheritance is formed by union of operations, allowing full overloading (treating the cointerface as a implicit parameter), and by concatenation of (each of two kinds of) the parameter lists. An operation has the syntax

opr name (<typed in-parameter list> **out** <typed out-parameter list>)

where the <typed in-parameter list> contains in-parameters, and out-parameters follow the keyword **out**. If there are no out-parameters, the keyword **out** is omitted. The parameters may be object variables (references) typed by interfaces, as well as ordinary variables (such as numbers, characters, text etc.). The **where**-clause is used to define auxiliary functions.

An assumption is used to state assumptions on the communication history between an object of the current interface (referred to by the keyword "this") and any object of the cointerface (referred to by the keyword "caller"). It has the form **asm** A(h,this,caller), where A is a first order predicate and h denotes the communication history between "this" and a "caller".

An invariant has the form **inv** P(h,this) where P is a first order predicate referring to h, which here denotes the communication history between "this" and *all* objects in the environment.

Semantically, the assumption and invariant pair defines a trace set, i.e set of possible finite traces, prefix-closed (i.e such that all prefixes of a trace in the set also are in the set). This is sufficient to describe safety properties, and also liveness properties in case the object is deterministic. For non-deterministic objects we add an ad hoc mechanism to describe certain forms of liveness. The keyword **deadlock deterministic** specifies that "this" object is such that all possible deadlocks of the object are visible in the set of finite traces. This implies that all deadlocks are absolute deadlocks (have no extension in the trace set), which often is a desirable property for non-deterministic objects. We let deadlock determinism be a part of the specification of an object, indicated by the presence or absence of the keyword **deadlock deterministic**. For deadlock deterministic objects we may then reason about liveness, for other objects, we may not (and this would require a more advanced specification language).

3.2 Contracts

A contract has the general form

contract name [type parameters](typed parameterlist)

inherits <list of supercontracts>
begin
invariant
where-clause
end

The invariant may refer to the variables in the parameterlist and their history, the local interaction history of the object variables in the parameterlist. Thus a contract serves to describe the interactions of the objects named in the parameterlist. Multiple inheritance applies, but the examples here only require single inheritance.

A typical way of developing specifications, is to start with interfaces without invariants and assumptions (these interfaces may for instance be obtained from UML class diagrams or collaboration diagrams. Semantic requirements may then be given by contracts (each involving two or more object parameters of the given interfaces). One may then refine the interfaces adding invariants and assumptions. It may then be proved that the contracts are satisfied. We use

lemma contract $C(I_1, I_2, I_3)$

to represent the obligation that a contract C holds for interfaces I_1, I_2 and I_3.

3.3 Semantics

Objects are not static parts of a system, they evolve through interaction with the environment. One may think of the current "state" of an object of a system as the result of its past interactions with the environment by way of method calls. Accordingly, the basic semantic concept of OUN is the notion of a finite communication history (trace).

A communication history is a sequence of events, which are understood as calls to and responses from methods in the objects. Two events are associated with each method declared in an interface, representing the initiation and completion of a call to that method. The events are denoted

$$\begin{array}{ll} x \rightarrow y.m(\ldots) & \text{Initiation event} \\ x \leftarrow y.m(\ldots) & \text{Termination event} \end{array}$$

where an object x calls the method m in an object y. Therefore input to an object is either initiation events of method calls to methods implemented by that object, or termination events from methods called by the object. Initiation events include values for the in-parameters only, whereas termination events also include values for the out-parameters. The symbol $\leftrightarrow$ will be used to denote either an initiation or a termination event.

In OUN objects are seen through interfaces. Consequently, for our semantic understanding of the language, we will work with the traces of interfaces, supported[1] by objects. The notation $o{:}\,I$ is used to denote that such an object o supports the interface I. If a method is syntactically declared in an interface I or in a superinterface of I, we will say that the method is of I.

[1] For simplicity, we will often refer to an interface I supported by some object o simply as an object $o{:}\,I$.

Definition 3 *Let I be an interface supported by an object o and let $I_1, \ldots, I_n$ be interfaces inherited by I. Then the alphabet $\alpha(o{:}\,I)$ is defined as*

$$\begin{aligned}\alpha(o{:}\,I) = \{o' \leftarrow o.m(\ldots)\} \cup \{o' \rightarrow o.m(\ldots)\} \\ \cup \{o \leftarrow o'.n(\ldots)\} \cup \{o \rightarrow o'.n(\ldots)\},\end{aligned}$$

where m is a method of I and n is a method of the cointerface of I or in the cointerface of a superinterface of I and o' is a variable ranging over other objects.

The cointerface is the interface declared in the **with**-clause of the specification. This interface is statically known, so the methods of the cointerface are statically available to the current interface. If the cointerface is **any**, we have no knowledge about the alphabet of the cointerface and no methods of the cointerface are available.

The parameters to the methods are assumed to be type correct, using contravariance for actual parameter matching. The language assumes that all actual method calls are type-correct, so these issues may safely be ignored here.

The alphabet thus defined can be split into two parts: the input and the output events of objects of given interfaces.

Definition 4 *Output events to an object $o{:}\,I$ are either terminations of calls to methods of I or initiations of calls to methods of other interfaces, known to I. The output events of $o{:}\,I$ is a subset of the alphabet $\alpha(o{:}\,I)$ defined as*

$$\alpha^{\text{output}}_{o:I} == \{o' \leftarrow o.m(\ldots) \in \alpha(o{:}\,I)\} \cup \{o \rightarrow o'.m(\ldots) \in \alpha(o{:}\,I)\},$$

where o' is a variable ranging over other objects.

Input events $\alpha^{\text{input}}_{o:I}$ are defined to be complementary to output events, so that $\alpha^{\text{input}}_{o:I} \cup \alpha^{\text{input}}_{o:I} = \alpha(o{:}\,I)$ for any interface I supported by an object o.

