Presented at the Formal Methods Europe 1993 Symposium, January 1993.

Using Relative Refinement for Fault Tolerance[*]

Antonio Cau[**] and Willem-Paul de Roever[***]

Institut für Informatik und Praktische Mathematik II
Preußerstr. 1-9
Christian-Albrechts-Universität zu Kiel
D-24105 Kiel, Germany

Abstract. A general refinement methodology is presented based on ideas of Stark, and it is explained how these can be used for the systematic development of fault-tolerant systems. Highlights are: (1) A detailed and comprehensive exposition of Stark's temporal logic and development methodology. (2) A formalization of a general systematic approach to the development of fault-tolerant systems, accomplishing increasing degrees of coverage with each successive refinement stage. That is, faults are already identified and modeled at the first implementation level, which is shown to be a relative refinement, i.e., correct for all computations in which faults do not occur. The second implementation is a fail-stop implementation, i.e., an implementation that stops on the first detected occurrence of a fault. This implementation is also a relative refinement, i.e., correct in all computations in which the program never stops. The final implementation is correct in all computations, except those that display severe faults that violate the fault-tolerance assumptions, such as all n components failing in an n-way redundant way in case of stable storage. (3) A detailed example of a multi-disk system providing stable storage, illustrating this general methodology.

1 Introduction

Current formal methods are far from solving the problems in software development. The simplest view of the formal paradigm is that one starts with a formal specification and subsequently decomposes this specification in subspecifications which composed together form a correct refinement. These subspecifications are decomposed into "finer" subspecifications. This refinement process is continued until one gets subspecifications for which an implementation can easily be given. This view is too idealistic in a number of respects. First of all, most specifications of software are wrong (certainly most informal ones unless they have been formally analyzed) and contain inconsistencies [9]. Secondly, even if a formal specification is produced, this is only after a number of iteration steps because writing a correct specification is a process whose difficulty is comparable with

[*] Partially supported by ESPRIT/BRA project 6021 (REACT)
[**] E-mail: ac@informatik.uni-kiel.d400.de
[***] E-mail: wpr@informatik.uni-kiel.d400.de

that of producing a correct implementation, and should therefore be structured, resulting in a number of increasingly less abstract layers with specifications which tend to increase in detail (and therefore become less readable [8]). Thirdly, even an incorrect refinement step may be useful in the sense that from this incorrect refinement step one can easier derive the correct refinement step. This is especially the case with intricate algorithms such as those concerning specific strategies for solving the mutual exclusion problem. An interesting illustration of this third view is provided by E.W. Dijkstra's "Tutorial on the split binary semaphore" [5] in which he solves the readers/writers problem by subsequently improving incorrect refinement steps till they are correct. If this master of style prefers to approximate and finally arrive at his correct solution using formally incorrect intermediate stages, one certainly expects that a formally correct development process for that paradigm is difficult to find! The strategy described in [5] is necessarily informal, reflecting the state of the art in 1979. We have formalized this strategy in [3].

In the present paper we present a formal development strategy for deriving a correct refinement step using relative correct intermediate stages, and its application to fault tolerant systems. The formal strategy is as follows: one starts with an implementation for a specified fault tolerant system. This implementation contains some faults, i.e., the refinement step is incorrect because of these faults. It is however relative correct because when these faults don't occur it is a correct implementation. In the next step we try to detect these faults, i.e., we construct a detection layer upon the previous implementation that stops that implementation when it detects an error caused by these faults. This is called a fail-stop implementation [7] and it is better than the previous one because now at least the implementation stops on the occurrence of a fault. The second implementation is also relative correct because when no faults occur and the detection layer doesn't detect any error it is correct. In the third approximation we recover these faults, i.e., we don't stop anymore upon the detection of an error but merely recover the fault by executing some special program that neutralizes that fault. This third approximated refinement step is correct under the assumption that certain conditions are fulfilled, which exclude the occurrence of faults different from the ones neutralized, i.e., it is relative correct. We use Stark's formalism in order to describe this process of approximation. In this formalism a specification is separated into a safety (machine) part and a liveness (validity) part. The machine part is used by us for describing the faulty implementation and the validity part for restricting the machine part to the correct behavior of that implementation. It is this separation that enables us to handle incorrect approximations: although the machine part of the implementation doesn't refine the specification, the intersection of the machine part and the validity part of the implementation does refine the specification, indeed.

The structure of the paper is as follows: In sect. 2 we introduce Stark's formalism and give some simplifications/improvements based on [4]. We present in sect. 3 the formal development of a fault tolerant system for stable storage, using the proto-formalization of [10]. Section 4 contains a conclusion.

2 Stark's Formalism

2.1 Introduction

In this section we present Stark's Dense time Temporal Logic (DTL) formalism for describing correct refinement steps ([11, 12] are not easily accessible) and our use of it to describe incorrect refinement steps. Our simplification of this logic is based on that of [4]. In sect. 2.2 the semantic notions of module and machine are defined. The idea is that modules are a kind of black box notion. Machines are introduced to relate them to actual computations. Stark's machine notion is a handy normal form reminiscent of Lamport's notion of "machine closure" [1], and makes it easy to define liveness properties as global restrictions on the machine's behavior. Stark makes a distinction between local -safety- properties and global properties, such as liveness. Here is where we depart from Stark's formalism: instead of using the global condition for defining liveness, by excluding computations which do not satisfy certain fairness assumptions, we use it for describing the allowed behavior of a machine by removing behavior which is undesirable in other respects. Another important reason for using Stark's formalism is that Stark's dense time temporal logic deals with the stuttering problem in an elegant abstract way, with the added advantage that one can express *within this logic* what a correct development step is (also Lamport's notion of correct refinement step can be expressed within that logic). We adopt in sect. 2.3 a slightly simplified/improved version of the logic as defined in [4]. Figure 1 illustrates the underlying model. A salient feature of Stark's DTL is the "immediately after" operator $'$.
In sect. 2.4 the semantic notion of a correct refinement step of modules is defined. The idea is that a module can play a certain role in a refinement step. This role can be an abstract one, a composite one, or a concrete one. During a refinement step an abstract module is implemented by a collection of concrete modules. A third kind of module, the composite module, relates these abstract and concrete modules with each other. It first defines how the concrete modules are composed and secondly describes how internal behavior, arising from internal events, is abstracted away from within this composed module. So the composite module defines actually the refinement relation between the abstract and concrete modules.

In sect. 2.5 machines and their allowed computations are related to correct refinement steps, expressed by verification conditions. These verification conditions are expressed in DTL. This section also explains how Stark's formalism can be used to describe relative refinement steps.

2.2 Modules and Machines

In [11] a method for specifying reactive systems is introduced. Such systems are assumed to be composed of one or more *modules.* A module is specified by the pair $\langle E, B \rangle$ where E is syntactic and denotes a finite set of possible events, called *interface*, and B is semantic and denotes its *allowed (visible) behavior.*

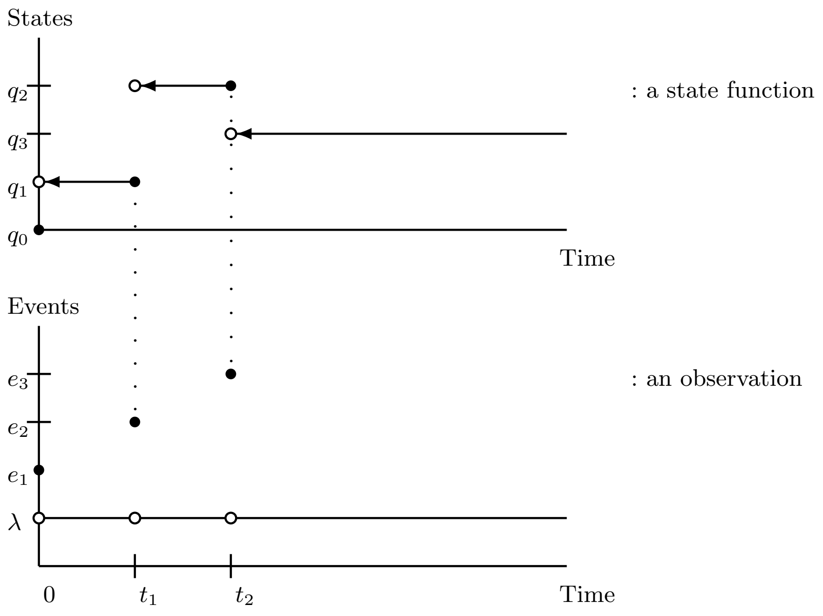

Fig. 1. This picture illustrates the notion of a state function and observation, which together characterise the notion of computation of a machine. It illustrates the following computation: the initial state is q_0, on the occurrence of event e_1 the state changes in q_1. Between time 0 and time t_1 there are no interesting event occurrences, only of the uninteresting stuttering event λ, so the state doesn't change until the next interesting event e_2 occurs resulting in state q_2, and so on.

