Presented at the Twenty-Third Annual International Symposium on Fault-Tolerant Computing, Toulouse, France, June 22-24, 1993.

Specifying Fault Tolerance within Stark's Formalism[*]

——improved version——

Antonio Cau Willem-Paul de Roever

Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik[†]
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 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. (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 decompose 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 and contain inconsistencies [10]. Secondly, writing a correct specification is a process whose difficulty is comparable with 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 [9]). Thirdly, even an incorrect refinement step may be useful in the sense that from such a 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, see [6]. An formalization of the last has been given in [3].

In the present paper we present a formal development strategy for deriving a correct refinement step using incorrect intermediate stages and its application to a fault tolerant system. The formal strategy is as follows: one starts with an implementation for a specified fault tolerant system containing some faults, i.e., the refinement step is incorrect because of these faults. In the next step we try to detect these faults, i.e., we construct a layer upon the previous implementation that on detection of an error caused by a fault, stops that implementation, i.e., a fail-stop implementation. The second implementation is better than the previous one because now at least the implementation stops on detection of an error caused by a fault, but it is still incorrect. In the third approximation we recover these faults, i.e., we don't stop anymore upon detection of an error but merely recover the error detected by executing some special program that neutralizes the damages caused by a fault. The third implementation is correct under the assumption that certain conditions are fulfilled.

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 paper is structured as follows: Section 2 contains a a short introduction of Stark's formalism (see [2] for a more detailed introduction). Section 3 contains the formal development of a fault tolerant system for stable storage [4, 11]. Section 4 contains a conclusion.

[*]Partially supported by ESPRIT project 6021 (REACT)
[†]E-Mail:{ac, wpr}@informatik.uni-kiel.d400.de

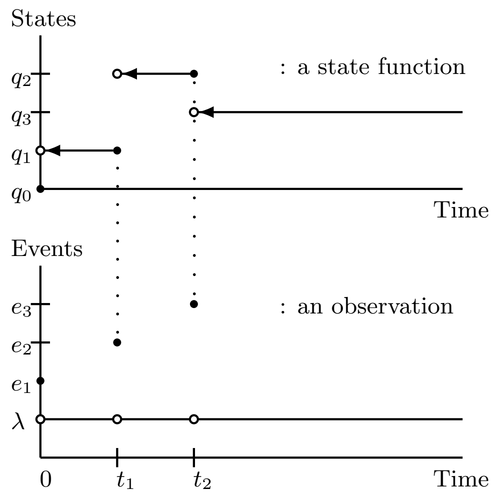

Figure 1: Following computation is illustrated: the initial state is q_0, on the occurrence of event e_1 the state changes in q_1. Between time 0 and t_1 only the uninteresting stuttering event λ occurs, so the state doesn't change until the next interesting event e_2 occurs, and so on.

2 Stark's Formalism

In this section we present Stark's Dense time Temporal Logic DTL for describing correct refinement steps ([12, 13] are not easily accessible) and our use of it to describe incorrect refinement steps. In sect. 2.1 the semantic notion of a computation is introduced, needed for the semantics of DTL: it is a merge of an observation (when and which events occur) and a state function (when and which state changes occur) see fig. 1. It is well know [1] that two kinds of properties can be expressed about these computations: safety (local) and liveness (global) properties. We use the first one to describe the program that is developed till now and the second one to describe the desirable computations of that program (so undesirable computations are removed). Furthermore we introduce the syntax of DTL (based on [5]) in order to express these properties. In Stark's formalism the stuttering problem is solved in an easy way and it is possible to express *within this logic* what a correct refinement step is. In sect. 2.2 the semantic notion of a correct refinement step is defined. Three levels of set of allowed computations (intersection of our safety and liveness properties) are distinguished in order to express this refinement: the abstract level, the concrete level and the composite level that relates these two previous levels. In sect. 2.3 this semantic refinement is expressed in DTL by two verification conditions. Furthermore it is shown how "incorrect" refinement steps can be expressed in DTL.

2.1 DTL: semantics and syntax

In [12] a method for specifying reactive systems is introduced. Such systems are assumed to be composed of one or more modules. A module specifies the set of allowed computations, i.e., it consists of two parts: (1) the machine part for describing the safety (local) properties and (2) the validity part for describing the liveness (global, in our case the desired computations of the machine) properties.

