Specifying fault tolerant programs in deontic logic

Eindhoven University of Technology

Department of Mathematics and Computing Science

Specifying Fault Tolerant Programs in Deontic Logic

by

J. Coenen

91/34

Computing Science Note 91/34
Eindhoven, December 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Specifying Fault Tolerant Programs in Deontic Logic *

J. Coenen †

Dept. of Math. and Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands

Abstract

Fault tolerant systems are, like most complex systems, structured in layers. In a fault tolerant layered system, a malfunction on a lower level layer occurs at a higher level as an imperfection, that may prohibit the upper layer from exhibiting its preferred behaviour. The behaviour of a system in such less than perfect circumstances should not be left unspecified. Deontic logic offers the possibility to specify layered fault tolerant systems in a natural way. More specifically, dyadic modalities are used to specify the preferred behaviors of a fault tolerant system in different conditions. The use of dyadic rather than monadic modalities is also discussed in the context of a particular problem that arises when specifying fault tolerant systems and which is referred to as the 'lazy programmer' paradox.

The 'lazy programmer' paradox is illustrated by a few 'toy' examples. The application of deontic logic as a specification language for fault tolerant systems is illustrated by the specification of a non-trivial example.

Keywords: Deontic logic, Exception handling, Fault tolerance, Layered systems, Lazy programmer paradox, System specification.

1 Introduction

A promising application of deontic logic in computer science is the specification of fault tolerant systems, in particular fault tolerant software. The distinction between the behaviour of a system in perfect and less than perfect worlds comes naturally when considering its degree of reliability. In dyadic deontic logic [vWright81] it is possible to distinguish the behaviour in an ideal world from the (preferred) behaviour in a less ideal world. This possibility makes deontic logic a suitable language for asserting fault tolerance properties about program. To illustrate the advantages of deontic logic, we may compare it with a more classical system for proving the correctness of programs.

In Hoare's original logic [Hoare69] a program is considered correct, if it behaves according to its specification under the assumption of a faultless execution mechanism.

*To appear in *Proc. First International Workshop on Deontic Logic in Computer Science.*

†Supported by NWO/SION Project 612-316-022: "Fault Tolerance: Paradigms, Models, Logics, Construction." E-mail: wsinjosc@win.tue.nl

However, under realistic conditions all hardware is error-prone. For many applications it is unacceptable that a system's behaviour is left unspecified in case a particular fault occurs. A short power cut should not result in an arbitrary behaviour of your car's cruise control system. To overcome this deficiency, Cristian [Cristian85] extended Hoare's logic to deal with a particular class of failures called exceptions. Exceptions are used to signal a process that an error has been detected. For sequential programs it is necessary that all possible faults can and will be detected. As such, an exception can be understood as a detected fault that can be dealt with in an elegant way. Although exceptions model 'nice' faults — they can effect a program's behaviour only in a very restricted manner — it is a useful concept in fault tolerant system design [Cristian89].

One of the main advantages of Cristian's approach to exception handling is that his theory is a straightforward extension of Hoare's logic, and therefore easy to comprehend by practitioners in the field. So, the question arises why one should use deontic logic to specify fault tolerant systems? To answer this question in a satisfactory way it is necessary to shed some light on the construction of fault tolerant systems.

A fault tolerant system is, like most complex systems, structured in layers. On the one hand, a layer may use the services delivered by its lower level layer to provide a service to its upper level layer. On the other hand, a layer may receive an exception from its lower level layer or raise an exception to signal its upper layer that it cannot provide a requested service. At each level, the system tries to handle the exceptions raised by the layer below. If the current layer is unable to cope with the current situation it may decide to raise an exception itself. In this way a malfunctioning of the underlying execution mechanism may gradually propagate to a layer which can deal with it in a satisfactory manner. A layer can therefore be regarded as an *ideal fault tolerant component* in the sense of Anderson and Lee [Anderson90], see figure 1. The arc directed from 'exceptional

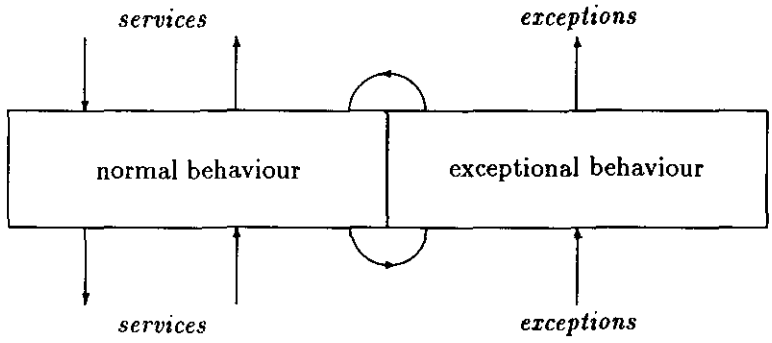

Figure 1: *Layer viewed as an ideal fault tolerant component*

behaviour' to 'normal behaviour' represents the case that the current layer handles an exception raised by a lower level (or the current level). The arc directed from 'normal behaviour' to 'exceptional behaviour' represents the case that an exception is raised by a lower level (or the current level).