An *event* is an observable instantaneous occurrence during the operation of a module, that can be generated by that module or its environment and that is of interest at the given level of abstraction. E always contains a λ_E-event which represents all, at that level, uninteresting events. We model faults also with events because a fault is an observable instantaneous occurrence during the operation of a module. It is generated by the environment of the module and is of interest at our level of abstraction.

The B-part of specification $\langle E, B\rangle$ characterizes the allowed behavior of the module. An *observation* x over interface E is a function from $[0, \infty)$ -representing the time domain- to E, such that $x(t) \neq \lambda$ for at most finitely many $t \in [0, \infty)$ in each bounded interval; this means that in any bounded interval only a finite number of interesting events can occur (the so called finite variability condition). Figure 1 illustrates the notion of observation. At time 0 the event e_1 occurs, at time t_1 event e_2, and at time t_2 event e_3; at all other moments the event λ occurs.

Let $Obs(E)$ denote the set of all observations over E. Then the *allowed behavior* B is a subset of $Obs(E)$. $Beh(E)$ denotes the set of all behaviors at interface E, i.e., the set of all subsets $Obs(E)$..

Until now we have specified the allowed behavior of a module by a set of observations. We now introduce a state-transition formalism to generate this set. In this state-transition formalism, at any instant of time a module can be thought of as being in a *state*. In each state an event may cause a transition to another state. With a state-transition relation one specifies which event can occur in a particular state and what the state will be after the occurrence of that event. Thus a state-transition specification describes the desired functioning of a module in terms of computations of an idealized machine, i.e., an observation now corresponds with a computation of that machine.

One can divide the properties that can be specified by the state-transition technique in two classes. The first class consists of the so called *local (safety) properties*, which describe how an event causes a state transformation to the next state. The second class consists of the so called *global properties*; these describe the global relationship between events and states that cannot be directly described in terms of state successor relations.

The local properties are specified by the above mentioned machine and the global properties are specified by defining a set of *validity conditions* to be satisfied by computations of that machine. The set of computations that satisfy the validity conditions is called the *set of valid computations*. The intersection of this set with the set of computations that are generated by the machine describes the allowed behavior of the corresponding module.

The machine M that specifies the local properties of a module is defined as follows: $M = (E_M, Q_M, IQ_M, TR_M)$ where:

- E_M : is the interface of M; events labeled with a$\downarrow$ are input events, events labeled with a$\uparrow$ are output events, and events without an arrow are internal events,

- Q_M : is the countably infinite set of states of M; a state is a function from the set of observable variables Var and freeze variables F ($Var \bigcap F =$) to the set of values Val, i.e., $Q_M : (Var \bigcup F) \rightarrow Val$. Note: the "normal" variables will be bold faced in order to distinguish them from freeze variables.
- IQ_M : a non-empty subset of Q_M, the set of initial states,
- TR_M : the state-transition relation (finite), $TR_M \subseteq Q_M \times E_M \times Q_M$, such that for all q the stuttering step $\langle q, \lambda_{E_M}, q\rangle \in TR_M$ and that for all $q, r \in Q_M$ $\langle q, \lambda_{E_M}, r\rangle \in TR_M$ iff $r = q$. This latter condition expresses the requirement that λ-events can not change a state. Furthermore M is input-cooperative, i.e., in every state of M any input event can be received; therefore the state-transition relation for an input event ist defined for all states of M.

A *state function* over a set of states Q is a function $f : [0, \infty) \rightarrow Q$ such that for all $t \in [0, \infty)$, there exists $\varepsilon_t > 0$ such that f is constant on intervals of type $(t - \varepsilon_t, t] \cap [0, \infty)$ and $(t, t + \varepsilon_t]$. We write $f(t^{\leftarrow\bullet})$ for the value of the state just before and at time t (the first type of interval) and write $f(t^{\circ\rightarrow})$ for the value of f just after time t (the second type of interval). Figure 1 illustrates the notion of state function. At time 0 the machine is in state q_0, in interval $(0, t_1]$ the machine is in state q_1, and so on.

A *history* over an interface E and a state set Q is a pair $X = \langle Obs_X, State_X\rangle$, where Obs_X is an observation over E (a function from $[0, \infty)$ to E), and $State_X$ is a state function over Q. These two concepts are related by the notion of *computation* of a machine M. A computation of M intuitively expresses that an observation and a state function fit together in that at any moment of time t any triple consisting of (1) the state just before and including t, (2) the observation at t, and (3) the state just after t, belongs to the state transition relation of M (see fig. 1). For a formal definition we need the notion of *step at* t in history X.

Let $Hist(E, Q)$ denote the set of all histories over interface E and state set Q. If $X \in Hist(E, Q)$ and $t \in [0, \infty)$, then define the *step* occurring at time t in X by: $Step_X(t) = \langle State_X(t^{\leftarrow\bullet}), Obs_X(t), State_X(t^{\circ\rightarrow})\rangle$.

A *computation of a machine* M is a history $X \in Hist(E_M, Q_M)$ such that: $State_X(0) \in IQ_M$ and $Step_X(t) \in TR_M$ for all $t \in [0, \infty)$.

$Comp(M)$ denotes the set of all computations of M, and $Reachable_M$ the set of reachable states of M.

2.3 Stark's dense linear time logic DTL

As mentioned above, the global properties are described by a set of validity conditions. Stark uses a dense time temporal logic to describe these validity conditions. Our modified temporal logic DTL looks like the one of Stark and is defined as follows:

Syntax

variables are elements of Var
values of variables are elements of Val

freeze variables are elements of F; $F \cap Var = \emptyset$
events are elements of E_M
special symbol e
event term $\mathbf{e} = f$ where f denotes an element of E_M
state terms x and x' where x is an element of Var, denoting the current state, and x' is the "immediately after" state ($'$ denotes the "immediately after" temporal operator)
terms can be event terms, state terms, freeze variables or function symbols
quantification over freeze variables using $\forall, \exists$
formulae are built from terms, and relation symbols using boolean connectives, quantification and
temporal operators $\Box$, $\Diamond$ and $'$

Examples of DTL formulae

$(\mathbf{x} = 0 \wedge e = d_0) \rightarrow \mathbf{x}' = 1$ (a state-transition), $\Box\mathbf{x} > 0$ (a safety property), and $\Box(\mathbf{x} = 0 \rightarrow \Diamond\mathbf{x} > 0)$ (a liveness property).

Semantics

We take as semantic model histories (a history is a pair $\langle Obs_h, State_h\rangle$). Let $h^{(\tau)}$ denote the history $\lambda t.h(t + \tau)$.
For all freeze variables $v \in F$, $v(h) = State_h(0)(v)$. Note: because v is a freeze variable the value doesn't change in a history, at time 0 it is initialized.
For all variables $v \in Var$, $v(h) = State_h(0^{\leftarrow\bullet})(v) = State_h(0)(v)$.
For all variables $v \in Var$, $v'(h) = State_h(0^{\circ\rightarrow})(v)$.
For $\mathbf{e}$, $\mathbf{e}(h) = Obs_h(0)$.
For f with interpretation $\overline{f}$, and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)(h) = \overline{f}(t_1(h), \ldots, t_n(h))$.
$h \models R(t_1, \ldots, t_n)$ if $\overline{R}$ is the interpretation of R, and $t_1, \ldots, t_n$ are terms, and $\overline{R}(t_1(h), \ldots, t_n(h))$ holds;
$h \models \neg\varphi$ if $h \not\models \varphi$;
$h \models \varphi \rightarrow \psi$ if $h \models \neg\varphi$ or $h \models \psi$;
$h \models \exists x.\varphi$ if there exists an assignment $State_{h'}(0)$ differing from $State_h(0)$ only in the value assigned to freeze variable x such that $h' \models \varphi$;
$h \models \Diamond\varphi$; if there exists an $t \in [0, \infty)$ such that $h^{(t)} \models \varphi$;
$h \models \Box\varphi$; if for all $t \in [0, \infty)$ $h^{(t)} \models \varphi$;

The initial states and the transition relation of a machine, can and will be, from now on be expressed as DTL formulae. The *enabling condition* of an event in a machine M, denoted by $Enabled_M(e)$ is the precondition for the transition that corresponds with that event. The local properties of a module Z can now be expressed by formula $IQ_Z \wedge \Box TR_Z$. Thus $\mathrm{Comp}(M_Z) \overset{\text{def}}{=} \{X \in Hist(E_Z, Q_Z) \mid X \models IQ_Z \wedge \Box TR_Z\}$. The liveness properties can now be added, expressed by some extra DTL formula V_Z, the *validity condition.* The complete behavior of module Z is the following set of histories: $\{X \in \mathrm{Comp}(M_Z) \mid X \models V_Z\}$, and is described by DTL formula $IQ_Z \wedge \Box TR_Z \wedge V_Z$.