The formal definition of a *machine* $M = (E, Q, I, T)$ is as follows:

• E: is the *interface* of M, i.e., the set of possible events. An *event* is an observable instantaneous occurrence during the operation of a machine, that can be generated by that machine 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 machine. It is generated by the environment of the machine and is of interest at our level of abstraction. Events labeled with$\downarrow$ are input events, and events labeled with$\uparrow$ are output events, and events without an arrow are internal events,

• Q: 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, with $Var \cap F = \emptyset$, to the set of values *Val*, i.e., $Q : (Var \cup F) \to Val$. Note: the "normal" variables will be bold faced in order to distinguish them from freeze variables.

• I: a non-empty subset of Q, the set of initial states,

• T: the state-transition relation, $T \subseteq Q \times E \times Q$, such that for all $q \in Q$ the stuttering step $\langle q, \lambda_E, q\rangle \in T$ under the condition that for all $r, q \in Q$ $\langle q, \lambda_E, r\rangle \in T$ iff $r = q$, i.e., λ-events can't change a state. Furthermore M is input-cooperative, i.e., in every state of M any input event can be received.

An *observation* over interface E is a function *obs* from $[0, \infty)$ to E, such that $obs(t) \neq \lambda$ for at most finitely many $t \in [0, \infty)$ in each bounded interval, which means that in a bounded interval only a finite number of interesting events can occur (the finite variability condition). Fig. 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 λ event occurs.

A *state function* over a set of states Q is a function *sf* from $[0, \infty)$ to Q such that for all $t \in [0, \infty)$, there exists $\varepsilon_t > 0$ such that *sf* is constant on intervals

$(t - \varepsilon_t, t] \cap [0, \infty)$ and $(t, t + \varepsilon_t]$. We write $sf(t^{\leftarrow\bullet})$ for the value of the state just before and at time t (the first interval) and write $sf(t^{\circ\rightarrow})$ for the value of sf just after time t (the second interval). Fig. 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 $h = \langle obs, sf \rangle$, where *obs* is an observation over E, and *sf* 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 the state just before and including t, the observation at t, and 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* occurring at time t in h which is defined as follows: $step(t) = \langle sf(t^{\leftarrow\bullet}), obs(t), sf(t^{\circ\rightarrow}) \rangle$. Let $Hist(E, Q)$ denote the set of all histories over interface E and state set Q. A *computation of a machine* M is a history h st: $sf(0) \in I$ and $step(t) \in T$ for all $t \in [0, \infty)$. $Comp(M)$ denotes the set of all computations of M, and Rea_M the set of reachable states of M.

The *validity part* of a module is just a set of computations.

Till now we have only a semantic notion of machine and validity. We now present the syntax for expressing these two parts.

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 interface E;
special symbol $\mathbf{e}$;
event term $\mathbf{e} = f$ where f is an event;
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 $\Box$, $\Diamond$

Examples:
$(\mathbf{x} = 0 \wedge \mathbf{e} = d_0) \rightarrow \mathbf{x}' = 1$ (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. Let history h be defined as follow $h \doteq \langle obs, sf \rangle$. Let $h^{(\tau)}$ denote the history $\lambda t.h(t + \tau)$.
For all freeze variables $v \in F$, $v(h) = sf(0)(v)$. (Note: because v is a freeze variable the value doesn't change in a history, at time 0 it is initialized.)
For all variables $v \in Var$, $v(h) = sf(0)(v)$.
For all variables $v \in Var$, $v'(h) = sf(0^{\circ\rightarrow})(v)$.
For $\mathbf{e}$, $\mathbf{e}(h) = obs(0)$.
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 $sf_0(0)$ differing from $sf(0)$ only in the value assigned to freeze variable x such that $\langle obs, sf_0 \rangle \models \varphi$;
$h \models \Diamond\varphi$; if there exists an $t \in [0, \infty)$ st $h^{(t)} \models \varphi$;
$h \models \Box\varphi$; 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 $En_M(e)$ is the precondition of the transition that corresponds with that event. The local properties of a module Z can now be expressed by formula $I_Z \wedge \Box T_Z$. Thus Comp$(M_Z) \doteq \{h \in Hist(E_Z, Q_Z) \mid h \models I_Z \wedge \Box T_Z\}$. 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: $\{h \in \text{Comp}(M_Z) \mid h \models V_Z\}$, and is described by DTL formula $I_Z \wedge \Box T_Z \wedge V_Z$.