The specification control of the communication events of an object is distributed between its assumption, for inputs, and its invariant, for outputs. We denote respectively by $\mathbf{in}(h, o)$ and $\mathbf{out}(h, o)$ the longest left prefix of a trace h ending by an input and by an output event, relative to an object o. Consider an object $o{:}\,I$. Let h be a trace over some alphabet of events including the alphabet $\alpha(o{:}\,I)$ of $o{:}\,I$. The specified assumption $A(h,\text{this},\text{caller})$ is then understood as

$$A^{in}(h, o) == \forall o' \neq o \bullet A(\mathbf{in}(h/o', o), o, o')$$

and the specified invariant $P(h,\text{this})$ is understood as

$$P^{out}(h, o) == P(\mathbf{out}(h, o), o) \wedge A^{out}(h, o),$$

where A^{out} denotes the assumption on sequences[2] ending with output events. The invariant thus restricts the communication history of the object whereas the assumption states a requirement that must hold for every object communicating with $o{:}\,I$. A specification of an interface I with an assumption A and an invariant P, implemented by an object o, is semantically understood as a set of traces $\mathcal{T}_{o:I}$ that is the largest prefix closed subset of

$$\{h : \text{Seq}[\alpha(o{:}\,I)] \mid \forall h' \leq h \bullet A^{in}(h', o) \Rightarrow P^{out}(h', o)\}.$$

[2] Notation concerning sequences is explained in appendix A.

For reasoning about safety properties, this set is sufficient. However, it is often desirable to include the facility to reason about deadlock. This is done through the deadlock determinism predicate. This predicate, however, cannot be defined within finite trace semantics, so it is necessary to define an underlying semantics, from which the finite trace sets can be derived.

Let $\alpha(o{:}\,I)$ be the alphabet of an object $o{:}\,I$, including all initiation and termination events known to $o{:}\,I$. Denote by $\mathcal{T}^*_{o:I}$ the set of finite communication sequences describing terminated executions of $o{:}\,I$, $\mathcal{T}^*_{o:I} \subseteq \alpha(o{:}\,I)^*$. Denote by $\mathcal{T}^\infty_{o:I}$ the infinite traces satisfying the same predicates and thus describing the possible non-terminating executions of the object $o{:}\,I$. Safety properties are typically given as predicates on finite sequences. An infinite sequence satisfies a safety property P (continuously) if all finite prefixes of the sequence satisfy P. The set of all possible executions, $\mathcal{T}^\omega_{o:I}$, can then be defined as

$$\mathcal{T}^\omega_{o:I} == \mathcal{T}^*_{o:I} \cup \mathcal{T}^\infty_{o:I}.$$

Let $t_1 < t_2$ denote that the sequence t_1 is a finite prefix of the sequence t_2. The finite trace set $\mathcal{T}_{o:I}$ of OUN can be obtained from this set $\mathcal{T}^\omega_{o:I}$ by the (finite) prefix closure,

$$\mathcal{T}_{o:I} == \{h \mid \exists h' \in \mathcal{T}^\omega_{o:I} \bullet h < h'\} \cup \mathcal{T}^*.$$

Definition 5 *If there is no extension h' to h involving output events for the object $o{:}\,I$ such that $h \vdash h' \in \mathcal{T}_{o:I}$, we will say that h is an* absolute deadlock *in $\mathcal{T}_{o:I}$. Define*

$$\mathcal{D}_{o:I}(h) == \forall h' \bullet h \vdash h' \in \mathcal{T}_{o:I} \Longrightarrow h'/\alpha^{\text{output}}_{o:I} = \varepsilon.$$

This reflects that even a dead object may not deny its environment to give it inputs.

The set $\mathcal{T}^*_{o:I}$ represents all possible deadlocks (or terminations) caused by o or its environment. As $\mathcal{T}^*_{o:I}$ may contain deadlocks due to objects not responding in the environment, these do not represent deadlocks of o.

Definition 6 *An object is* deadlock deterministic *if all deadlocks of o are absolute, i.e.*

$$\forall h \in \mathcal{T}^*_{o:I} \bullet \mathcal{D}_{o:I}(h).$$

This notion cannot be defined within finite trace semantics, because the prefix closure hides the possible deadlocks, so we need to look at the completed executions. For deadlock deterministic objects, the finite trace set $\mathcal{T}_{o:I}$ allows reasoning about liveness as well as safety.

4 Fault-Tolerant Design using OUN

4.1 Incremental FT Specification

The idea of 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, but such reasoning around fault-tolerance has not been extensively studied. Refinement relations enable the addition of detail in layers as the specification evolves. Hence

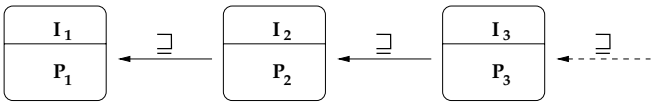

Figure 1: The evolution of the specification of an interface I_1, where the arrow represents a refinement relation $\sqsubseteq$ on interfaces.

fault-tolerant refinement should depart from an ideal, fault-free specification and incrementally add robustness to the specification.

Hardware faults can be represented in a model as regular operations occurring at random time intervals, as done in [5]. In a trace model, hardware faults as well as software faults can be represented by non-deterministically replacing regular events by exceptions. To the observer, there is no reason to further differentiate these. A fault-class within this framework is represented by the set of events identifying a fault, i.e. typically an identifier event ranging over parameter values. The specification language will therefore be sufficiently extended to handle fault-tolerance issues. Since the introduction of faults into the specifications happens through non-determinism, however, traditional refinement relations, i.e. subset-based refinement on trace sets, cannot be expected to handle this kind of development, as the non-determinism typically augments the set of traces, unless the initial specification is very loose. The treatment of faults will introduce a choice of termination event to an initiation in the fault-free specification, as we represent possible fault-occurrences through non-determinism. So a situation occurs where known input may generate unknown output and we need a refinement relation reflecting this. We consider the evolution of a specification through refinement steps, starting with an interface I_1, with a semantic specification P_1 (cf. figure 1). Our approach to fault-tolerance is typically related to an abstract specification or invariant of the program. This is illustrated by adding an abstract interface I_A to the figure (thus obtaining figure 2). The refinement of interface I_1 is relative to the semantic specification of the abstract interface, i.e. P_A.