2.4 Modules and correct refinement steps

In Stark's view there are three kinds of roles a module can play during a refinement step. The first one is that of an *abstract* module -then it serves as a high level specification of a system. The second one is that of a *concrete* one, serving as a lower level specification of a system component. A concrete module may become an abstract module in the next refinement step. The third one is that of a *composite* one, defined as the Cartesian product of the abstract module and the concrete modules, and used as a device for defining a refinement mapping between the abstract module and the parallel composition of the concrete modules for that development step. The interface of the composite module is the Cartesian product of the interfaces of the abstract module and the concrete modules.

To specify a refinement step of a system one therefore needs an abstract module, a composite module, and one or more concrete modules. An *interconnection* relates these modules with each other, i.e., it relates the interface of the composite module with both the interface of the abstract module and the interface of the concrete modules. Hence it characterizes a refinement relation between an abstract module and a set of concrete modules. It defines how an event on the abstract level is implemented by events on the concrete level.
An interconnection $\mathcal{I}$ is a pair $\langle \alpha, \langle \delta_j \rangle_{j \in J} \rangle$ where:

- α denotes a function from the interface $E = A \times \prod_{j=1}^{N} F_j$ of the composite module, with $\lambda_E = \langle \lambda_A, \lambda_{F_1}, \ldots, \lambda_{F_N} \rangle$, to the interface A of the abstract module such that $\alpha(\lambda_E) = \lambda_A$ holds; α is called an *abstraction* function,
- δ_j denotes a function from interface $E = A \times \prod_{j=1}^{N} F_j$ of the composite module, with $\lambda_E = \langle \lambda_A, \lambda_{F_1}, \ldots, \lambda_{F_N} \rangle$, to interface F_j of the j-th concrete module such that $\delta_j(\lambda_E) = \lambda_{F_j}$ holds; δ_j is called a *decomposition* function.

So intuitively the requirement upon both α and the δ_j's is that uninteresting events of the composite module are not turned into interesting events of the abstract or concrete modules.

The definition of interconnection can easily be extended to (hold between the) behaviors of the mentioned modules. When $\mathcal{I}$ is an interconnection between the interfaces of the modules, $\mathcal{I}^*$ denotes the corresponding interconnection between the behaviors of the modules. We shall omit the * from now on when we use the interconnection between behaviors of modules. Although this is not mathematically correct, we hope it is clearer for the reader.

With the above defined interconnection the allowed behavior of the composite module can be defined as $\alpha^{-1}(B_A) \cap \bigcap_{j=1}^{N} \delta_j^{-1}(B_j)$ where B_A denotes the allowed behavior of the abstract module A and B_j that of module j. A *refinement step* is defined as a triple $\langle \mathcal{I}, S_A, \langle S_j \rangle_{j \in J} \rangle$ where $\mathcal{I}$ is the interconnection (between behaviors), S_A the specification $\langle A, B_A \rangle$ ($B_A \in Beh(A)$ denotes the set of allowed behaviors of the abstract module) of the abstract module, and S_i the specification $\langle F_i, B_i \rangle$ ($B_i \in Beh(F_i)$ denotes the set of allowed behaviors of concrete module i) of concrete module i.

A refinement step is *correct* if the following, hopefully now intuitively obvious, inclusion holds: $\bigcap_{j=1}^{N} \delta_j^{-1}(B_j) \subseteq \alpha^{-1}(B_A)$.

2.5 Describing (relative) correct refinement steps in DTL

With every kind of module we can associate a machine -whether abstract, concrete or composite. For concrete and abstract modules this is obvious, but how is this done for a composite module? First we assume that the sets of states of the abstract and the concrete machines are disjoint. So every machine has its own set of states. If we have an abstract machine M_A, described by temporal formula $IQ_A \wedge \Box TR_A$, concrete machines M_j, described by temporal formula $IQ_j \wedge \Box TR_j$, and if we have furthermore an interconnection $\mathcal{I} = \langle \alpha, \langle \delta_j \rangle_{j \in J} \rangle$ that links both kinds of machines, then we can construct the composite machine M_c as follows:

- The interface $E_c \stackrel{\text{def}}{=} E_A \times \prod_{j=1}^{N} F_j$ as in the definition of the composite module.
- The set of states $Q_c \stackrel{\text{def}}{=} Q_A \times \prod_{j \in J} Q_j$, i.e, the product of (1) the set of states of the abstract machine, and (2) the product of the sets of states of all the concrete machines.
- The set of initial states IQ_c of M_c is defined by: $IQ_c \stackrel{\text{def}}{=} IQ_A \wedge \bigwedge_{j \in J} IQ_j$.
- In order to express the state-transition relation TR_c of the composite machine, we use the definitions of α and the δ_j's to transform event terms in TR_A and TR_j into event terms of TR_c. Event term $\mathbf{e} = d$ in TR_A is transformed into $\mathbf{e} = \alpha^{-1}(d)$ and event term $\mathbf{e} = f$ in TR_j into $\mathbf{e} = \delta_j^{-1}(f)$ (Note: α^{-1} and δ_j^{-1} are not functions (in general), so $\alpha^{-1}(d)$ and $\delta^{-1}(f)$ are sets. But these sets are finite because the interfaces are finite. Thus $\mathbf{e} = \alpha^{-1}(d)$ and $\mathbf{e} = \delta_j^{-1}(f)$ are not proper defined expressions but can be translated into $\bigvee_{g \in \alpha^{-1}(d)}(\mathbf{e} = g)$ and $\bigvee_{h \in \delta_j^{-1}(f)}(\mathbf{e} = h)$. But nevertheless we will use $\mathbf{e} = \alpha^{-1}(d)$ and $\mathbf{e} = \delta_j^{-1}(f)$ for clarity although it is mathematically wrong.)
  We introduce the following notation:
  $[f]_\alpha \stackrel{\text{def}}{=} f[\alpha^{-1}(d)/d]$ for $d \in E_A$ and $[f]_{\delta_j} \stackrel{\text{def}}{=} f[\delta_j^{-1}(f)/f]$ for $f \in F_j$.
  Then the state-transition relation TR_c of M_c is defined as follows:
  $TR_c \stackrel{\text{def}}{=} \Box([TR_A]_\alpha \wedge \bigwedge_{j \in J}[TR_j]_{\delta_j})$,
  with $[TR]_\beta$ for abstraction/decomposition function β defined by replacing any occurrence $\mathbf{e} = f$ in TR by $\mathbf{e} = [f]_\beta$.

The correctness condition of the refinement step, as seen above, is as follows:
$$\bigcap_{j=1}^{N} \delta_j^{-1}(B_j) \subseteq \alpha^{-1}(B_A).$$
In the present formalism, this translates into
$\bigcap_{j=1}^{N} \delta_j^{-1}(\{X \in \mathit{Comp}(M_j) : X \models V_j\}) \subseteq \alpha^{-1}(\{X \in \mathit{Comp}(M_A) : X \models V_A\})$.
The following temporal formula expresses this condition:
$$\bigwedge_{j \in J}[IQ_j \wedge \Box TR_j \wedge V_j]_{\delta_j} \rightarrow [IQ_A \wedge \Box TR_A \wedge V_A]_\alpha$$
Due to the separation of the allowed behavior into a machine part (a pure safety DTL formula) and a validity part (a pure liveness DTL formula) -see [2]

for an explanation of pure safety and pure liveness- we can split this verification condition into two verification conditions, one for machines and one for validity:

– *maximality* : any event that can be generated by the system of concrete machines can also be performed by the abstract machine, i.e.,
  $\forall e \in E_c.(Reachable_c \wedge \bigwedge_{j \in J} Enabled(\delta_j(e))) \rightarrow Enabled(\alpha(e))$
  where $Reachable_c$ is a condition that checks if a state of the composite machine is reachable, $Enabled(\delta_j(e))$ is the enabling condition of event $\delta_j(e)$ of machine j, and $Enabled(\alpha(e))$ is the enabling condition of event $\alpha(e)$ of the abstract machine.
– *validity* : any allowed computation of each concrete machine corresponds with an allowed computation of the abstract machine, i.e.,
  $Comp(M_c) \models (\bigwedge_{j \in J}[V_j]_{\delta_j}) \rightarrow [V_A]_\alpha$
  where V_j is the validity condition of module j, and V_A is the validity condition of the abstract module.