To achieve a layered structure as described above, it must be possible to program a deliberately raised exception. In Cristian's formalism it is not possible to distinguish

between deliberately raised exceptions and exceptions due to a physical fault in the executing hardware. Now, consider a specification of a program that computes the factorial $N!$ for input N. If an intermediate result of the computation causes an integer overflow, signalled by the exception *ovf*, it is specified that the result is zero. A lazy programmer might be tempted to write a program that outputs zero immediately and raises the exception *ovf* deliberately. This is of course not an acceptable implementation — the exception should only be raised due to an overflow in the underlying hardware — which can be avoided by explicitly stating that the programmer is not allowed to raise the exception *ovf*. This works well for this particular example, but it was already mentioned that it should be allowed to raise certain exceptions deliberately, e.g. to prevent undefined results. Because it is in general not possible to predict when such exceptions may occur, the lazy programmer cannot be prohibited from abusing his privilege to raise exceptions deliberately. This problem is referred to as the 'lazy programmer' paradox, although it is actually an expressiveness problem rather than a formal paradox. The solution to this problem presented in this report exploits the possibility of deontic logic to express that some behaviors are preferred over others.

The remainder of this report is organized as follows. In section 2, a programming language is defined and an intuitive explanation of the language constructs is given. In this section three small programs are explained. These programs are also used in section 3 to motivate the introduction, and explain the meaning of, dyadic modalities in the specification language. Section 3 introduces the specification language, and discusses the 'lazy programmer' paradox in more detail. Section 4 includes an informal description of a non-trivial fault tolerant system. The application of deontic logic as a specification language is illustrated in section 5, by establishing a small property of the example outlined in section 4. Finally, section 6 contains a comparison with related work and some suggestions for future work.

2 Programming Language

In this section a small subset of an Ada-like 'programming' language [Ada83], called $\mathcal{P}rog$ is defined. This programming language is also used in section 4 to describe some of operations used in the example. The main feature of the programming language $\mathcal{P}rog$ is that it provides a notation for exception handling.

Given the following basic sets:

- $\mathcal{V}ar$, the set of program variables, with typical element x;
- $\mathcal{E}xc$, the set of exceptions, with typical element exc;
- $\mathcal{E}xpr$, the set of expressions with occurrences of program variables, with typical element exp;
- $\mathcal{B}exp$, the set of boolean expressions with occurrences of program variables, with typical element b;

the syntactic class $\mathcal{P}rog$ of programs, with typical element S, is defined by

$$S ::= \mathbf{null} \mid x := exp \mid \mathbf{raise}\ exc \mid \mathbf{begin}\ S\ \mathbf{end}$$

$| S_1;\ S_2 |$ **if** b **then** S_1 **else** S_2 **fi** | **while** b **do** S

| **begin** S_0 **exception when** $exc_1 \Rightarrow S_1 \ldots$ **when** $exc_k \Rightarrow S_k$ **end**

The meaning of the programming language constructs in $\mathcal{P}rog$ is as follows.

- The empty statement **null** has no effect other than skipping to the next statement.
- The assignment statement $x := exp$ assigns the value of the expression exp to the program variable x.
- The raise statement **raise** exc raises the exception exc. As a side effect it causes the execution of the program to continue at the innermost enclosing exception handler, that handles exc exceptions. If such enclosing exception handler does not exist, program execution is aborted.
- The simple block statement **begin** S **end** groups the statements in S in a single block. It may be regarded as a pair of parenthesis.
- $S_1;\ S_2$ is the sequential composition of the programs S_1 and S_2. First S_1 is executed, and if S_1 terminates successfully, then S_2 executed.
- In case of the alternative statement **if** b **then** S_1 **else** S_2 **fi**, the subprogram S_1 is executed if the boolean guard b is true, and S_2 is executed otherwise.
- The iterative statement **while** b **do** S is skipped if b is initially false. If b initially is true, then execution of S is repeated until b becomes false.
- **begin** S_0 **exception when** $exc_1 \Rightarrow S_1 \ldots$ **when** $exc_k \Rightarrow S_k$ **end** is executed as follows. The program starts with the execution of S_0. If during the execution of S_0 an exception exc_i $(i = 1, \ldots, k)$ is raised, then the execution of S_0 is aborted and the program resumes with the execution of S_i. If an exception other than exc_i $(i = 1, \ldots, k)$ is raised, then execution of S_0 is aborted, and the exception is passed to the next enclosing block. If there isn't an enclosing block the program is aborted. If S_0 terminates without raising an exception, then the program terminates normally.