In figure 2, the interface I_1 can be thought of as a possible solution to a requirement P_A, and possibly to other requirements as well. We wish to study the further evolution of this proposed design with respect to the abstract requirement. By assumption, I_1 is a fault-free specification. But I_1 is oversimplified because som fault f hitherto ignored, may occur and must be considered by the specification. So we expand I_1 with an ability to handle the fault f. The expanded interface I_2 is then f-tolerant (whereas I_1 is f-intolerant).

When we propose our original solution to I_A, i.e. I_1, we need not be aware of all possible faults. Our aim is to start with a simple picture and introduce the faults gradually, as proposed by [3]. Our point of departure is therefore the intolerant interface I_1. If we were to start with I_A and move directly to the (final) fault-tolerant specification, we would be in a more traditional analysis and verification setting, where all possible faults must be considered initially, as in a.o. [9], [14], [15]. By avoiding this, we allow an incremental reasoning strategy around fault-tolerance issues.

Consider three interfaces I_A, I_i and I_{i+1}, with semantic specifications P_A, P_i and P_{i+1}. The property P_i is intolerant to some fault f, i.e. f breaks the

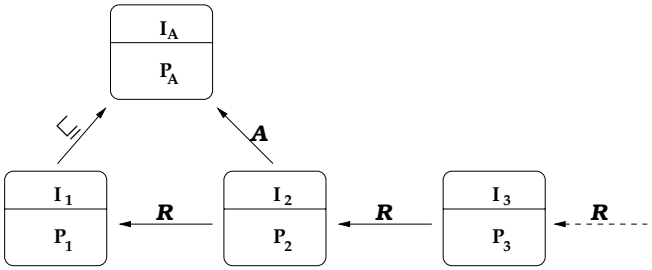

Figure 2: The evolution of the specification of an interface I_1 which considers fault-tolerance issues. The interface I_1 can be understood as the proposed solution to some abstract requirement, represented by an interface I_A such that I_1 refines I_A by a standard refinement relation, represented by $\sqsubseteq$. When we consider fault-tolerant refinement, we will relate a new specification to the abstract property of I_A by the abstraction relation $\mathcal{A}$ as well as to the previous specification by the relation $\mathcal{R}$, which is a non-standard refinement relation.

safety requirement of I_i, whereas I_{i+1} is f-tolerant so f does not break the specification P_{i+1}. Furthermore the introduction of f-tolerance may or may not affect I_{i+1} as a proposed solution to the abstract specification P_A. Our aim is to study the possible relationship between these interfaces, represented by the three relations $\sqsubseteq$, $\mathcal{A}$ and $\mathcal{R}$.

We expect $\sqsubseteq$ to be the ordinary OUN refinement relation. Furthermore, we assume that the relation $\mathcal{A}$ depends on the kind of fault-tolerance [3] we aim at, i.e. $\mathcal{A}$ describes how the treatment of the fault affects the relationship between I_{i+1} and I_A. Finally, $\mathcal{R}$ is the specification development relation, which tells us that the introduction and treatment of the fault is done correctly in I_{i+1} with respect to the intolerant specification I_i.

4.2 Types of Fault-Tolerance in OUN

The different types of fault-tolerance identified in [3] are translated into the OUN setting of (sets of) finite traces. These tolerance classes consider the inheritance of both safety and liveness properties in the software development process. In the OUN specification language we have a liveness notion in the deadlock determinism of objects, so we represent inheritance of liveness properties by the conservation of this property. Generally any trace of I_{i+1} is expected to behave like a trace of I_i until the fault occurs, the behavior changes only after the occurrence of the fault. It is assumed that all faults are represented by the irregular termination of a method call, i.e. by an exceptional termination event in the history, so fault-detection is easy within the specification language. There are practical reasons for doing this. Consider an object that produces non-deterministic output of some type, say a natural number; the only way to represent faults for this object is by a new event [17]. Hence it is necessary to completely distinguish exceptional termination from regular termination in the general case.

We introduce a function on traces that selects the leftmost maximal prefix over an alphabet α from a trace (possibly over another alphabet).

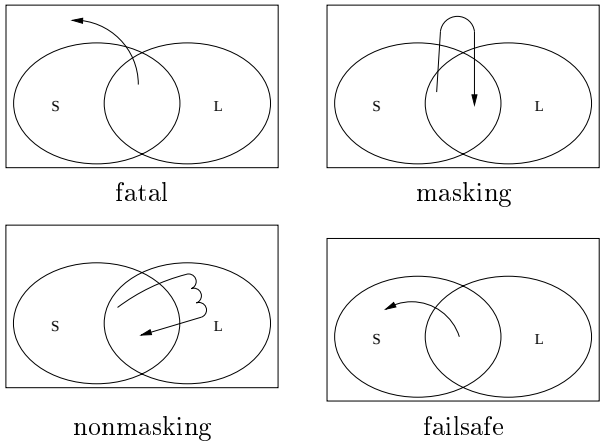

Figure 3: The different types of fault-tolerance seen as the conservation of safety and liveness properties.

Definition 7 *Let α be an alphabet and let h be a trace over another alphabet α' such that $\alpha \subseteq \alpha'$. Then the leftmost maximal prefix of h from the alphabet α is defined as*

$$lmp(\varepsilon, \alpha) == \varepsilon$$
$$lmp(e \dashv h, \alpha) == e \dashv lmp(h, \alpha) \text{ iff } e \in \alpha$$
$$lmp(e \dashv h, \alpha) == \varepsilon \text{ iff } e \notin \alpha$$

The ability to predict, to a certain extent, the future lies implicit in the idea of recovery after the occurrence of a fault. To formalize this, we define the possible futures of a trace (see for example [12]).