Next we explain how to describe *relative correct* refinement steps in the development of a program, as promised in sect. 1. Well, as said before, instead of using the validity condition for expressing a liveness condition, we use it for characterizing the correct computations of a machine. Now in general that latter condition is not a pure liveness condition. Furthermore, since the machine part describes the implementation as developed until now, it contains in general also the unallowed computations. But since the total specification is the intersection of the machine part and the validity part, the total specification is still *relatively correct.* More formally:
Let the abstract specification be denoted by $IQ_A \wedge \Box TR_A \wedge V_A$, where V_A is a pure liveness condition, i.e., the machine part of the abstract specification doesn't characterize unallowed computations.
Each concrete specification is denoted by $IQ_j \wedge \Box TR_j \wedge P_j \wedge L_j$ where P_j is the part that characterizes the allowed computations of the machine part and L_j is the pure liveness part, i.e., the validity condition has been split up into P_j and L_j. In the stable storage example of the next section the P_j part is always a safety condition that *disallows certain transition of TR_j from being taken.* This allows us to express the *relative* correctness of a refinement step as follows:
For correctness we have to prove, as seen above,
$\bigwedge_{j \in J}[IQ_j \wedge \Box TR_j \wedge P_j \wedge L_j]_{\delta_j} \rightarrow [IQ_A \wedge \Box TR_A \wedge V_A]_\alpha$.
Because (1) both P_j and TR_j are safety conditions, (2) the conjunction of two safety conditions is again a safety condition, and (3) P_j disallows certain transitions of TR_j from being taken, $\Box TR_j \wedge P_j$ can be transformed into $\Box TRnew_j$, which is also expressed as a safety condition. Note: $TRnew_j$ is a new transition relation. So we get the following:
$\bigwedge_{j \in J}[IQ_j \wedge \Box TRnew_j \wedge L_j]_{\delta_j} \rightarrow [IQ_A \wedge \Box TR_A \wedge V_A]_\alpha$.
This form allows one to use Stark's two verification conditions, because $TRnew_j$ is a pure safety condition and L_j a pure liveness condition.

3 Relative Refinement of Fault Tolerant Systems

3.1 Introduction

In this chapter we first introduce in sect. 3.2 a *general methodology* for proving fault tolerant systems correct. This general methodology uses the relative refinement concept of sect. 2.5. The remaining sections of this chapter give an illustration of this general methodology by applying it to a fault tolerant system consisting of a number of disks implementing stable storage. Section 3.3 introduces this application. In sections 3.4, 3.5, 3.6 and 3.7 the four steps of this general methodology are applied to the stable storage example.

3.2 The General Methodology

The general methodology consists of four steps. In the first step we give the abstract specification A (a DTL formula) of the fault tolerant system. In this specification no faults are visible, hence don't occur as observables. The designer's task is to give an implementation of this system under the assumption that only faults from certain classes can occur. These faults are called *anticipated faults*. These are faults which may affect the implementation in that they may give rise to errors in the state of the implementation, resulting subsequently in failures of that implementation. But the implementation must be such that these errors and faults -this is why they are called *anticipated* faults- are not leading to failures. This justifies the very term fault tolerance, and explains the very task to be carried out by fault tolerant systems. This task will be described in step 2,3 and 4 of the general methodology.

The second step of the general methodology, *identifies* the anticipated faults which can affect an implementation I. This implementation serves as first approximation to the final implementation of A. It should be clear that I is not a refinement of A because of the possible occurrences of anticipated faults. I is only a refinement when these faults do not occur, i.e., I is a *relative refinement* of A. We have seen in sect. 2.5 that this relative refinement step can expressed using ordinary refinement, i.e., $I \wedge NFO$ is a refinement of A where NFO expresses that the anticipated faults never occur. So in step 2 the proof obligation is:

$$I \wedge NFO \rightarrow A. \tag{1}$$

Note: we have omitted the abstraction and decomposition functions that were needed to describe a refinement step in DTL because these would complicate the explanation of the general methodology.

In the third step of our development one specifies *how these anticipated faults are detected*, i.e., one has to specify a detection layer DL_{fs} for these faults. This layer is added in bottom-up fashion to the implementation I of the second step and stops upon detection of the first error, i.e., DL_{fs} is a *fail-stop* implementation. So the second approximation to the final implementation consists of the parallel composition of I and DL_{fs}. Is this approximation a refinement of A? No, because when in I a fault occurs, and DL_{fs} detects the corresponding error, the

whole approximation stops. One would like to have (eventually) an approximation that doesn't stop. This means that one must consider $I \wedge NFO$ instead of I. But then DL_{fs} should detect no error because if it detects an error and the fact that no corresponding fault has occurred the detection layer is not "correct". So $I \mid\mid DL_{fs}$ is a relative refinement of A can be described by the following ordinary refinement:
$[(I \wedge NFO) \mid\mid (DL_{fs} \wedge NED)]_{DL_{fs}} \rightarrow A$ (2).
(Note: $\mid\mid$ can be expressed in DTL using certain abstraction and decomposition functions.) Here NED expresses that no errors are detected, and $[\ldots]_{DL_{fs}}$ the hiding of the detection layer. Proof obligation (2) is proved using the result of (1): under the assumption that $\neg NED \rightarrow \neg NFO$ (error detected implies corresponding fault occurred) the following holds
$[(I \wedge NFO) \mid\mid (DL_{fs} \wedge NED)]_{DL_{fs}} \rightarrow (I \wedge NFO)$
Using (1) one infers (2).

In the fourth step of our development one specifies the course of action after detection of an anticipated fault, i.e., *one specifies the corrective action to be undertaken after detection of such a fault.* This means in general that one needs *redundancy*, i.e., several copies of I and DL components, because when a detection layer DL detects an error, the state before that error has to be recovered and that can only be done by accessing another copy of I through its corresponding detection layer DL. Note that the DL component doesn't stop anymore on the detection of an error but merely waits for the corrective action to be undertaken. Say, we need N copies of I and DL. These will be abbreviated by respectively $\overline{I}$ and $\overline{DL}$. Thus the final implementation consists of $\overline{I} \mid\mid \overline{DL} \mid\mid ER$ where ER is the error recovery layer. This implementation is correct if following holds:
$[\overline{I} \mid\mid \overline{DL} \mid\mid ER \wedge REC]_{DL,ER} \rightarrow A$ (3).
Here REC is global restriction on the the kind of faults that can occur. One can prove the correctness of (3) using (2) (and therefore also (1)). First one proves the case when no faults occur:
$[(\overline{I} \wedge NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC)]_{DL,ER} \rightarrow A.$
Under the assumption that $NFO \rightarrow NED$ holds one can infer
$(\overline{I} \wedge NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC) \rightarrow (\overline{I} \wedge NFO) \mid\mid (\overline{DL} \wedge NED) \mid\mid (ER \wedge REC).$
And under the assumption that $REC \rightarrow \neg NED$ holds one can infer
$[(\overline{I} \wedge NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC)]_{ER} \rightarrow (I \wedge NFO) \mid\mid (DL_{fs} \wedge NED).$
Using (2) one now can infer
$[\overline{I} \wedge NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC)]_{DL,ER} \rightarrow A.$
The second case to be considered is when faults do occur
$[(\overline{I} \wedge \neg NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC)]_{DL,ER} \rightarrow A.$
Under the assumption that $NED \rightarrow NFO$ holds one can infer
$(\overline{I} \wedge \neg NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC) \rightarrow (\overline{I} \wedge \neg NFO) \mid\mid (\overline{DL} \wedge \neg NED) \mid\mid (ER \wedge REC).$
And under the assumption that $\neg NED \rightarrow REC$ holds (i.e., the detected error is recovered -this is actually a programming problem), one can infer
$[(\overline{I} \wedge \neg NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC)]_{ER} \rightarrow (I \wedge NFO) \mid\mid (DL_{FS} \wedge NED)$
Using step (2) one now can infer

$[(\overline{I} \wedge \neg NFO) \mid\mid \overline{DL} \mid\mid (ER \wedge REC)]_{DL,ER} \rightarrow A$.
Combining the two cases together one gets
$[\overline{I} \mid\mid \overline{DL} \mid\mid (ER \wedge REC)]_{DL,ER} \rightarrow A$.

Recapitulating: in the proof above the following assumptions have been made:
($NFO \leftrightarrow NED$: perfect detection,
($\neg NED \leftrightarrow REC$): perfect recovery and
step (1): implementation I is a relative refinement of A.

This ends our description of a general methodolology for proving fault-tolerant systems (qualitatively) correct.