For example, the programs listed in figure 2 are executed as follows. Program a assigns the factorial of N to variable x unless an ovf exception occurs — meaning that an overflow has been detected —in which case x is set to zero. Program b sets x to zero and then raises ovf deliberately. Program c assigns $N!$ to x if initially N is less or equal than K, and sets x to zero in case N is larger than K.

3 Specification Language

The specification language combines deontic logic with first-order predicate logic, and is inspired by the logic used in [vEck82]. The basic modality of the deontic logic is the dyadic obligation $\varphi O \psi$. The introduction of dyadic modalities is motivated near the end of this section when specifying the running examples in figure 2. The first-order predicates in the specification language are used to quantify over logical variables only. Thus program variables, both primed and unprimed, occur free only in specifications.

Assume that the following sets are defined:

(a) **begin** $x := N!$ **exception** $ovf \Rightarrow x := 0$ **end**

(b) **begin** $x := 0$; **raise** ovf **end**

(c) **begin if** $N \leq K$ **then** $x := N!$ **else** $x := 0$ **fi end**

Figure 2: *Running examples*

- $\mathcal{E}xpr'$, the extended set of expressions over program variables, which may be decorated with a prime. Thus $\mathcal{E}xpr \subset \mathcal{E}xpr'$.
- $\mathcal{L}var$, the set of logical variables, such that $\mathcal{L}var \cap \mathcal{E}xpr = \emptyset$. Logical variables never have primes attached to them.

A primed program variable x' refers to the value of the variable x before executing a program, whereas an unprimed variable x refers to the value of x after the execution of the program. The use of primed and unprimed variables in expressions captures the concept of initial and final states syntactically. Hence, it gives the specification language the dynamic nature which is needed to reason about programs.

Given the sets above, the syntax of assertions $\varphi, \psi \in \mathcal{A}ssn$ is defined by ($exp_0, exp_1 \in \mathcal{E}xpr'$, $exc \in \mathcal{E}xc$, and $g \in \mathcal{L}var$)

$$\varphi \ ::= \ \mathsf{true} \mid exp_0 = exp_1 \mid exp_0 \leq exp_1 \mid \delta(exc) \mid \neg\varphi \mid \varphi \rightarrow \psi \mid \exists_g(\varphi) \mid \varphi O \psi$$

Notice that quantification is only allowed over logical variables. Besides the usual abbreviations for predicate logic (such as $\forall_g(\varphi)$ for $\neg\exists_g(\neg\varphi)$), the following derived operators are defined

$$\begin{array}{lcllcl} O\varphi & \triangleq & \mathsf{true} O\varphi & & & \\ \varphi F\psi & \triangleq & \varphi O\neg\psi & , & F\varphi & \triangleq & \mathsf{true} F\varphi \\ \varphi P\psi & \triangleq & \neg(\varphi O\neg\psi) & , & P\varphi & \triangleq & \mathsf{true} P\varphi \end{array}$$

The meaning of $\delta(exc)$ is that exception exc was raised. The notation δ is used to stress the difference with variables that refer to states instead of events. The meaning of $\varphi O\psi$ is that in all φ-perfect worlds (worlds that are perfect except that φ is the case) ψ is true. Hence, $O\varphi$ expresses that φ is the case in all perfect worlds. Similarly, $\varphi P\psi$ and $\varphi F\psi$ express that in all φ-perfect worlds ψ is respectively permitted and forbidden.

Below two standard derivation rules of deontic logic are given (see e.g. [Aqvist83]).

$$\frac{\varphi,\ \varphi \rightarrow \psi}{\psi} \ \text{(MP)} \qquad \frac{\vdash \psi}{\vdash \varphi O\psi} \ \text{(Necessitation)}$$

The axioms below are more typical for the application discussed in the introduction. The first two are still quite common axioms, that should cause no problems. The third axiom is more typical for the logic. It expresses that all relative perfect worlds are perfect alternatives to themselves. Or more loosely, there is only one perfect alternative for each

world. It is motivated by the fact that the set of possible executions of a program does not change unless new faults are introduced.

$$\vdash \varphi O(\psi \rightarrow \chi) \rightarrow (\varphi O\psi \rightarrow \varphi O\chi)$$
$$\vdash \varphi O(\psi \wedge \chi) \leftrightarrow (\varphi O\psi \wedge \varphi O\chi)$$
$$\vdash \varphi O(O\psi) \rightarrow \varphi O\psi$$

This is of course not intended to be a complete axiomatization. The axiomatization of the logic itself is part of ongoing research in which there are still a lot of questions to be settled.