2.2 Semantical refinement

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 interface of the abstract module and that of 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 both interface E of the composite module with interface A of the abstract module, and interface E of the composite module with each of the interfaces F_i of the concrete modules. Hence it characterizes a refinement relation between an abstract module and a set of concrete modules. It defines which event on the abstract level is implemented by events on the concrete level.

Define the composite interface as $E \doteq A \times \prod_{j\in J} F_j$ and $\lambda_E \doteq \langle \lambda_A, \langle \lambda_j \rangle_{j\in J} \rangle$. The interconnection $\mathcal{I}$ is a pair $\langle \alpha, \langle \delta_j \rangle_{j\in J} \rangle$ where:
• α denotes a function from the composite interface E to the abstract interface A such that $\alpha(\lambda_E) = \lambda_A$ holds; α is called an *abstraction* function.
• δ_j denotes a function from the composite interface E to the concrete interface F_j such that $\delta_j(\lambda_E) = \lambda_j$ holds; δ_j is called a *decomposition* function.

So intuitively the requirement about 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) computations of the mentioned modules.

Let B_A denote the set of allowed computations of the abstract module. and B_j denote that of concrete module j. A *refinement step* is defined as a triple $\langle \mathcal{I}, B_A, \langle B_j \rangle_{j\in J} \rangle$ where $\mathcal{I}$ is the interconnection (between computations), A refinement step is *correct* if the following holds: $\bigcap_{j\in J} \delta_j^{-1}(B_j) \subseteq \alpha^{-1}(B_A)$

2.3 (In)correct refinement 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 $I_A \wedge \Box T_A$, concrete machines M_j, described by temporal formula $I_j \wedge \Box T_j$, 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 $M = (E, Q, I, T)$ as follows:
• $E \doteq A \times \prod_{j\in J} F_j$;
• $Q \doteq Q_A \times \prod_{j\in J} Q_j$, and $I \doteq I_A \wedge \bigwedge_{j\in J} I_j$;
• In order to express the state-transition relation T we must use the definitions of α and the δ_j's to transform event terms in T_A and T_j into event terms of T. Event term $\mathbf{e} = d$ in T_A is transformed into $\mathbf{e} = \alpha^{-1}(d)$ and event term $\mathbf{e} = f$ in T_j into $\mathbf{e} = \delta_j^{-1}(f)$. We introduce the following notation: $[f]_\alpha \doteq f[\alpha^{-1}(e)/e]$ for $e \in A$ and $[f]_{\delta_j} \doteq f[\delta_j^{-1}(e)/e]$ for $e \in F_j$. Then the state-transition relation T of M is defined as follows: $T \doteq \Box([T_A]_\alpha \wedge \bigwedge_{j\in J}[T_j]_{\delta_j})$.

The correctness condition of the refinement step, as seen above, is as follows:

$$\bigcap_{j\in J} \delta_j^{-1}(B_j) \subseteq \alpha^{-1}(B_A).$$

Following DTL formula expresses this condition:

$$\bigwedge_{j\in J}[I_j \wedge \Box T_j \wedge V_j]_{\delta_j} \to [I_A \wedge \Box T_A \wedge V_A]_\alpha$$

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 [1] 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 conditions:
• *maximality:* any event that can be generated by the system of concrete machines can also be performed by the abstract machine, i.e.,

$$\forall e \in E.(Rea_c \wedge \bigwedge_{j\in J} En_c(\delta_j(e))) \to En_c(\alpha(e))$$

where Rea_c is a condition that checks if a state of the composite machine is reachable, $En_c(\delta_j(e))$ is the enabling condition of event $\delta_j(e)$ of machine j, and $En_c(\alpha(e))$ 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.,