Definition 8 *Let $\mathcal{T}$ be a trace set and let h be a trace in the set, $h \in \mathcal{T}$. The possible extensions to h in $\mathcal{T}$ are then defined as*

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

From this definition a notion of equivalence for traces (with respect to a trace set $\mathcal{T}$) is obtained. This equivalence relation actually corresponds to the fusion closure property discussed in [3]. Now, recovery after a fault-occurrence implies that at some point in the possible extensions to the trace, a normal situation is obtained. At this point, the possible fault-free extensions to the trace should be comparable to a trace in the intolerant specification. The combination of the above definitions yield exactly what we need.

Definition 9 *Let α and α' be alphabets such that $\alpha \subseteq \alpha'$, and let h be a trace over α'. Define the α-extensions of h in a trace set $\mathcal{T}$ as*

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

The behavioral specifications of OUN objects have two properties, a safety requirement P and deadlock determinism, where the latter is optional. Further refinement of the specifications with respect to faults are divided into different types of tolerance according to how the development of a specification preserves

these two properties. The tolerance types in OUN are understood as follows (cf. figure 3):

- Fatal fault-tolerance: We lose both safety and deadlock determinism.
- Masking fault-tolerance: We keep both safety and deadlock determinism.
- Nonmasking fault-tolerance: We lose both safety and deadlock determinism for some time, but we eventually regain it, assuming that the object does not deadlock during "recovery".
- Failsafe fault-tolerance: We keep safety but lose deadlock determinism.

It is important to keep in mind that fault-tolerance relates to an abstract safety property P_A. In what follows, we will assume that the interfaces I_A, I_1 and I_2 of figure 2 are all supported by some given object. The specification I_2 is said to be fatal, masking, nonmasking or failsafe fault-tolerant for I_1 with respect to P_A. Each of these fault-tolerance types can be defined precisely by their relations to P_A (with respect to I_A), that is through various formulations of the abstraction relation $\mathcal{A}$. We assume that I_A is deadlock deterministic and behaviorally described by a safety property P_A. Then I_2 is

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

$$\forall h \in \mathcal{T}_{I_2} \bullet P_A(lmp(h, \alpha(I_1))/\alpha(I_A)) \wedge (\neg P_A(h/\alpha(I_A)) \Longrightarrow \\ \forall h' \in \mathcal{T}^{\omega}_{I_2}\ \exists h_r \in \mathcal{T}_{I_2} \bullet h < h_r < h' \wedge P_A(f(h_r)/\alpha(I_A))$$

   where $f : \mathrm{Seq}[\alpha(I_2)] \rightarrow \mathrm{Seq}[\alpha(I_1)]$ is such that $\alpha(I_1)$-extensions$(h_r, \mathcal{T}_{I_2}) \subseteq$ extensions$(f(h_r), \mathcal{T}_{I_1})$ when h_r denotes the trace at recovery and f indicates how the recovery is done.
4. Failsafe fault-tolerant if $\forall h \in \mathcal{T}_{I_2} \bullet P_A(h/\alpha(I_A))$

This describes the different possibilities for the relation $\mathcal{A}$ discussed in the previous section. When it is convenient to distinguish these, they will be referred to as $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ and $\mathcal{A}_4$ respectively.

4.3 Fault-Tolerant Refinement

Let us examine the specification development relation $\mathcal{R}$ between the intolerant and the tolerant specification. Before the first fault occurs, the two specifications give us the same histories, but faults may occur several times (in the case of "repair"), so this is not true at any point in a trace where a fault occurs. The central idea here seems to be that of repair; i.e. permitting (at least) the same actions as the intolerant specification, at some point in the future.

By assumption, faults are introduced solely be alphabet expansion, so a trace in the tolerant trace set $\mathcal{T}_{I_2}$ that *only* contains events from the intolerant alphabet is without faults, and should be expected to behave according to the intolerant specification I_1. The α-extensions predicate finds all extensions

without new faults, and these are the ones that should behave as fault-free. After a fault has been treated, we will therefore use the α-extensions of the trace for comparison with the traces in the fault-intolerant specification. The difference between masking and nonmasking fault-tolerance relates to the abstract interface and these notions are indistinguishable at this level, so there are three different relations.

1. Fatal fault-tolerance: All leftmost prefixes of h consisting solely of events from $\alpha(I_1)$ must be in the trace set $\mathcal{T}_{I_1}$. After a fault-occurrence, anything may happen.
$$\forall h \in \mathcal{T}_{I_2} \bullet lmp(h, \alpha(I_1)) \in \mathcal{T}_{I_1}.$$

2. Masking/nonmasking fault-tolerance: Let $h \in \mathcal{T}_{I_2}$. If h contains a fault, we want its extensions to return to a situation where normal behavior is possible. We will denote by h_r the extension to h that coincides with a fault-free history. To find h_r, we have to look at the complete traces of I_2 because all we know is that this will happen at some time in the future, even for non-terminating traces. The trace h' denotes a completed execution sequence.
$$\begin{array}{l} \forall h \in \mathcal{T}_{I_2} \bullet lmp(h, \alpha(I_1)) \in \mathcal{T}_{I_1} \wedge \\ \quad h \neq lmp(h, \alpha(I_1)) \Longrightarrow \\ \qquad \forall h' \in \mathcal{T}^{\omega}_{I_2} \; \exists h_r \bullet h < h_r < h' \\ \qquad\qquad \wedge \alpha(I_1)\text{-extensions}(h_r, \mathcal{T}_{I_2})) \subseteq \text{extensions}(f(h_r), \mathcal{T}_{I_1}), \end{array}$$
where $f : \text{Seq}[\alpha(I_2)] \rightarrow \text{Seq}[\alpha(I_1)]$ is such that $\alpha(I_1)\text{-extensions}(h_r, \mathcal{T}_{I_2}) \subseteq \text{extensions}(f(h_r), \mathcal{T}_{I_1})$. The function f finds the history for the fault-free specification that corresponds to h_r, and can in most examples easily be given as part of the specification. Similarly, the extension of h to h_r defines the 'repair sequence' after fault-occurrence, and can also often be specified. Hence the existential quantifier can be eliminated from the above formula in most practical cases.