The most characteristic difference of the logic defined above and the ones that can be found in the literature about system specification (e.g. [Maibaum86, Khosla88]) is that the above logic includes *dyadic* modalities. For example, Khosla [Khosla88] uses the monadic modalities $O\alpha$ respectively $P\alpha$ to express that the action α *must* respectively *may* be performed. Thus the deontic aspect of the specification language in [Khosla88] is used only to reason over the freedom of choice. In particular, a predicate $O\alpha$ is defined such that α is the *only* action that is obliged. Hence the formula $O\alpha \wedge O\beta$ is equivalent to false per definition if $\alpha \neq \beta$. When specifying fault tolerant systems this causes a problem, which in the more general context of deontic logic is known as the Chisholm paradox (see [Aqvist67]).

Consider the following specification for a program that tries to anticipate a possible division by zero, when computing $1/x$ for input x.

$$O(x' \neq 0) \wedge O(x' \neq 0 \rightarrow y = 1/x') \wedge (x' = 0 \rightarrow O(y = 0)) \qquad (1)$$

This specification expresses that the input x is expected not to be zero, and it should be the case that if input x is not zero then y is $1/x$, and if x is zero then y ought to be zero. This seemingly correct specification is inconsistent in case the input x is zero. The problem is that the monadic modalities refer to perfect worlds only, which may lead to a conflict of duties once one finds himself in a less than perfect world. The behaviour of a fault tolerant system in less than perfect conditions should be specified, as the predicate 'fault tolerant' suggests.

Dyadic deontic logic allows one to make assertions about less than perfect worlds. For example, the last conjunct of (1) may be replaced by $(x' = 0)O(y = 0)$, which does not result in an inconsistency if $x' = 0$. Notice that $\varphi O\psi$ implicitly implies that ψ ought to be established, but φ is 'provided' for and not to be established. This observation is the key to the solution of the 'lazy programmer' paradox at the end of this section.

Although in this report the use of dyadic modalities is limited to the special case in which only exceptions occur on the left side of the modality, i.e. dyadic modalities occur in specifications only according to the format $\delta(exc)O\psi$, it is permitted to have predicates on the left side also. The practical use of having predicates on the left side is illustrated by the example in the next paragraph.

A standard technique to obtain a higher degree of reliability is the duplication of system components. For example, one may use two different algorithms to compute a certain value and compare the outcomes, say x and y, of these computations. In case $x \neq y$ at least one of the computations resulted in an error, and in case $x = y$ either both computations were correct or both computations yielded the same erroneous result. If one assumes that the probability of the latter case occurring is zero, the system sketched

above is fault tolerant. A schematic drawing of a component that compares the outputs x and y is pictured in figure 3. The comparator may be specified by

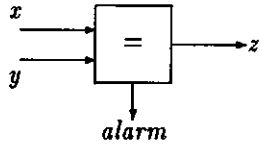

Figure 3: *Comparator*

$$O(z = x \wedge z = y \wedge \neg alarm) \wedge (x \neq y)O(alarm) ,$$

According to its specification, the comparator ought to set z equal to x and y, and set *alarm* to false . In case $x \neq y$ — and hence, it is not possible to set z equal to both x and y — *alarm* must be set to true .

3.1 On Lazy Programmers, and How to Put Them to Work

In the introduction it was already mentioned that the possibility to express which behaviors are preferred may be exploited to solve the 'lazy programmer' paradox. The particular formulation of the 'lazy programmer' paradox for fault tolerance [1] has a striking similarity with the 'Good Samaritan' paradox (see [Aqvist67]). A program that is designed to tolerate only faults intentionally caused by that program itself, hardly deserves the predicate 'fault tolerant', just as little as a thief who salvages his own victims deserves to be called a Good Samaritan.