$$Comp(M_c) \models (\bigwedge_{j\in I}[V_j]_{\delta_j}) \to [V_A]_\alpha$$

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 $I_A \wedge \Box T_A \wedge V_A$, where V_A is a pure liveness condition, i.e., the machine part of the abstract specification doesn't characterize unallowed computations. Let each concrete specification be $I_j \wedge \Box T_j \wedge P_j \wedge L_j$ 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* T_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, $\bigwedge_{j\in J}[I_j \wedge \Box T_j \wedge P_j \wedge L_j]_{\delta_j} \rightarrow [I_A \wedge \Box T_A \wedge V_A]_\alpha$.
Because (1) both P_j and T_j are safety conditions, (2) the conjunction of two safety conditions is again a safety condition, and (3) P_j disallows certain transitions of T_j from being taken, $\Box T_j \wedge P_j$ can be transformed into $\Box Tnew_j$, which is also expressed as a safety condition. Note: $Tnew_j$ is a new transition relation. So we get the following:
$\bigwedge_{j\in J}[I_j \wedge \Box Tnew_j \wedge L_j]_{\delta_j} \rightarrow [I_A \wedge \Box T_A \wedge V_A]_\alpha$.
This form allows one to use Stark's two verification conditions, because $Tnew_j$ is a pure safety condition and L_j a pure liveness condition.

3 Relative refinement in fault tolerance

In this chapter we first introduce in sect. 3.1 a *general methodology* for proving fault tolerant systems correct. This general methodology uses the relative refinement concept of sect. 2.3. 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 3.2 introduces this application. In sections 3.3, 3.4, 3.5 and 3.6 the four steps of this general methodology are applied to the stable storage example [4, 11].

3.1 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. In step 2,3 and 4 of the methodology a fault-tolerant implementation is developed.

The second step of the general methodology, *identifies* the anticipated faults which can affect an implementation P. This implementation serves as first approximation to the final implementation of A. It should be clear that P is not a refinement of A because of the possible occurrences of anticipated faults. P is only a refinement when these faults do not occur, i.e., P is a *relative refinement* of A. We have seen in sect. 2.3 that this relative refinement step can expressed using ordinary refinement, i.e., $P \wedge \neg FO$ is a refinement of A where $\neg FO$ expresses that these anticipated faults never occur. So in step 2 the proof obligation is:
(1) $[P \wedge \neg FO]_{\delta_P} \rightarrow [A]_\alpha$.
Where δ_P and α are respectively the decomposition and abstraction function needed for expressing this refinement.

In the third step of our development one specifies *how these anticipated faults are detected*, i.e., one has to specify a detection layer D_{fs} for these faults. This layer is added in bottom-up fashion to the implementation P of the second step and stops upon detection of the first error, i.e., D_{fs} is a *fail-stop* implementation. So the second approximation to the final implementation consists of the parallel composition of P and D_{fs}. This approximation is clearly not a refinement because when in P a fault occurs, and D_{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 $P \wedge \neg FO$ instead of P. But then also D_{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 $P \| D_{fs}$ is a relative refinement of A can be described by the following ordinary refinement:
(2) $([(P \wedge \neg FO)]_{\delta_P} \wedge [D_{fs} \wedge \neg ED)]_{\delta_{D_{fs}}}) \rightarrow [A]_\alpha$.
Where $\delta_P, \delta_{D_{fs}}$ and α are respectively the decomposition and abstraction functions needed for expressing this refinement. Here $\neg ED$ expresses that no errors are detected.
Proof of (2): Assume left-hand side holds. Then if $ED \rightarrow FO$ (in general a mathematical theorem of cyclic redundancy coding) holds then $[P \wedge \neg FO]_{\delta_P}$ holds. Using (1) one then infers $[A]_\alpha$. $\Box$

In the fourth step of our development *one specifies the corrective action to be undertaken after detection of an error.* This means in general that one needs *redundancy*, i.e., several copies of P and D components, because when a detection layer D detects an error, the state before that error has to be recovered and that can only be done by accessing another copy of P through its corresponding detection layer D. Note that the D 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 P and D. The final implementation is then as follows: $\|_{j=1}^N (P^j \| D^j) \| R$ where R is the error recovery layer. This implementation is correct if following refinement holds:
(3) $(\bigwedge_{j=1}^N ([P^j]_{\delta_{Pj}} \wedge [D^j]_{\delta_{Dj}}) \wedge [R \wedge RC]_{\delta_R}) \rightarrow [A]_\alpha$.