3. Fail-safe fault-tolerance: It is possible to introduce new (absolute) deadlocks, while the safety properties are unchanged. We cannot express this with the prefix-closed trace sets, so we need to look at the completed traces. This suggests that while we cannot differentiate the finite trace sets of I_1 and I_2 because of the prefix closure, more deadlocks are allowed, so when we consider infinite traces, the set of completed execution sequences for I_1 is in fact only a subset of those for I_2. More formally, let $h, h_e \in \mathcal{T}_{I_1}$ and let $h' \in \mathcal{T}_{I_2}$. Then
$$\begin{array}{l} \forall h' \in \mathcal{T}_{I_2} \; \exists h_e \in \mathcal{T}_{I_1} \bullet h < h_e \wedge h < h' \Longrightarrow \\ \quad \alpha(I_1)\text{-extensions}(h', \mathcal{T}^{\omega}_{I_2}) \subseteq \text{extensions}(h_e, \mathcal{T}_{I_1}) \cup \text{extensions}(h_e, \mathcal{T}^{\omega}_{I_1}) \end{array}$$

This gives us the exact definitions for the various formulations of the relation $\mathcal{R}$, discussed in section 4.1. When it is convenient to distinguish these, they will be referred to as $\mathcal{R}_1, \mathcal{R}_2$ and $\mathcal{R}_3$ respectively.

5 Case Study: Design of a Bank 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

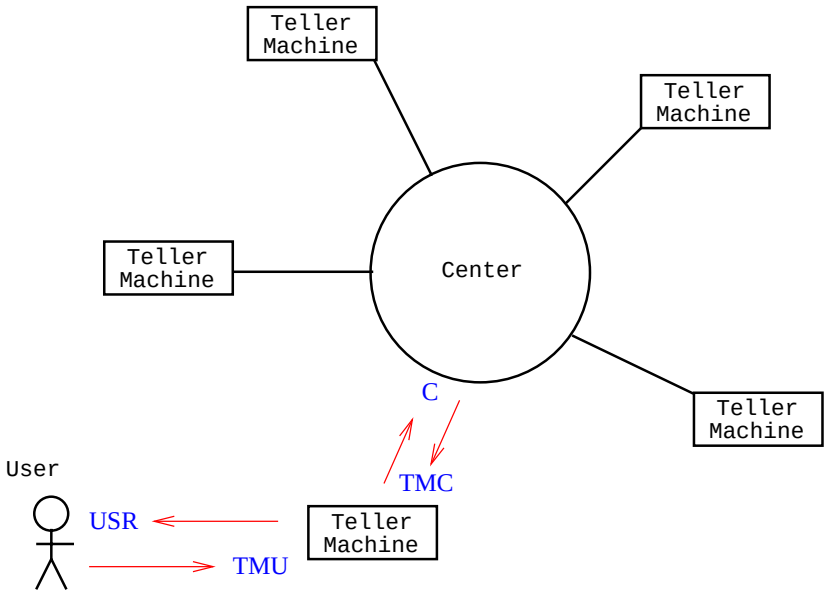

Figure 4: The teller machine system, illustrated by objects (capitalized) and interfaces (upper case). The specification leaves the teller machine in control of the transactions, regarding the user and the center as passive objects.

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 leaves 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 (figure 4) and use the multiple inheritance property of sub-interfacing to obtain a (more complete) specification of the teller machine from the partial descriptions. Class implementation is beyond the aim of this paper. A preliminary version of the case study was presented in [8].

5.1 Syntax of the Intolerant Interfaces

The interfaces are considered pairwise as they communicate with each other, see figure 4. We will first describe the syntax and later consider the behavior of the system. 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. The interface of the user becomes

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

The corresponding interface of the teller machine becomes

interface TMU (x:USR)
begin
end

The method-less interface gives us a nonempty alphabet because of the formal parameter, so it still has a function as it enables us to specify requirements. The interface declared as a formal parameter is known to the current interface, hence we can reason about the events of the former as well as the latter.

Next consider the interaction between the teller machine and the center. We will assume that this interaction consists of the following operations:

- authorize: Check that a code and a card correspond.

- debit: 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.

- credit: Augment an account with a given amount

We follow the outlined approach to the design and place all the methods in the interface of the center, enforcing the teller machine to be the active object.

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

The interface of the teller machine is empty as expected.

interface TMC (x:C)
begin
end

5.2 The Abstract System

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

> "Neither the bank nor the user can lose money due to failures."

The requirement, as it will be formalized here, says that the transactions involving a teller machine will decrease the amount on a user's account with exactly the sum received by that user. However, when we formalize this requirement, we need to accept that there may be moments during the transaction when this is simply not true. Therefore we only expect the requirement to hold at every completion of a transaction.

First we calculate the amount received by the card holder u of some account from the teller machine m. For this purpose it is sufficient to consider the termination events

- $u \leftarrow m$.insertCard(*card* : *nat*)

- $u \leftarrow m$.dispense(*sum* : *nat*)

Ignoring other events, we can assume that a sequence of dispense actions is preceded by an insertCard action that identifies the card holder about to receive the dispensed sum. Similarly, we need to calculate the amount withdrawn from the account. It is assumed that all changes in the balance of an account can be identified through the following termination events:

- $m \leftarrow c$.debit(*sum* : *nat*, *card* : *nat*, _, true)

- $m \leftarrow c$.credit(*sum* : *nat*, *card* : *nat*)

We assume that the cases are considered in the order listed (à la ML). The notation "others" is here used to denote any event. Finally, the invariant must hold every time a card is returned to the user, i.e. when the history h ends with (the initiation of) the method insertCard and x is an account.

interface TMA(u:USR, c:C)
begin
  inv $\forall x \bullet \mathcal{B}(x, h \vdash \text{this} \rightarrow u.\text{insertCard}()/\alpha(c{:}\,C))$
  $= \mathcal{R}(x, h \vdash \text{this} \rightarrow u.\text{insertCard}()/\alpha(u{:}\,USR))$
  where
    $\mathcal{R} : \text{nat} \times \text{Seq}[\alpha(c{:}\,C)] \rightarrow \text{nat}$
    $\mathcal{B} : \text{nat} \times \text{Seq}[\alpha(u{:}\,USR)] \rightarrow \text{nat}$

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

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

5.3 Semantics of the Intolerant Interfaces

The semantic requirements on the interfaces are given in the form of contracts that the interfaces should satisfy.