The programs in figure 2 serve to illustrate the previous discussion. Consider the following naive specification for a program that is to compute the factorial of N.

$$O((\neg\delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)) .$$

The specifier has anticipated that, due to hardware limitations, it is possible that during the computation of $N!$ an overflow, signalled by exception *ovf*, occurs. If the overflow indeed occurs then x should be set to zero, otherwise x ought to be equal to $N!$. However, nothing prevents the lazy programmer from simulating an overflow as in program b of figure 2. Unfortunately, program b does satisfy the above specification. Using dyadic modalities one can specify program a as follows.

$$O(x = N!) \wedge \delta(ovf)O(x = 0) . \tag{2}$$

This specification expresses that is preferred to set x equal to $N!$, and if this is not possible due to an overflow x ought to be zero. Program b may be specified by

$$O(x = 0 \wedge \delta(ovf)) , \tag{3}$$

[1] Lazy programmers were already a problem in Hoare's logic, where each *divergent* program is a correct implementation of every specification. This particular version of the 'lazy programmer' paradox is solved in so called *total* correctness formalisms, whereas the logic of Hoare [Hoare69] is a *partial* correctness formalism.

which expresses that x ought to be set to zero and the exception *ovf* ought to be raised. Provided that the axiomatization of the deontic logic does not allow one to derive (2) from (3), it is not possible to prove that program b is a correct implementation of (2). As a matter of fact, program b can be excluded more explicitly by adding the conjunct $F\delta(ovf)$ to (2). Hence, the lazy programmer has to think of other means to avoid working.

Basically, the 'lazy programmer' paradox is solved by making the specification language more expressive. This imposes some requirements on the semantics and axiomatization of the programming language to avoid the situation in which an intuitively correct program does not satisfy a given specification. For example, suppose that the maximum number, say *MaxInt*, that can be computed without causing an overflow is known. If K is chosen such that $K! \leq MaxInt < (K+1)!$, then program c in figure 2 is intuitively a correct implementation of (2). The specification of program c is, however, as follows.

$$O((N \leq K \wedge x = N!) \vee (\neg N \leq K \wedge x = 0)) \tag{4}$$

The only way to prove that (4) specifies a correct implementation, i.e. to prove that (4) implies (2), is by making the knowledge about the hardware limitation explicit. For instance by including the following axioms.

$$\vdash O(N \leq K) \tag{5}$$

$$\vdash O(\neg N \leq K \rightarrow \varphi) \rightarrow \delta(ovf)O\varphi \tag{6}$$

Axiom (5) expresses that it ought to be the case that $N \leq K$. Axiom (6) expresses that if one is obliged to establish φ if $\neg N \leq K$ in a faultless world, this implies that φ ought to be established even if an overflow occurred. The second axiom is motivated by the knowledge that the overflow would have occurred anyway if $\neg N \leq K$. Because (4) is equivalent with

$$O(N \leq K \rightarrow x = N!) \wedge O(\neg N \leq K \rightarrow x = 0)\,,$$

which implies

$$(O(N \leq K) \rightarrow O(x = N!)) \wedge O(\neg N \leq K \rightarrow x = 0)$$

it follows from (5) and (6) that program c is a correct implementation of (2), *provided* that above assumptions hold. Thus only if the hardware limitations are such that the axioms are justified the above reasoning holds.

It is possible to think of clever variations on the programs in figure 2, e.g. the ones in figure 4, for which the correct arguments to accept or reject them as correct implementations of (2) are not so easily found.[2] However, these problems should be solved in the semantics and the axiomatization of the programming language. The purpose of the previous discussion is to demonstrate that dyadic deontic logic is expressive enough to distinguish deliberate errors from unintentional ones.

4 Informal Description of a Stable Storage

An important concept in fault tolerant computing is the atomicity of actions. An action is atomic if it is either executed successfully or not executed at all. Atomic actions can

[2]Program d should be rejected, but just including $F\delta(ovf)$ in the specification would also exclude program e which might be acceptable.

(d)	**begin** $x := N!$; **raise** *ovf* **exception** *ovf* $\Rightarrow$ $x := 0$ **end**
(e)	**begin if** $N \leq K$ **then** $x := N!$ **else raise** *ovf* **fi exception** *ovf* $\Rightarrow$ $x := 0$ **end**

Figure 4: *The lazy programmer strikes back*[3]

be implemented by creating a checkpoint before the action is executed, and if an error is detected by recovering the original state from this checkpoint. The checkpoint should be recorded on a reliable medium, called a stable storage. This section contains a summary of some aspects of a particular stable storage and focuses on the implementation of the read operation. A more complete description of the stable storage described below is given in [Schepers91].