Here RC is a global restriction on the kind of faults that can be recovered.
Proof of (3): Assume left-hand side holds. There are two cases (I) no faults or (II) faults occur.
(I): then left-hand side of (3) rewrites to $\bigwedge_{j=1}^{N}([P^j \wedge \neg FO]_{\delta_{P^j}} \wedge [D^j]_{\delta_{D^j}}) \wedge [R \wedge RC]_{\delta_R}$. Under the assumption that $\neg FO \rightarrow \neg ED$ holds one infers $\bigwedge_{j=1}^{N}([P^j \wedge \neg FO]_{\delta_{P^j}} \wedge [D^j \wedge \neg ED]_{\delta_{D^j}}) \wedge [R \wedge RC]_{\delta_R}$. Since R accesses the first non-faulty P component, and no error detected means no recovery, one infers $[P^1 \wedge \neg FO]_{\delta_P} \wedge [D^1_{fs} \wedge \neg ED]_{\delta_{D_{fs}}}$. Using the result of (2) one infers that $[A]_\alpha$ holds.
(II): Let P^l be the first non-faulty component that the error-recovery component R finds (we assume that such a component can always be found) and thus the first $l-1$ P components are affected by faults. Let Y denote $\bigwedge_{j=l+1}^{N}([P^j]_{\delta_{P^j}} \wedge [D^j]_{\delta_{D^j}}) \wedge [R \wedge RC]_{\delta_R}$ then left-hand side of (3) rewrites to $\bigwedge_{j=1}^{l-1}([P^j \wedge FO]_{\delta_{P^j}} \wedge [P^l \wedge \neg FO]_{\delta_{P^l}} \wedge Y$. The first $l-1$ P components can be affected by either a recoverable fault or a non-recoverable fault. Let k be the index of the component under consideration (initially $k = l-1$) and let rep be the set of repaired components (initially $rep = \{l\}$). Then use following inductive scheme:
$(*)$ Consider component P^k: In case that P^k is affected by a recoverable fault $P^k$is repaired using the correct information of P^l (we assume that this correction doesn't introduce new faults) so $rep := rep \cup \{k\}$. In case P^k is affected by a non-recoverable fault then P^k will be "disabled". Now $\bigwedge_{j=1}^{k-1}([P^j \wedge FO]_{\delta_{P^j}} \wedge \bigwedge_{j \in rep}[P^j \wedge \neg FO]_{\delta_{P^j}} \wedge Y$ holds. If $k > 1$ then $k := k-1$ and proceed as $(*)$ above else end of scheme.
At the end of the above scheme $\bigwedge_{j \in rep}[P^j \wedge \neg FO]_{\delta_{P^j}} \wedge Y$ holds. Since this is a special case of (I) we can infer that $[A]_\alpha$ holds. $\square$

This ends our exposition of the general methodology. In the next sections this methodology will be applied to a stable storage example.

3.2 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.

3.3 First step: Stable storage

In step 1, as seen in sect. 3.1, 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 $A \doteq M \wedge V$ where M and V are specified below::
1. **Events:**
$E : \{r(sn)\downarrow, s(c)\uparrow, w(sn,c)\downarrow : sn \in SN \wedge c \in DA\}$
where SN is the set of sector numbers and DA is the set of information items that could be stored and retrieved by stable storage but that 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. $w(sn,c)$: write information item c onto sector sn.
2. **States:**
$Q : (\{\mathbf{S}[i] : i \in SN\} \rightarrow DA) \times (\mathbf{s} : \{idle, retr, error\})$
$\quad \times (\mathbf{sb} : SN)$
$\mathbf{S}[i] = v_i$: sector i contains information item v_i; $\mathbf{s}$ stands for $\mathbf{s}$table storage state, i.e., describes the current state of stable storage as follows: $\mathbf{s} = idle$: 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 **sb** is used to store the current sector from which the user has requested the contents.

3. **Initial States:**
$I \doteq \forall i \in \mathit{SN}.\mathbf{S}[i] = \mathit{dflt} \wedge \mathbf{s} = \mathit{idle}$
Where $\mathit{dflt} \in \mathit{DA}$ 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 [7]. Note: in this figure transitions of the form $e[c]/a$ occur, where e is an event, c a condition and a an action. For example $\mathit{r(sn)}\Downarrow\ /\mathbf{sb}' = \mathit{sn}$ (condition c is here true). The meaning of this transition is that when event *r(sn)* occurs variable **sb** 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*.