3.3 Application: Introduction

Stable storage is defined as follows. A disk is used to store and retrieve data. During these operations some faults can occur in the underlying hardware. To make the disk more reliable one introduces layers for the detection and correction of errors, due to these faults. The system with these detection and correction layers is called "stable storage". This stable storage is a fault tolerant system because it stores and retrieves data in a reliable way under the assumption that faults from a certain class are recovered (corrected). This class consists of two kinds of faults. The first one consists of faults that damage the disk surface - the contents of the disk are said to be corrupted by these faults. The second one consists of faults that affect the disk control system, and results into the contents of the disk being read from or written to the wrong location. Notice that other kinds of faults, such as power failure or physical destruction of the whole stable storage system, are not taken into account. I.e., stable storage should function correctly provided such latter faults do not occur.

3.4 Application: First Step

Introduction In step 1, as seen in sect. 3.2, the abstract specification A of a stable storage system is given, i.e., the system as we ideally would like it to look like: no faults are observed. If they occur internally, they should be repaired by the system without leaving any observable trace. For that is the meaning of 'stable' here!

Specification The abstract specification of stable storage specifies the following: The user signals with an request event that he wants to read the contents of some location of a medium for stable storage. This medium then responds by sending the requested contents. The user can also signal with a write event that some data have to be written on some location of the medium. Note: we have a very simple stable storage medium that can handle only one request at a time. If the user requests the contents of location before the stable storage medium has responded to a previous request then our stable storage medium will get into the error state and will not respond anymore to any request from the user.

We specify this medium by a machine M (and V-set V) which is in this case rather simple because this is an idealized machine with no faults. It only ensures that the user and the stable storage medium communicate with each other correctly. The specification $A \stackrel{\text{def}}{=} M \wedge V$ where M and V are specified below:

1. **Events:**
   The events act as signals between the user and stable storage.
   $E : \{\mathit{r(sn)}{\downarrow}, \mathit{s(c)}{\uparrow}, \mathit{w(sn,d)}{\downarrow} \colon \mathit{sn} \in \mathit{SN} \wedge c, d \in \mathit{INF}\}$
   where SN is the set of sector numbers: $[1, .., \mathit{SNMAX}]$ and INF is the set of information items that could be stored and retrieved by stable storage; it will not be further specified.
   r(sn): a request to read sector *sn*; *s(c)*: the response to the previous request where c are the contents of sector *sn*; *w(sn,d)*: write information item d onto sector *sn*.
2. **States:**
   $Q : (\{\mathbf{SS}[i] : i \in \mathit{SN}\} \rightarrow \mathit{INF}) \times (\mathbf{s} : \{\mathit{idle}, \mathit{retr}, \mathit{error}\}) \times (\mathbf{bf} : \mathit{SN})$
   $\mathbf{SS}[i] = v_i$: sector i contains information item v_i; **s** stands for **s**table **s**torage **s**tate, i.e., describes the current state of stable storage as follows: $\mathbf{s} = \mathit{idle}$: stable storage is waiting for an event to occur, $\mathbf{s} = \mathit{retr}$: the user has requested some contents of stable storage and stable storage is *retr*ieving them, $\mathbf{s} = \mathit{error}$: the user has requested the contents of a location before stable storage has responded to a previous request. Variable **bf** is used to store the current sector from which the user has requested the contents.
3. **Initial States:**
   $\mathit{IQ} \stackrel{\text{def}}{=} \forall i \in \mathit{SN}.\mathbf{SS}[i] = \mathit{dflt} \wedge \mathbf{s} = \mathit{idle}$
   Where $\mathit{dflt} \in \mathit{INF}$ is some default information item.
   Stable storage is waiting for an event to occur and each location (sector) of stable storage contains some default information item.
4. **Transitions:**
   Figure 2 illustrates the transitions of stable storage, in a notion reminiscent of Harel's Statecharts [6]. Note: in this figure transitions of the form e/a occur, where e is an event and a an action. For example $\mathit{r(sn)}{\downarrow}\ /\mathbf{bf}' = \mathit{sn}$. The meaning of this transition is that when event *r(sn)* occurs variable **bf** gets as new value *sn*. The $'$ is the "immediately after" temporal operator of DTL. So in state $\mathbf{s} = \mathit{retr}$ the value of **bf** equals *sn*.
5. **Validity conditions:**
   As said before, the validity must express the fact that the user and the stable storage medium must communicate correctly; for instance, it is not allowed that the user generates an *r(sn)* event or *w(sn,d)* before the stable storage medium has replied with an *s(c)* event to a previous *r(sn)* event. This is because our version of stable storage can only handle one request at a time.
   $V \stackrel{\text{def}}{=} R \rightarrow G$
   $R \stackrel{\text{def}}{=} \Box(\mathbf{s} = \mathit{retr} \rightarrow (\mathbf{e} \neq \mathit{r(sn)} \wedge \mathbf{e} \neq \mathit{w(sn,d)}))$
   This expresses that the user will never generate the request and write events when the stable storage medium is busy retrieving the contents of some

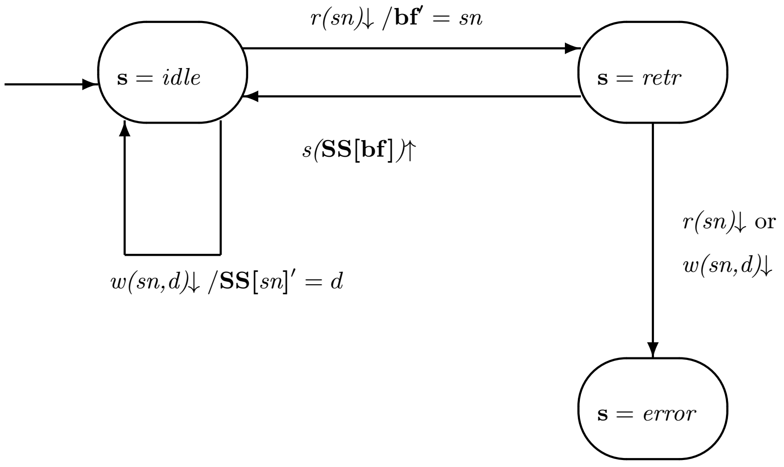

Fig. 2. Transitions of Stable Storage

sector. So R expresses that the user should live with the fact that the stable storage medium can only handle one request at a time.
$G \overset{\text{def}}{=} \Box(\mathbf{e} = r(sn) \rightarrow \Diamond\mathbf{e} = s(\mathbf{SS}[sn]))$
When the user requests the contents from some sector, the stable storage medium should guarantee that the user eventually gets these contents. Thus, the validity condition expresses the property that the stable storage never enters into the *error* state.

3.5 Application: Second step

Introduction In this step, which is the first stage in our task to develop a fault tolerant system, we give the specification of a physical disk. This specification is a first approximation to our fault tolerant system, i.e., it acts as the undecorated basic layer of our desired implementation. In this specification we must specify which are the anticipated faults of our system, i.e., we have to specify which faults are the focus of our interest that could affect a physical disk. These faults are represented in our formalism as events.

Specification We must specify a physical disk, the anticipated faults and their impact on the physical disk. We take as anticipated faults the following ones (cf. [10]):

– Damages of the disk surface causing corruption of the contents of a physical sector.
– Disk control faults causing the contents of a particular physical sector to be read from or written to a wrong location.

These faults are described using two events: the *dam* event, standing for the fault expressing damage to the disk surface and the *csf* event standing for a fault in the disk control system.