5.3.1 Interaction between the Teller Machine and the User

We have specified by TMU an interface of the teller machine describing its interaction with users and by USR 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. 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.

The contract[3] between the two interfaces is then specified as follows:

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

Abstract syntax is used in the predicate to make it easy to read, and the details are declared following the **where** keyword. 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 its termination. In the parameters to an event, we use a semicolon to separate input from output; everything that precedes the semicolon is input and everything that succeeds the semicolon is output.

The obligation on the interfaces is given as

lemma USR↔TMU(m: TMU, u: USR).

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

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

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.

5.3.2 Interaction between the Teller Machine and the Center

At the fault-free level we do not consider credit-operations. The communication between interfaces TMC and C is given by the following contract:

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

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. The obligation on the interfaces is then given by

lemma C↔TMC(m: TMC, c:C).

[3] In OUN, behavioral predicates are often given in the form of *patterns* of this kind. A short explanation is included in appendix B.

5.3.3 The Intolerant Teller Machine

A difficulty with partial specifications like those we have made for the teller machine, is the interleaving of events in the traces of the composition. In this case, the interfaces of the teller machine have disjoint alphabets, so we could in principle allow any interleaving of traces from the trace sets of the two interfaces in the implementation of the teller machine. This is of course undesirable, and we specify a new interface for the teller machine, inheriting the two interfaces TMU and TMC.

interface TM(u: USR, c: C)
  inherits TMU(u), TMC(c)
begin
  asm $\mathcal{H}$ **prp** (start(x, y)
                (withdrawSession(x, y)|badCode(x, y)) $\bullet$ x, y : nat)*
  inv $\mathcal{H}$ **prp** (start(x, y)
                (withdrawSession(x, y)|badCode(x, y)) $\bullet$ x, y : nat)*
  where
    start (x, y) == this$\leftrightarrow u$.insertCard(x) this$\leftrightarrow u$.giveCode(y)
    goodCode(x, y) == this$\leftrightarrow c$.authorize$(x, y; true)$
    badCode (x, y) == this$\leftrightarrow c$.authorize$(x, y; false)$ this$\leftrightarrow u$.returnCard()
    goodWithdraw(x, y) == this$\leftrightarrow u$.query(;"wd")
                                wrongAmount$(x, y)^*$ Dispense(x, y)
    Dispense(x, y) == okAmount(x, y, sum) this$\leftrightarrow u$.dispense(sum) $\bullet$ sum : nat
    okAmount(x, y, sum) == this$\leftrightarrow u$.withdraw(sum)
                                this$\leftrightarrow c$.debit$(sum, x, y; true)$
    wrongAmount(x, y) == this$\leftrightarrow u$.withdraw(sum)
                                this$\leftrightarrow c$.debit$(sum, x, y; false)$
    withdrawSession(x, y) == goodCode(x, y) goodWithdraw$(x, y)^*$ quit
    quit == this$\leftrightarrow u$.query(;"end") this$\leftrightarrow u$.returnCard()
end

It is easy to verify that this interface refines the two previous interfaces of the teller machine, as the new interface satisfies the two contracts when the traces are restricted to the relevant alphabets.

5.3.4 Correctness of the Intolerant Specification

We need to verify that the interface TM refines of the abstract interface TMA of the teller machine. Let $h \vdash m \rightarrow u$.insertCard() be a trace in the trace set of $m{:}TM$. Ignoring irrelevant events, it is a rather straightforward exercise to verify that any h for which the abstract requirement (i.e. the invariant of TMA) is expected to hold is of the form

$(m\leftrightarrow u.\text{insertCard}(\_)^*\ m\leftrightarrow u.\text{insertCard}(x)$
  $[m\leftrightarrow c.\text{debit}(sum, x, y; true)\ m\leftrightarrow u.\text{dispense}(sum)]^* \bullet x, sum : \text{nat})^*$
    $m\rightarrow u.\text{insertCard}()$,

and any such trace must also satisfy the abstract requirement. A formal proof can be given by an inductive argument over the possible traces of $m{:}TM$.

5.4 A First Level of Tolerance

Faults are simulated through nondeterminism in the behavioral description of the interfaces. In this example we consider possible failures in the dispenser mechanism of the teller machine. Assume that occurrences of dispenser failures (DF) 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")
- send credit call to the center to return the remainder to the user's account
- block card reader
- hibernate until repair
- after repair, unblock card reader and wait for new card

and the center to

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

To fulfill this requirement, it is necessary to consider the whole interface TM of the teller machine, as the partial descriptions TMU and TMC lack the alphabet necessary to describe the interleaving of events from the interaction scenarios.

5.5 An Additional Completion

We expand the alphabet of the user with an exception for dispenser failure and obtain a slightly modified interface:

interface USR-DF
begin
  with TMU-DF
    opr dispense(*sum* : nat) **throws** dispense$_{DF}$
    opr returnCard()
    opr insertCard(**out** *card* : nat)
    opr giveCode(**out** *code* : nat)
    opr withdraw(**out** *sum* : nat)
    opr query(**out** *choice* : nat)
end

This exception manifests itself at the semantic level by an additional termination event[4] between the teller machine m and the user u:

$$m \leftarrow u.\text{dispense}_{DF}(sum).$$

This event represents error detection in the formalism, in the example we can then express the necessary actions to regain the abstract requirement of TMA after the occurrence of faults.

[4] More formally, the event should include both the sum intended for dispensal as input and in addition the sum actually dispensed as output. The reasoning, however, would remain unchanged.