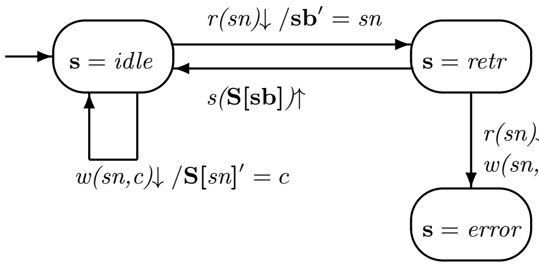

Figure 2: Transitions of Stable Storage

5. **Validity condition:**
The condition that our stable storage medium can (and eventually will) only handle one request at a time is expressed by the the validity condition $V \doteq R \rightarrow G$. Where the rely condition
$R \doteq \Box(\mathbf{s} = \mathit{retr} \rightarrow (\mathbf{e} \neq \mathit{r(sn)} \wedge \mathbf{e} \neq \mathit{w(sn,c)}))$
expresses that the user will never generate the request and write events before the stable storage medium has responded to an earlier request. And the guarantee condition
$G \doteq \Box(\mathbf{e} = \mathit{r(sn)} \rightarrow \Diamond \mathbf{e} = \mathit{s}(\mathbf{S}[\mathit{sn}]))$
expresses that when the user requests the contents from some sector, the stable storage medium guarantees that the user eventually gets these contents.

3.4 Second step: Physical disk

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. [4, 11]):
• 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 generated by the adverse environment as is done in [4]: 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(pn)* event that it wants to read the contents of physical sector *pn*. The physical disk then signals with an *sP(c)* event that it has retrieved the contents from this location. With an *wP(pn,c)* event the user signals that the physical disk has to store information item *c* onto sector *pn*. Because stable storage can handle only one request at a time, we take a physical disk with the same feature. The formal specification is $P \doteq M_P \wedge V_P$ where M_P and V_P are defined as follows:

1. **Events:**
$E_P = \{\mathit{rP(pn)}\Downarrow, \mathit{sP(c)}\Uparrow, \mathit{wP(pn,c)}\Downarrow, \mathit{csf(pn)}\Downarrow,$
$\quad \mathit{dam(pn)}\Downarrow : \mathit{pn} \in \mathit{PN} \wedge c \in \mathit{PHY}\}$
where *PN* is the set of *P*hysical sector *N*umbers and *PHY* the set of information items that can be stored and retrieved by the *PHY*sical disk.
rP(pn): the request to read the contents of physical sector *pn*. *sP(c)*: the response to the previous request where *c* are the contents from physical sector *pn*. *wP(pn,c)*: write information item *c* onto physical sector *pn*.

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 $\mathbf{C}$. To describe the effect of an *dam* event we introduce array $\mathbf{P}$ 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 $\mathbf{p}$ indicating the current state of the **p**hysical **d**isk and a variable **pb** for storing the current **p**hysical sector number. More formally:
$Q_P : (\{\mathbf{P}[i] : i \in \mathit{PN}\} \to \mathit{PHY} \cup \{cd\}) \times (\mathbf{pb} : \mathit{PN})$
$\times(\{\mathbf{C}[i] : i \in \mathit{PN}\} \to \mathit{PN}) \times (\mathbf{p} : \{\mathit{idle}, \mathit{retr}, \mathit{error}\})$
$\mathbf{C}[i] = j$: the control system maps sector i to sector j. $\mathbf{P}[i] = v_i$: physical sector i contains information item v_i. $\mathbf{p} = \mathit{idle}$: the physical disk is waiting for an event to occur. $\mathbf{p} = \mathit{retr}$: the user has requested some contents of the physical disk and the physical disk is currently *retr*ieving them. $\mathbf{p} = \mathit{error}$: the user has requested the contents of a location before the physical disk has responded to a previous request.
3. **Initial States:**
$I_P \doteq \forall i \in \mathit{PN}.(\mathbf{P}[i] = \mathit{dflt} \wedge \mathbf{C}[i] = i) \wedge \mathbf{p} = \mathit{idle}$
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.