In analogy to the specification of stable storage, the user signals with an *rP(psn)* event that it wants to read the contents of physical sector *psn*. The physical disk then signals with an *sP(c)* event that it has retrieved the contents from this location. With an *wP(psn,d)* event the user signals that the physical disk has to store information item d onto sector *psn*. Because stable storage can handle only one request at a time, we take a physical disk with the same feature. The formal specification $I \overset{\text{def}}{=} M_{PD} \wedge V_{PD}$ where M_{PD} and V_{PD} are defined below:

1. **Events:**
   $E_{PD} = \{rP(psn){\downarrow}, sP(c){\uparrow}, wP(psn,d){\downarrow}, csf(psn){\downarrow}, dam(psn){\downarrow}:$
   $\quad psn \in PSN \wedge c, d \in PHY\}$
   where *PSN* is the set of *P*hysical *S*ector *N*umbers : $[1, \ldots, PSNMAX]$ and *PHY* the set of information items that can be stored and retrieved by the *PHY*sical disk.
   rP(psn): the request to read the contents of physical sector *psn*; *sP(c)*: the response to the previous request where c are the contents from phy. sector *psn*; *wP(psn,d)*: write information item d onto physical sector *psn*.
2. **States:**
   We must somehow model how the anticipated faults can affect the disk. Therefore we introduce array **C** to model the effect of an *csf* event. **C** is a mapping from sector numbers to sector numbers. The initial value of **C** is the identity mapping. When a *csf* event occurs a sector number will be remapped to another sector number. So the physical disk will retrieve the contents from the location mapped into by **C**. To describe the effect of an *dam* event we introduce array **P** which is a mapping from sector numbers to set of information items that can be stored on a sector plus a special information item *cd* indicating that the sector contains *c*orrupted *d*ata. As in the specification of stable storage we also need a variable **pds** indicating the current **s**tate of the **p**hysical **d**isk and a variable **pbf** for storing the current **p**hysical sector number. More formally:
   $Q_{PD} : (\{\mathbf{P}[i] : i \in PSN\} \rightarrow PHY \cup \{cd\}) \times (\{\mathbf{C}[i] : i \in PSN\} \rightarrow PSN)$
   $\quad \times(\mathbf{pds} : \{idle, retr, error\}) \times (\mathbf{pbf} : PSN)$
   $\mathbf{C}[i] = j$: the control system maps sector i to sector j. $\mathbf{P}[i] = v_i$: physical sector i contains information item v_i. $\mathbf{pds} = idle$: the physical disk is waiting for an event to occur. $\mathbf{pds} = retr$: the user has requested some contents of the physical disk and the physical disk is currently *retr*ieving them. $\mathbf{pds} = error$: the user has requested the contents of a location before the physical disk has responded to a previous request.
3. **Initial States:**
   $IQ_{PD} \overset{\text{def}}{=} \forall i \in PSN.(\mathbf{P}[i] = dflt \wedge \mathbf{C}[i] = i) \wedge \mathbf{pds} = idle$
   All sectors contain the default data item *dflt* and the control system has not been affected by control system fault.

4. **Transitions:**
Figure 3 illustrates the transitions of the physical disk.

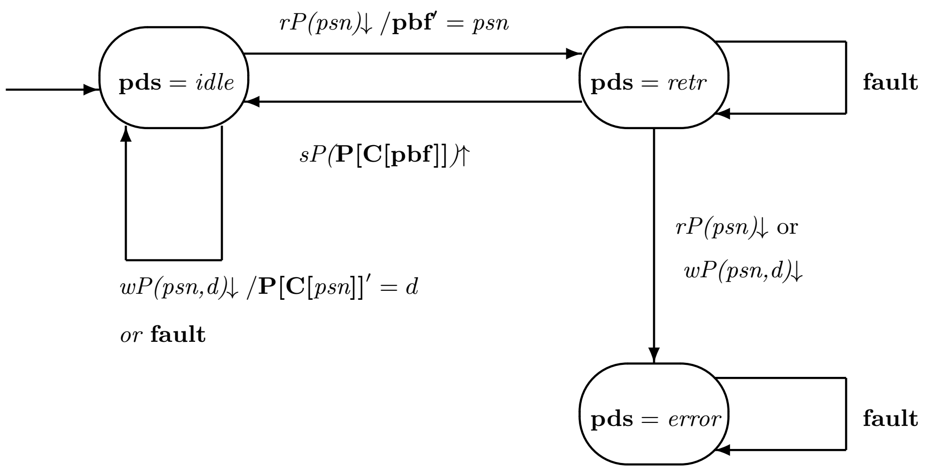
$rP(psn)\downarrow$ /$\mathbf{pbf}' = psn$
$\mathbf{pds} = idle$
$\mathbf{pds} = retr$
fault
$sP(\mathbf{P}[\mathbf{C}[\mathbf{pbf}]])\uparrow$
$rP(psn)\downarrow$ or
$wP(psn,d)\downarrow$
$wP(psn,d)\downarrow$ /$\mathbf{P}[\mathbf{C}[psn]]' = d$
or **fault**
$\mathbf{pds} = error$
fault

Fig. 3. Transitions of the physical disk, where
fault stands for $(csf(psn)\downarrow$ /$\mathbf{C}'[psn] = j)$ *or* $(dam(psn)\downarrow$ /$\mathbf{P}'[psn] = cd)$, for some $j \in PSN$.

5. **Validity conditions:**
$V_{PD} \stackrel{\text{def}}{=} R_{PD} \rightarrow G_{PD}$
$R_{PD} \stackrel{\text{def}}{=} \Box(pds = retr \rightarrow (\mathbf{e} \neq rP(psn) \wedge \mathbf{e} \neq wP(psn,d)))$
The rely condition R_{PD} expresses that the user never generates the request and write events before the physical disk has responded to a previous request.
$G_{PD} \stackrel{\text{def}}{=} \Box(\mathbf{e} = rP(psn) \rightarrow \Diamond\mathbf{e} = sP(\mathbf{P}[\mathbf{C}[psn]]))$
The physical disk then guarantees that the user will get eventually a response to a request.

Correctness As seen in sect. 3.2 one must define the condition NFO that expresses that the anticipated faults never occur. NFO is defined as follow:
$NFO \stackrel{\text{def}}{=} \Box(\mathbf{e} \neq dam(psn) \wedge \mathbf{e} \neq csf(psn))$
For correctness of relative refinement one has to prove $[I \wedge NFO]_\delta \rightarrow [A]_\alpha$ where $\alpha : E \times E_{PD} \rightarrow E$ and $\delta : E \times E_{PD} \rightarrow E_{PD}$. The proof of this is given in the full paper and goes intuitively as indicated in section 2.5, i.e., NFO and $\Box TR_{PD}$ (the DTL formula corresponding to fig. 3) are taken together resulting in a new state-transition relation which is the same as fig. 3 except that the **fault** transitions are removed. Comparing this new state-transition relation with that of TR (the DTL formula corresponding to fig. 2) shows that these are the same except that states and events have another name.

3.6 Application: Third Step

Introduction In this step, the second stage in our development of the fault tolerant system, we specify the layer that detects the faults that we assumed could affect the physical disk (the anticipated faults); this layer is added in bottom-up fashion to the physical disk specified in section 3.5. The detection layer acts as a sort of "interface" between the user and the physical disk. It stops the machine when an anticipated fault is detected by the detection mechanism, i.e., the whole system (detection layer plus physical disk) stops when such a fault occurs. This is called a *fail-stop implementation* [7]. It also informs the user which kind of anticipated fault has occurred. As seen above, there are two classes of anticipated faults. Consequently there are two kinds of detection mechanisms. The first one checks whether the contents read from the physical disk are corrupted, i.e., detects errors due to damage of the disk surface. This is done with a cyclic redundancy mechanism [7]. The second one checks whether the contents read from the physical disk originate from the right location. This is done using an address checking mechanism [7] which encodes the location of the contents of the physical disk inside these contents.

Specification The detection layer consists of three parts: the first part checks whether the data retrieved from the physical disk is affected by a corrupt data fault (the fault that damages the disk surface), using a cyclic redundancy check (CRC) mechanism [7]. The second part checks whether the data retrieved from the physical disk is from the correct physical location, i.e., whether it is affected by a disk control system fault, using an address checking (ADR) mechanism [7]. The third part prevents further access by the user of the physical disk when one of these two mechanisms detects a fault by having the detection layer act as "interface" between the user and the physical disk; the detection layer refuses to communicate with the user and the physical disk when such faults occur. Furthermore this part gives a message to inform the user which anticipated fault has occurred.