5.6 Repair of the Broken Dispenser

Repair of the teller machine is represented by an interface R with a method

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

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
begin
  with TMR
    opr repair()
  asm $\mathcal{H}$ **prp** caller↔this.repair()*
  inv true
end

Let r be an object which supports the interface R. The interface of the teller machine is then defined in the usual way.

interface TMR (r:R)
begin
  asm $\mathcal{H}$ **prp** this↔caller.repair()*
  inv true
end

5.7 The DF Fault-Tolerant Teller Machine

Finally, we specify the interleaving of the DF-tolerant interfaces of the teller machine. To achieve this, multiple inheritance for interfaces is used:

interface TM-DF(u: USR, c: C, r: R)
  inherits TMU-DF(u), TMC(c), TMR(r)
begin
  asm $\mathcal{H}$ **prp** (start(x,y) [goodCode(x,y)
                goodWithdraw$(x,y)^*$
                  (badWithdraw(x,y) | quit)|badCode(x,y)] • x,y : nat)*
  inv $\mathcal{H}$ **prp** (start(x,y) [goodCode(x,y)
                goodWithdraw$(x,y)^*$
                  (badWithdraw(x,y) | quit)|badCode(x,y)] • x,y : nat)*
  where
    start(x,y) == this↔u.insertCard(x) this↔u.giveCode(y)
    goodCode(x,y) == this↔c.authorize$(x,y;true)$
    badCode (x,y) == this↔c.authorize$(x,y;false)$ this↔u.returnCard()
    goodWithdraw(x,y) == this↔u.query(;"wd")
                              wrongAmount$(x,y)^*$ Dispense(x,y)
    wrongAmount(x,y) == this↔u.withdraw(sum)
                          this↔c.debit$(sum,x,y;false)$ • sum : nat
    okAmount(x,y,sum) == this↔u.withdraw(sum) this↔c.debit$(sum,x,y;true)$
    Dispense(x,y) == okAmount(x,y,sum) this↔u.dispense(sum) • sum : nat
    Dispense$_{DF}(x,y,sum-sum')$ == this↔u.withdraw(sum)

$$\text{this}{\leftrightarrow}u.\text{dispense}_{DF}(sum') \bullet\ sum' < sum \leq 3000$$

$$\text{quit} == \text{this}{\leftrightarrow}u.\text{query}(;\text{“end”})\ \text{this}{\leftrightarrow}u.\text{returnCard}()$$

$$\text{badWithdraw}(x, y) == \text{this}{\leftrightarrow}u.\text{query}(;\text{“wd”})\ \text{wrongAmount}(x, y)^*$$
$$\text{Dispense}_{DF}(x, y, sum)\ \text{this}{\leftrightarrow}c.\text{credit}(x, sum)$$
$$\text{this}{\leftrightarrow}u.\text{returnCard}()\ \text{this}{\leftrightarrow}r.\text{repair}() \bullet sum : \text{nat}$$

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.

5.8 Correctness for DF Fault-Tolerance

We want to show that the proposed interface TM-DF is correct with respect to the intolerant one originally specified. Obviously we want the treatment of dispenser failure to be in the masking/nonmasking category of fault tolerant refinement, so we want to show that TM-DF is a masking/nonmasking fault-tolerant refinement (relation $\mathcal{R}_2$) of TM (for any object m such that m: TM-DF). Let us identify the extension h_e to the trace h, where the dispenser failure has just occurred:

$$h_e == h \vdash m{\leftrightarrow}u.\text{returnCard}() \vdash m{\leftrightarrow}c.\text{credit}(x, sum) \vdash m{\leftrightarrow}r.\text{repair}()$$

After the fault has been repaired, the protocol recommences, so we can let the fault free equivalent to h_e be the empty trace ε.

So far we have not discussed the relationship between the FT specification and the abstract requirement (section 5.2). 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 teller machine session can break the abstract requirement, i.e. the invariant of TMA. Such an erroneous session will be on the form

$$\ldots m{\leftarrow}u.\text{insertCard}(x)\ m{\leftarrow}c.\text{debit}(sum, x, \_, true)$$
$$m{\leftarrow}u.\text{dispense}_{DF}(sum')\ m{\leftarrow}c.\text{credit}(sum - sum')\ldots$$

when all irrelevant events have been 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,

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

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

Assuming deadlock determinism, we get masking fault-tolerance. It can immediately be observed that the difference between masking and nonmasking fault-tolerance, when considering the safety specifications only, is very much a question of the formulation of the abstract requirement in TMA. For this example, if we were to demand that the invariant of TMA held when $\text{last}(h) = {\leftarrow} u.\text{returnCard}()$, we would get non-masking fault-tolerance instead.

Finally, TM-DF is deadlock deterministic, under the assumption that the environment does not deadlock, i.e. that all method calls to the environment are eventually terminated. The introduction of time-out termination events into the formalism would remove this assumption, as deadlock in the environment could then be handled locally.

6 Conclusion

In this paper the fault-tolerance notions of [3] are interpreted within a framework based on trace semantics. Different types of fault-tolerance are distinguished by the preservation of safety and liveness properties when tolerance for a fault is included in the design.

The difficulties involved in reasoning about liveness with prefix closed sets of finite traces leads to the identification of deadlock deterministic objects, for which liveness reasoning can be done within these prefix closed sets of finite traces. This property, however, cannot itself be defined within the formalism, so a broader approach is taken from which the original formalism can be derived.

It is necessary to distinguish between the preservation of an abstract system property and the actual refinement of a specification, although they correlate. This is due to the different levels of abstraction encountered. The intolerant specification is presumed to implement the abstract requirement, so the tolerant specification relates to both the abstract and the intolerant specifications. We refer to these relations as abstraction and refinement, respectively. In the case of refinement, the distinction between nonmasking and masking fault-tolerance disappears as this distinction depends on the level of abstraction.