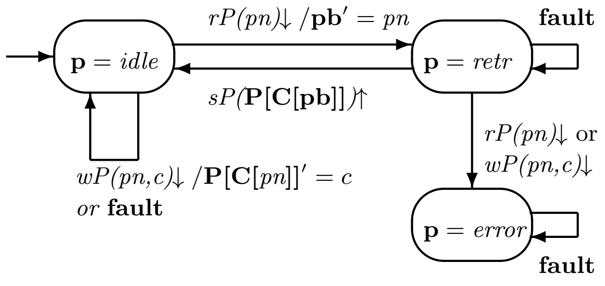
$rP(pn)\downarrow\!\!\downarrow / \mathbf{pb}' = pn$
fault
$\mathbf{p} = \mathit{idle}$
$\mathbf{p} = \mathit{retr}$
$sP(\mathbf{P}[\mathbf{C}[\mathbf{pb}]])\uparrow\!\!\uparrow$
$rP(pn)\downarrow\!\!\downarrow$ or
$wP(pn,c)\downarrow\!\!\downarrow$
$wP(pn,c)\downarrow\!\!\downarrow / \mathbf{P}[\mathbf{C}[pn]]' = c$
or fault
$\mathbf{p} = \mathit{error}$
fault

Figure 3: Transitions of the physical disk, where
fault stands for $(csf(pn)\downarrow\!\!\downarrow / \mathbf{C}'[pn] = j)$
or $(dam(pn)\downarrow\!\!\downarrow / \mathbf{P}'[pn] = cd)$,
for some $j \in \mathit{PN}$.

5. **Validity condition:**
The validity condition is defined as $V_P \doteq R_P \to G_P$ where the rely condition
$R_P \doteq \Box(\mathbf{p} = \mathit{retr} \to (\mathbf{e} \neq rP(pn) \wedge \mathbf{e} \neq wP(pn,c)))$
expresses that the user never generates the request and write events before the physical disk has responded to a previous request. And the guarantee condition
$G_P \doteq \Box(\mathbf{e} = rP(pn) \to \Diamond\mathbf{e} = sP(\mathbf{P}[\mathbf{C}[pn]]))$
expresses that the physical disk then guarantees that the user eventually will get a response to a request.

Correctness As seen in sect. 3.1 one must define the condition $\neg FO$ that expresses that the anticipated faults never occur, i.e.,
$\neg FO \doteq \Box(\mathbf{e} \neq dam(pn) \wedge \mathbf{e} \neq csf(pn))$.
For correctness of relative refinement one has to prove cf. sect. 3.1 $[P \wedge \neg FO]_{\delta_P} \to [A]_\alpha$.

The proof goes intuitively as indicated in section 2.3, i.e., $\neg FO$ and $\Box T_P$ (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 T (the DTL formula corresponding to fig. 2) shows that these are the same except that states and events have another name.

3.5 Third Step: Detection layer

In this step, the second stage in our development of stable storage, we specify the layer that detects the faults that we assumed could affect the physical disk; this layer is added in bottom-up fashion to the physical disk specified in section 3.4. 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* [8]. 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. The second one checks whether the contents read from the physical disk originate from the right location.

Specification The detection layer consists of three parts: the first part detects the disk surface errors using a cyclic redundancy check (CRC) mechanism [8]. The second part detects the control system errors using an address checking (ADR) mechanism [8]. 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.