The protocol of this interface between user and physical disk is as follows. If the user wants to read the contents of some physical sector it generates an *rD* event for the detection disk layer (the interface). This layer generates after receipt of this event an *rP* event. That event is received by the physical disk. The physical disk then signals with an *sP* event to the detection layer that it has retrieved the contents of that physical sector, upon which the detection layer signals with an *sL* event to the user that the contents are retrieved. The same holds mut. mut. for the writing of data onto the physical disk. The events will be explained below. We also introduce logical sector numbers; these logical sector numbers will be needed in the fourth step, but we give them already here. In the fourth step, when the detection layer detects that data from a physical sector number is affected by a disk surface damage fault, the correct data will be written to another physical sector number. In order to retrieve these contents from this new location *logical* sector numbers are introduced. When contents are stored at a new physical sector the logical sector number will be pointing to this new sector. So actually the data are retrieved from their logical sector (number). In this section however, the mapping between the logical sector numbers and the physical sector numbers will be the identity mapping because they are not needed here. The specification of detection layer is $DL_{fs} \overset{\text{def}}{=} M_{DL} \wedge V_{DL}$ where M_{DL} and V_{DL} are defined below:

1. **Events:**
   $E_{DL} \overset{\text{def}}{=} \{rD(lsn)\!\downarrow, sD(dc)\!\uparrow, wD(lsn,d_1)\!\downarrow, rP(psn)\!\uparrow, sP(c)\!\downarrow, wP(psn,d)\!\downarrow$
   $: lsn \in LSN \wedge psn \in PSN \wedge dc, d_1 \in LOG \wedge c, d \in PHY\}$
   where LSN is the set of logical sector numbers: $([1, ..., LSNMAX])$, LOG the set of data items that the user wants to store on, or retrieve from, the physical disk and PHY the set information items that can be stored on or retrieved from the physical disk. (Note: an item from PHY is an crc-encoded and address-encoded item of LOG.)
   rD(lsn): the request from the user to read logical sector *lsn*. *sD(c)*: the response of the detection layer to the previous request where c are the crc-decoded and address-decoded contents of logical sector *lsn*. *wD(lsn,d)*: (user signal) write information item d onto logical sector *lsn*. *rP(psn)*: the request from the detection layer to read physical sector *psn*. *sP(c)*: the response of the physical disk to the previous request where c is the crc-encoded and address-encoded contents of physical sector *psn*. *wP(psn,d)*: (detection layer signal) write information item d onto physical sector *psn*.
2. **States:**
   $Q_{DL} : (\mathbf{dls} \rightarrow \{idle, retrD, retrP, sent, stop1, stop, error, write\})$
   $\times(\mathbf{pbf} \rightarrow PHY) \times (\mathbf{msg} \rightarrow \{DSDF, DCSF\} \bigcup LOG)$
   $\times(\mathbf{lbf} \rightarrow LSN) \times (\mathbf{LS} : LSN \rightarrow PSN)$
   dls stands for **d**etection **l**ayer **s**tate, i.e., the current state of the detection layer. **dls** = *idle*: the detection layer is waiting for an event from the user to occur. **dls** = *retrD*: the user has requested some contents and the *D*etection layer is *retr*ieving them. **dls** = *retrP*: the detection layer has requested some contents and the *P*hysical disk is retrieving them. **dls** = *sent*: the physical disk has *sent* the requested contents and these contents are correct according to the detection layer, i.e., not corrupted by the anticipated faults. **dls** = *stop1*: the physical disk has sent the requested contents and these contents are not correct according to the detection layer, i.e., at least one fault has affected the contents. **dls** = *stop*: the detection layer will not repond to any request from the user or physical disk, i.e., it *stop*s. **dls** = *error*: the user has requested the contents of a location before the detection layer has responded to a previous request, or the physical disk has responded to a request not given by the detection layer. **dls** = *write*: the user has signaled that some data has to be *writ*ten onto the physical disk and the detection layer is taking care of that. **msg** is a variable indicating the contents which the user has requested, or, if the contents are not correct, which anticipated fault has occurred, i.e., *DSDF* upon a *D*isk *S*urface *D*amage *F*ault and *DCSF* upon a *D*isk *C*ontrol *S*ystem *F*ault. **pbf** is a variable to store the crc-encoded and address encoded contents of a physical sector. **lbf** is a variable to store the

logical sector number from which the user has requested the contents. **LS** is a mapping from logical sector numbers to physical sector numbers.

3. **Initial states:**
   In the initial state the detection layer is ready to receive events from the user and the mapping between the logical sector numbers and the physical sector numbers is the identity mapping.
   $IQ_{DL} \overset{\text{def}}{=} \mathbf{lds} = \mathit{idle} \wedge \bigwedge_{i \in PSN} \mathbf{LS}[i] = i$
4. **Transitions:** Are described in fig. 4. In this figure transitions of the form $e[c]/a$ occur. The meaning of this transition is that when event e occurs and condition c holds then action a will be performed. The intuitive meaning of **ok** is that contents just retrieved from the physical disk are not affected by the two kinds of fault, of **error** is that contents are affected. The **uwrite** transition is the signal of the user towards the detection layer that data has to be written. The **dwrite** transition is the signal from the detection layer towards the physical disk that data has to be written.

   To describe the two detecting mechanisms as transitions, the set $ADR \overset{\text{def}}{=} LOG \times PSN$ and the following functions are needed (see [7] for more information about this CRC-coding):
   $CC : PHY \rightarrow Bool$ (CrcCheck): used to check whether data from the physical disk is damaged by a disk surface fault. $CD : PHY \rightarrow ADR$ (CrcDecode): used to decode the CRC-coded physical data into ADR format. $CE : ADR \rightarrow PHY$ (CrcEncode): used to encode data in ADR format into physical CRC format. $AC : ADR \times PSN \rightarrow Bool$ (AdrCheck): used to check whether data is read from the correct physical location. $AD : ADR \rightarrow LOG$ (AdrDecode): used to decode data in ADR format into LOG format. $AE : LSN \times LOG \rightarrow ADR$ (AdrEncode): used to encode a physical sector number and a information item given by the user into ADR format.
5. **Validity conditions:**
   $V_{DL} \overset{\text{def}}{=} R_{DL} \rightarrow G_{DL}$
   $R_{DL} \overset{\text{def}}{=} \Box(\mathbf{e} = rP(psn) \rightarrow \Diamond \mathbf{e} = sP(\mathbf{P}[psn]))$
   The rely condition R_{DL} expresses the fact that when the detection layer requests the contents of a physical disk sector it eventually gets these contents.
   $G_{DL} \overset{\text{def}}{=} \Box(\mathbf{e} = rD(lsn) \rightarrow \Diamond(\mathbf{e} = sD(AD(CD(\mathbf{P}[\mathbf{LS}[lsn]])))))$
   The guarantee condition G_{DL} expresses the fact that, when the user requests the contents from a logical sector, the detection layer eventually sends the contents of the physical sector to which this logical sector number is mapped.

Correctness As seen in sect. 3.2 one needs the condition NED that expresses that no errors are detected. This condition is defined as follows:
$NED \overset{\text{def}}{=} \Box(\bigwedge_{i \in PSN}(CC(\mathbf{P}[i]) \leftrightarrow \mathbf{P[i]} \neq cd) \wedge (AC(CD(\mathbf{P}[i]), i)) \leftrightarrow \mathbf{C}[i] = i)$.
For correctness one only have to prove that $NFO \rightarrow NED$ holds.

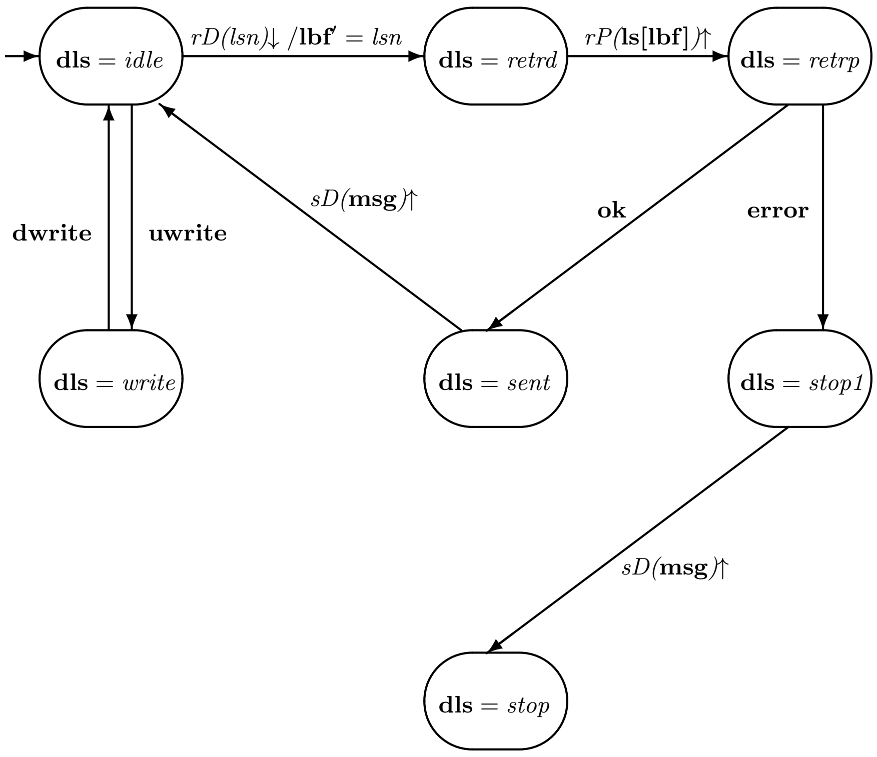

Fig. 4. Transitions of the detection layer where
ok stands for $sP(c)\downarrow\ [CC(c) \wedge \neg AC(CD(c), \mathbf{LS}[\mathbf{lbf}]))]/\mathbf{msg}' = AD(CD(c)))$,
error stands for $sP(c)\downarrow\ [CC(c) \wedge AC(CD(c), \mathbf{LS}[\mathbf{lbf}]))]/\mathbf{msg}' = DCSF$ or
$sP(c)\downarrow\ [\neg CC(c)]/\mathbf{msg}' = DSDF$,
uwrite stands for $wD(lsn,d1)\downarrow\ /\mathbf{pbf}', \mathbf{lbf}' = CE(AE(lsn,d1)),lsn$ and
dwrite stands for $wP(\mathbf{LS}[\mathbf{lbf}], \mathbf{pbf})\uparrow$