A stepwise development of specifications is considered where an intolerant specification is refined to include tolerance for faults. The exact refinement relation to be used depends upon the type of fault-tolerance needed for each fault considered, i.e. on how abstract properties of the intolerant specification are preserved. The different abstraction and refinement relations are given precise definitions within the formalism. This distinction between abstraction and refinement seems to add clarity to the stepwise introduction of fault-tolerance in the design of critical systems.

A case study is developed in the specification language OUN 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-out, which can be handled like the faults in this paper, provides a natural extension to the formalism and allows a stronger notion of liveness.

References

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

[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), pp. 63–78, January 1998.

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

[5] F. Cristian, *A Rigorous Approach to Fault-Tolerant Programming*, IEEE transactions on Software Engineering, 11(1), pp. 23–31, January 1985.

[6] 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.

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

[8] E. B. Johnsen, *An Exercise in Fault Tolerance*, Proceedings of Norsk Informatikkonferanse, Tapir, 2000.

[9] 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.

[10] J. C. Laprie, *Dependability of computer systems: from concepts to limits*, in 1998 IFIP International Workshop on Dependable Computing and its Applications, Johannesburg (South Africa) 1998, pp. 108 – 126.

[11] Z. Liu and M. Joseph, *Stepwise Development of Fault-Tolerant Reactive Systems*, in Proc. of FTRTFT'94, Lecture Notes in Computer Science, 863, (1994), pp 529–546.

[12] T. S. Norvell, *On Trace Specifications*, technical report 305, Communications Research Laboratory, McMaster University, July 1995.

[13] O. Owe and I. Ryl, *A notation for combining formal reasoning, object orientation and openness*, Research report 278, Department of Informatics, University of Oslo, 1999.

[14] S. Owre, J. Rushby, N. Shankar and F. von Henke, *Formal Verification for Fault-Tolerant Architectures: prologomena to the Design of PVS*, IEEE transactions on Software Engineering 21(2), pp. 107–125, February 1995.

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

[16] H. Schepers and J. Hooman, *A trace-based compositional proof theory for fault tolerant distributed systems*, Theoretical Computer Science 128 (1994) pp. 127–157.

[17] S. D. Stoller, *A Method and Tool for Analyzing Fault-Tolerance in Systems*, Ph.D. Thesis, Cornell University, May 1997.

[18] 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.

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

A Summary of Notation for Sequences

The notation for operations on sequences used in this paper is presented briefly. The finite sequences over an alphabet α are denoted Seq[α]. These sequences can be defined inductively. Let $a, b \in \alpha$ and $h, h' \in \text{Seq}[\alpha]$. The sequences over α are constructed inductively as follows:

- ε denotes the empty sequence, and
- $h \vdash a$ denotes the sequence obtained when a is right appended to h.

From these definitions, left append and concatenation can be defined.

$$\begin{aligned} &a \dashv \varepsilon == \varepsilon \vdash a \\ &a \dashv (h \vdash b) == (a \dashv h) \vdash b \\ &h \vdash\!\!\dashv \varepsilon == h \\ &h \vdash\!\!\dashv (h' \vdash a) == (h \vdash\!\!\dashv h') \vdash a \end{aligned}$$

The finite prefixes of a sequence are defined. Let h be a finite sequence and let h' be a nonempty sequence ($h' \neq \varepsilon$). Then

$$h < h \vdash\!\!\dashv h'.$$

The reflexive closure of prefixing is denoted $\leq$. Restriction (filtering) on sequences is needed. We denote by h/α the sequence h restricted to elements of a set α, defined formally as follows:

$$\begin{aligned} &\varepsilon/\alpha == \varepsilon \\ &(h \vdash a)/\alpha == (h/\alpha) \vdash a \text{ if } a \in \alpha \\ &(h \vdash a)/\alpha == (h/\alpha) \text{ otherwise} \end{aligned}$$

By extension, we denote by h/o the projection of the sequence onto the set of events where an object o is involved, i.e. onto the set

$$\{o_1 \rightarrow o_2.m(\ldots) \mid o \in \{o_1, o_2\}\} \cup \{o_1 \leftarrow o_2.m(\ldots) \mid o \in \{o_1, o_2\}\},$$

and $h/\rightarrow o$ by and $h/\leftarrow o$ the projections on initiations and terminations of methods of o (with respect to an interface), defined as

$$\begin{aligned} &h/\rightarrow o == \{o' \leftarrow o.m(\ldots)\} \\ &h/\leftarrow o == \{o' \rightarrow o.m(\ldots)\} \end{aligned}$$

The function $\sharp(h)$ calculates the length of a sequence h, defined inductively by

$$\begin{aligned} &\sharp(\varepsilon) == 0 \\ &\sharp(h \vdash m) == \sharp(h) + 1. \end{aligned}$$

B OUN Patterns

In the specification language, legal traces are described by assumption and invariant predicates. We will often use patterns to define these predicates. A pattern is a regular expression with some added features. Generally, the traces are described by the legal prefixes of a pattern: a trace is a legal prefix of a pattern if it is a left prefix of a trace described by that pattern. We use the notation

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

to denote this predicate, where $\mathcal{H}$ ranges over traces and P is a pattern. This relation is well adapted for the OUN formalism because if the relation holds for some trace h, it also holds for all left prefixes of h. This is well suited for the prefix closure property of finite trace semantics. A pattern can consist of the following notation:

- Events 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 0 or more occurrences of the pattern a.
- a^+ is a pattern describing 1 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 0 or 1 occurrence of the pattern a.

Patterns have (formal) parameters, declared by scoping rules. We use the scoping mechanism to

- declare the type of the variable, and to
- declare the scope of the variable.

If we want to describe a pattern as a loop, where the events take different argument values for each run of the protocol, 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 $(a(x)\ b(x) \bullet x : \mathrm{Nat})^*$, but not by $(a(x)\ b(x))^* \bullet x : \mathrm{Nat}$. When arguments of events are not known to be of particular importance to the process of pattern recognition, they are left unspecified by a wildcard "_". We are also allowed to define subpatterns by means of auxiliary functions. Such auxiliary definitions are inherited in subinterfaces and may be used there for specification purposes.