The stable storage consists of three layers. At the lowest level, the stable storage is implemented by a number of physical disks. These physical disks, with the appropriate operations on them, are grouped in the so called 'physical disk' layer. Each physical disk has a corresponding logical disk, that abstracts from the physical location of sectors on the physical disk, by maintaining a flexible mapping between logical addresses and physical sector numbers. The logical disks are grouped together in the 'logical disk' layer. The layer at the top level is called the 'reliable disk' layer. The reliable disk layer provides a single stable storage, which is implemented by several logical disks.

It is assumed that the only relevant errors are caused by damaged sectors of the physical disks. In the remainder of this section the layers are examined in somewhat more detail.

4.1 Reliable Disk Layer

The reliable disk layer provides a *read_sector* operation, with the intention that the contents of the sector with logical address *address* is retrieved in the variable *sector*. For this purpose, the reliable disk layer records which logical disks are still operational, i.e. which logical disks have not yet caused a *logical_disk_crash* exception. The numbers of the operational logical disks are administered in the set *operational_disks*. On invocation of the *read_sector* operation, an operational logical disk is selected on which a *read_logical_disk* operation is performed.

The reliable disk layer must anticipate two exceptions that may be raised by the logical disk layer. The exception *logical_sector_lost* indicates that this logical disk is unable to return the contents of the sector with logical address *address*. The exception *logical_disk_crash* is raised when the logical disk layer can no longer guarantee consistency of the information stored in the logical disk. In case of a *logical_sector_lost* exception, the reliable disk layer attempts to retrieve the sector from another logical disk. The retrieve operation will be left unspecified, but notice that retrieving the lost sector might include a recursive call of *read_sector*.

The *logical_disk_crash* exception is handled simply by deleting the corresponding disk number from the set *operational_disks*. If the reliable disk layer runs out of operational

[3] Or is it a too diligent programmer?

logical disks it raises a *reliable_disk_crash* exception. See also figure 5 .

```
begin
  success := false ;
  while ¬success do
  begin
    disknr := a member of operational_disks ;
    read_logical_disk(disknr, address) ;
    success := true
  exception
    when logical_sector_lost ⇒
      retrieve the lost sector
    when logical_disk_crash ⇒
      operational_disks := operational_disks - {disknr} ;
      if operational_disks = ∅
        then raise reliable_disk_crash
        else null
      fi
  end
end
```

Figure 5: *read_sector*

Notice that the nondeterminism in the selection of an operational disk needs to be resolved. This freedom of choice may be exploited to obtain a more efficient *read_sector* operation.

4.2 Logical Disk Layer

Whereas the reliable disk layer achieves a higher degree of reliability through the redundancy of the logical disks, the logical disk layer, on its turn, achieves a higher degree of reliability through the redundancy of so called spare sectors on each logical disk. The spare sectors are recorded in the set *spare_sectors*. Furthermore, the logical disk layer abstracts from the physical location of sectors by maintaining a mapping *log_to_phys* between logical addresses and sector numbers.

The read operation at the logical disk level is listed in figure 6. The logical disk layer simply calls the *read_physical_disk* operation with the converted address. If the physical disk layer raises the *invalid_crc* exception and there are no spare sectors left, then the logical disk layer raises a *logical_disk_crash* exception. If the *invalid_crc* exception is raised and there are spare sectors, then one of the spare sectors is selected and the mapping *log_to_phys* is updated, and the *logical_sector_lost* exception is raised

4.3 Physical Disk Layer

The physical disk layer achieves reliability by using information redundancy. The contents of each logical sector is augmented with a cyclic redundancy code. It is assumed

```
begin
  read_physical_sector(log_to_phys(address))
exception
  when invalid_crc ⇒
    if spare_sectors = ∅
      then raise logical_disk_crash
      else new_sector := a member of spare_sectors ;
        spare_sectors := spare_sectors - {new_sector} ;
        update log_to_phys ;
        raise logical_sector_lost
    fi
end
```

Figure 6: *read_logical_disk*

that all relevant faults can be detected with this code. Or more precisely, the probability of not detecting a relevant error is sufficiently small. This means that faults like damaged disk drives etc. are not considered relevant in this example. The *read_physical_sector* operation is listed in figure 7.

```
begin
  sector := physical_disk[sector_nr] ;
  if ¬cyc_red_check(sector)
    then raise invalid_crc
    else null
  fi
end
```

Figure 7: *read_physical_sector*

The cyclic redundancy code is checked by the function *cyc_red_check*, which may be implemented by special purpose hardware.