3.7 Fourth Step: Error Recovery Layer

Introduction In this step we specify the error recovery layer. This is the layer that tries to correct the errors detected by the detection layer. The technique we use for error recovery is that of the *mirrored disk concept* [7]. This mirrored disk concept is as follows: instead of one physical disk and corresponding detection layer, we maintain N physical disks with identical contents and N corresponding detection layers ($N > 1$). In case some information can no longer be retrieved from one disk, the information is still available on another one. The user requests some contents from the error recovery layer. The error recovery layer selects a disk from which it can retrieve these contents. Then it requests these contents from the corresponding detection layer of that disk. The detection layer requests the contents from the physical disk and checks whether the contents are correct. The detection layer signals if the contents are correct and, if not, it will signal which error it has detected. If the contents are correct the error recovery layer will send them to the user and is then ready for new requests from the user. As seen before the detection layer can detect two kinds of errors. The error recovery layer will react as follows on these errors:

ad (1) First, the error recovery layer selects another disk from which it can retrieve the requested contents and, when the corresponding detection layer signals that the contents are correct, the error recovery layer writes these contents to another location of the affected disk. In order to retrieve these contents from this new location *logical* locations are introduced. When contents are stored at a new physical location the logical location will be pointing to this new location. So actually the data are retrieved from their logical location. Subsequently, the error recovery layer sends the contents to the user and is ready to receive new requests from the user. When the detection-layer of the second disk also reports an error, the error recovery layer will react as described in *ad(1)* and *ad(2)* depending on the kind of error detected.

ad (2) First, the error recovery layer disables the faulty disk and next it selects another disk from which it can retrieve the requested contents; when the corresponding detection layer signals that the contents are correct, the error recovery layer will pass them on to the user. When the detection-layer of the second disk also reports an error the error recovery layer reacts as described in *ad(1)* and *ad(2)* depending on the kind of error detected.

This error recovery process only works if we make the following assumptions:
(I) In order to store the contents on a new physical location enough spare locations should be available on an affected disk.
(II) Furthermore, the following must always hold in order to recover the *ad(1)*-type of error on a disk or to retrieve the contents from a logical location: for all logical locations there exists at least one non-disabled physical disk that has correct data stored on that logical location. This condition guarantees that, each logical location contains correct data (on which disk we don't know, but it is a non-disabled one and it is not a disk whose type 1 error has to be repaired).

Specification The error recovery layer acts as interface between the user and the N detection layers of the N physical disks. The user requests with an *rR* event the contents of some logical sector. Upon receipt of this event, the error recovery layer requests these contents with an *rD* event from one of the non-disabled detection layers. This detection layer responds with an *sD* event. As seen in the third step there are three possibilities:

1. If this *sD* event is a message saying that the (to this detection layer corresponding) physical disk has been affected by a disk control system fault then this detection layer will be disabled and the error-recovery layer will send an *rD* event to another non-disabled detection layer.
2. If the *rD* event is a message that the (to this detection layer corresponding) physical disk has been affected by a disk surface damage fault then the error recovery layer requests the contents with an *rD* to another non-disabled detection layer until it finds a detection layer that responds with the correct contents. Then the error recovery layer can "repair" the physical disk that has been affected by a disk surface damage fault by generating an *wR* write event with the correct data to the same logical sector number of the corresponding detection layer of that physical disk. The design decision we make is that the detection layer has to find the spare physical sector to which these contents can be written. After that, the error recovery layer responds with an *sE* event to the request of the user.
3. If *rD* event contains normal data the error recovery layer will respond with an *sE* event that it has retrieved the requested contents.

The user signals with an *wE* event to the error recovery layer that some data have to be written into a logical sector. The error recovery layer generates an *wD* write event with these contents to all non-disabled detection layers to ensure that the corresponding physical disks have identical contents. These physical disks have of course stored these contents not on the same physical sector but on the same logical sector.

The error recovery layer will not be specified in detail because it is rather lengthy. Only the validity condition will be given.

Validity Conditions:
$V_{ER} \overset{\text{def}}{=} R_{ER} \rightarrow G_{ER}$
$R_{ER} \overset{\text{def}}{=} \Box(\mathbf{e} = rD_i(lsn) \rightarrow \Diamond\mathbf{e} = sD_i(c))$
The rely condition R_{ER} expresses that when the error recovery layer requests some contents from a detection layer then the detection layer eventually responds to that request.
$G_{ER} \overset{\text{def}}{=} \Box(\mathbf{e} = rE(lsn) \rightarrow \Diamond\mathbf{e} = sE(\mathbf{P}[\mathbf{LS}_i[lsn]])))$
In that case the error recovery layer can guarantee that the user will get the requested contents.

Correctness As seen above the error recovery process is only correct under certain restriction on the occurrence of faults. Computations in which the error

recovery process doesn't work are: computations in which a physical disk has an error of type 1 and has no spare locations to store the correct contents, or in which a disk has an type 1 error and all the other disks are disabled. Thus condition REC is as follows
$REC \overset{\text{def}}{=} \Box(\forall lsn \in LSN. \exists i \in \mathbf{Ena}.CC_i(\mathbf{P}[\mathbf{LS}[lsn]]) \wedge AC(CD(\mathbf{P}[\mathbf{LS}[lsn]]), lsn)$
This expresses that for all logical sector numbers there exists a non-disabled disk that has correct data stored on that logical sector.
The condition that there are enough spare locations will be included in the condition NED of the detection layers because we have made the design decision that the detection layer is responsible for finding the spare sectors.
For correctness of the final implementation one must prove, as seen in sect. 3.2, the following assumptions:
$(NFO \leftrightarrow NED$: perfect detection,
$(\neg NED \leftrightarrow REC)$: perfect recovery and
step (1): implementation I is a relative refinement of A.

4 Conclusion

In this paper we have shown that it is possible to formally specify the development of a fault tolerant system. We have used Stark's formalism in a special way in order to achieve this. The part originally intended to specify liveness properties is used for deletion of faulty and undesirable computations. This also enables us to prove correctness of an implementation that couldn't be proven correct otherwise using standard techniques, because it is impossible to prove that an implementation with faulty and undesirable computations can implement in some sense a specification that specifies only good and desirable computations.

Acknowledgements

I would like to thank Henk Schepers and Ruurd Kuiper; Amir Pnueli and all other anonymous referees for their comments.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. In *Third annual symposium on Logic in Computer Science*, pages 165–175, July 1988.
2. B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.
3. A. Cau, R. Kuiper, and W.-P. de Roever. Formalising Dijkstra's development strategy within Stark's formalism. In R. C. Shaw C. B. Jones and Tim Denvir, editors, *Proc. 5th. BCS-FACS Refinement Workshop*, 1992.
4. E. Diepstraten and R. Kuiper. Abadi & Lamport and Stark: towards a proof theory for stuttering, dense domains and refinements mappings. In *LNCS 430:Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, pages 208–238. Springer-Verlag, 1990.

5. E.W. Dijkstra. A tutorial on the split binary semaphore, 1979. EWD 703.
6. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
7. P.A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems.* Springer-Verlag, second, revised edition, 1990.
8. S. Lee, S. Gerhart, and W.-P. de Roever. The evolution of list-copying algorithms and the need for structured program verification. In *Proc. of 6th POPL*, 1979.
9. P.R.H. Place, W.G. Wood, and M. Tudball. Survey of formal specification techniques for reactive systems. Technical Report, 1990.
10. H. Schepers. Terminology and Paradigms for Fault Tolerance. Computing Science Notes 91/08 of the Department of Mathematics and Computing Science Eindhoven University of Technology, 1991.
11. E.W. Stark. *Foundations of a Theory of Specification for Distributed Systems.* PhD thesis, Massachusetts Inst. of Technology, 1984. Available as Report No. MIT/LCS/TR-342.
12. E.W. Stark. A Proof Technique for Rely/Guarantee Properties. In *LNCS 206: Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 369–391. Springer-Verlag, 1985.