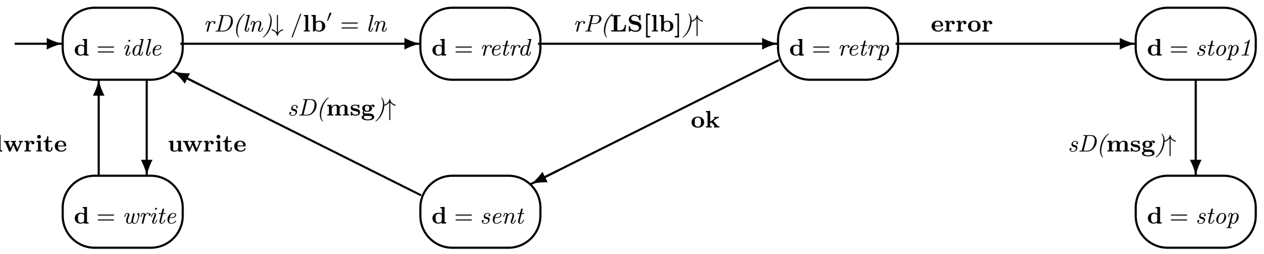

Figure 4: Transitions of the detection layer where
ok stands for $sP(c)\Downarrow$ $[CC(c) \wedge AC(CD(c), \mathbf{LS}[\mathbf{lb}]))]/\mathbf{msg}' = AD(CD(c)))$ and
error stands for $sP(c)\Downarrow$ $[CC(c) \wedge \neg AC(CD(c), \mathbf{LS}[\mathbf{lb}]))]/\mathbf{msg}' = DCSF$ or $sP(c)\Downarrow$ $[\neg CC(c)]/\mathbf{msg}' = DSDF$ and
uwrite stands for $wD(ln,d1)\Downarrow$ $/\mathbf{pb}', \mathbf{lb}' = CE(AE(ln,d1)),ln$ and **dwrite** stands for $wP(\mathbf{LS}[\mathbf{lb}], \mathbf{pb})\Uparrow$

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. We also introduce logical sector numbers; these logical sector numbers will be needed in the fourth step, but we give them already here, i.e., 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 $D_{fs} \doteq M_{D_{fs}} \wedge V_{D_{fs}}$ is not given in full detail. Of $M_{D_{fs}}$ only the transitions are given in fig. 4 where *CC* is used to check whether crc encoded data from the physical disk is damaged by a disk surface fault, and *CD* and *CE* are the crc-decode and -encode functions, and *AC* is used to check whether data is read from the correct physical location, and *AD* and *AE* are the address-decode and -encode functions.
correctness As seen in sect. 3.1 one needs the condition $\neg ED$ that expresses that no errors are detected. This condition is defined as follows:
$$\neg ED \doteq \Box(CC(\mathbf{P}[\mathbf{LS}[ln]]) \leftrightarrow \mathbf{P}[\mathbf{LS}[ln]] \neq cd \wedge AC(CD(\mathbf{P}[\mathbf{LS}[ln]]), \mathbf{LS}[ln])) \leftrightarrow \mathbf{C}[\mathbf{LS}[ln]] = \mathbf{LS}[ln]).$$
It prevents that the **error**-branch in fig. 4 will be taken.

3.6 Fourth step: Error recovery layer

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* [8]. 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).
The formal specification of the error-recovery layer is omitted.
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 RC is as follows

$$RC \doteq \Box(\exists i \in \mathbf{Ena}.CC_i(\mathbf{P}[\mathbf{LS}[ln]]) \\ \wedge AC(CD(\mathbf{P}[\mathbf{LS}[ln]]), ln))$$

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 $\neg ED$ of the detection layers because we have made the design decision that the detection layer is responsible for finding the spare sectors.

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 and the part originally intended to specify safety properties is used for the till now developed (possible faulty) implementation. This enabled us to prove relative refinement of a specification, i.e., the intersection of both parts of the implementation is a refinement, indeed.

Acknowledgements

We would like to thank Henk Schepers, Ruurd Kuiper, Amir Pnueli and all anonymous referees for their comments.

References

1. B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.
2. A. Cau and W.-P. de Roever. Using relative refinement for fault tolerance. In *Proc. of FME'93 symposium: industrial strength formal methods*, 1993.
3. A. Cau, R. Kuiper, and W.-P. de Roever. Formalising Dijkstra's development strategy within Stark's formalism. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *Proc. 5th. BCS-FACS Refinement Workshop*, 1992.
4. F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Trans. on Software Engineering*, 11(1):23–31, 1985.
5. 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.
6. E. Dijkstra. A tutorial on the split binary semaphore, 1979. EWD 703.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
8. P. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems.* Springer-Verlag, second, revised edition, 1990.
9. 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.
10. P. Place, W. Wood, and M. Tudball. Survey of formal specification techniques for reactive systems. Technical Report, 1990.
11. 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.

12. E. 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.

13. E. Stark. A Proof Technique for Rely/Guarantee Properties. In *LNCS 206: Fifth Conf. on FST and TCS*, pages 369–391. Springer-Verlag, 1985.