5 Specification of the Read Operations

A specification of an operation of a fault tolerant system typically has the following format

$$\varphi_1 O \psi_1 \wedge \ldots \wedge \varphi_n O \psi_n \ .$$

Each ψ_i specifies how the operation of this layer should behave, provided the lower level created the condition φ_i. Because the upper level layer cannot interfere with the actions of the lower level layer, the conditions φ_i are established facts for the upper level layer to

which it is supposed to react according to ψ_i. For example, at the top level of the stable storage the read operation may have been specified as follows.

$$O(sector = reliable\_disk'(address')) \wedge \delta(reliable\_disk\_crash)O\psi \ ,$$

where ψ is left open for the moment. Thus it is specified that the read operation ought to assign the initial contents of the stable storage at address *address* to *sector*. In case a *reliable_disk_crash* exception was raised, ψ ought to be established. Of course, one might also have specified that e.g. the address or contents of the storage ought to be left unchanged.

Because the physical disk layer is the lowest level of the stable storage and it is assumed that *cyc_red_check* detects all errors, there are no faults (from lower levels) that must be anticipated by this layer. Therefore, the specification of the *read_physical_sector* operation (figure 7) contains only monadic modalities. The *read_physical_sector* operation (for physical disk i) is specified by

$$O(sector = physical\_disk_i'[sectornr'])$$
$$\wedge O(\delta(invalid\_crc) \rightarrow \neg cyc\_red\_check(sector)) \ .$$

The second conjunct of this specification can be rewritten as

$$F(\delta(invalid\_crc) \wedge cyc\_red\_check(sector)) \ ,$$

which forbids to raise the *invalid_crc* exception when the sector passes the cyclic redundancy check. Now suppose an *invalid_crc* exception ought to be raised, i.e.

$$O\delta(invalid\_crc) \ .$$

From the specification of the *read_physical_sector* operation it follows that

$$O(\delta(invalid\_crc) \rightarrow \neg cyc\_red\_check(sector)) \ .$$

This together with the following axiom

$$O(\delta(invalid\_crc) \rightarrow \neg cyc\_red\_check(sector))$$
$$\rightarrow (O\delta(invalid\_crc) \rightarrow O\neg cyc\_red\_check(sector))$$

is sufficient to derive

$$O\delta(invalid\_crc) \rightarrow O\neg cyc\_red\_check(sector)$$

with modus ponens. One more application of modus ponens results in

$$O\neg cyc\_red\_check(sector) \ .$$

Hence, under the assumption that the physical disk functions correctly it is established that the *invalid_crc* exception ought to be raised only if the sector didn't pass the cyclic redundancy check.

The logical disk layer must anticipate an *invalid_crc* exception, but is allowed to raise a *logical_disk_crash* exception or a *logical_sector_lost* exception depending on whether there

are any spare sectors available (figure 6). The *read_logical_sector* operation (for logical disk i) is specified by

$$O(sector = logical\_disk_i'(address')) \wedge$$
$$\delta(invalid\_crc)O((\delta(logical\_disk\_crash) \wedge spare\_sectors_i' = \emptyset) \vee$$
$$(\delta(logical\_sector\_lost) \wedge spare\_sectors_i' \neq \emptyset)) .$$

A single logical disk cannot handle an *invalid_crc* exception by itself, but achieves graceful degradation through the discrimination between the fatal situation in which there aren't any spare sectors left, and the less harmful situation when there are enough redundant sectors. Assuming that this layer functions correctly, it follows that a *logical_disk_crash* exception is raised if an *invalid_crc* exception was detected and initially the number of spare sectors was zero. To ensure that a *logical_disk_crash* or *logical_sector_lost* is raised only in the situation described above, the specification may be strengthened by adding the conjunct $F(\delta(logical\_disk\_crash) \wedge \delta(logical\_sector\_lost))$, which forbids raising these exceptions deliberately. Notice that this specification is not complete because it does not specify that the mapping *log_to_phys* should be updated before raising the *logical_sector_lost* exception.

Although the reliable disk layer must handle both exceptions that may possibly be raised by the logical disk layer, the specification below only anticipates the occurrence of a *logical_disk_crash* exception. Therefore also this specification is not complete. The *read_sector* operation (figure 5) of the reliable disk layer is specified by

$$O\exists_i(i \in operational\_disks' \wedge sector = logical\_disk_i'(address'))$$
$$\wedge \quad \delta(logical\_disk\_crash)O(\delta(reliable\_disk\_crash) \rightarrow operational\_disks = \emptyset) .$$

Suppose that it is forbidden to raise the *reliable_disk_crash* exception deliberately, which may be accomplished by adding the conjunct $F\delta(reliable\_disk\_crash)$ to the specification above. Then it follows that an *reliable_disk_crash* exception is only raised if there are no other operational disks left and a *logical_disk_crash* was raised. Thus the only initially operational disk doesn't have the appropriate information.

6 Discussion

The previous section illustrates how deontic logic provides the possibility to specify fault tolerant systems in a natural way. It turns out that to derive certain properties of a specified system, one needs to make the assumptions about faults and their effect on the behaviour the system explicit. The possibility to express the preference of some behaviors over others, allows one to distinguish between conditions created by a possible malfunctioning of a lower level, and the conditions created by the layer under discussion itself. Although deontic logics have been suggested for system specification before, e.g. in [Maibaum86, Khosla88], the application to fault tolerant systems seems to be new, which partly explains the differences between the specification language used in this report and those appearing in the literature about system specification.

The deontic logic described in this report differs from the deontic logics for system specification in the existing literature mainly in two ways. Firstly, the logic in this report is a *dyadic* deontic logic, whereas the logics in e.g. [Meyer88] and [Khosla88] are *monadic*

deontic logics. Secondly, primed and unprimed variables are used to capture the dynamic aspect of programs in the specification language, whereas Meyer [Meyer88] and Khosla [Khosla88] use a dynamic logic in combination with the deontic logic.

The first difference, which seems to be the most essential one, can be explained by the particular application to fault tolerant systems. An important concept in fault tolerance is *graceful degradation*, which allows a system to temporarily sacrifice a service in favor of a more important one if a fault occurs. This corresponds in a natural way with deontic logic specification of the format $\varphi_1 O\psi_1 \wedge \ldots \varphi_n O\psi_n$ that specify the behaviour ψ_i of a system under less than perfect conditions φ_i $(i = 1, \ldots, n)$. Moreover, dyadic deontic logic offers a solution to the 'lazy programmer' paradox described in section 3. And, although the examples used to illustrate this paradox may be regarded as 'toy' examples, it should be evident from the example in section 4 that this problem becomes more important as the complexity of a system increases.

The second difference concerns primed variables. A nice property of the logic is that it captures *state* predicates as well as *action* predicates. State predicates are predicates with either only primed variables or only unprimed variables. Action predicates are predicates with both primed and unprimed variables. Thus, following the terminology of von Wright [vWright81], the logic described in this report is both a 'Seinsollen'-logic (concerning states) and a 'Tunsollen'-logic (concerning actions). A serious disadvantage of the primed and unprimed variables is that it is not clear how this method can be extended to deal with (distributed) real-time systems, which is an important application area of fault tolerance. Such systems may be specified in a logic that mixes deontic logic with a temporal logic, or in a logic with combined deontic-temporal modalities like the one in [vEck82]. For example, one may think of a propositional metric temporal logic as proposed by Koymans [Koymans90] that is extended with the dyadic obligation.

It is clear that a lot of work remains to be done. Future work will include the development of a formal model for the logic and a complete axiomatization that fits the application area of fault tolerant systems. A next step, thereafter, may be the development of a deontic logic proof system for (layered) fault tolerant systems.

Acknowledgment. I would like to thank Henk Schepers for providing the stable storage example, and Tijn Borghuis and Wim Koole for many helpful discussions.

References

[Ada83] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual.* ANSI/MIL-STD-1815A-1983, LNCS 155. Springer-Verlag, 1983.

[Anderson90] T. Anderson & P.A. Lee. *Fault Tolerance: Principles and Practice*, 2nd. revised edition. Springer-Verlag, 1990.

[Aqvist67] L. Åqvist. *Good Samaritans, Contrary-to-Duty Imperatives, and Epistemic Obligations.* Noûs 2, pp. 361–379, 1967.

[Aqvist83] L. Åqvist. *Deontic Logic.* In "Handbook of Philosophical Logic" Vol. II, pp. 605–714. D. Gabbay & F. Guenthner (Eds.), Reidel, 1983.

[Coenen90] J. Coenen, E. van de Sluis & E. van der Velden. *Design and Implementation Aspects of Remote Procedure Calls.* Report CSN-9018, Eindhoven University of Technology.

[Cristian85] F. Cristian. *A Rigorous Approach to Fault-Tolerant Programming.* IEEE Trans. on Softw. Eng. Vol. SE-11 pp. 23–31, 1985.

[Cristian89] F. Cristian. *Exception Handling.* In "Dependability of Resilient Computers", pp. 68–97. T. Anderson (Ed.), PSP Professional Books, 1989.

[vEck82] J.A. van Eck. *A System of Temporally Relative Modal and Deontic Predicate Logic and Its Philosophical Applications.* Logique et Analyse 100, pp. 249–381, 1982.

[Hoare69] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming.* Communications of the ACM, Vol. 12 pp. 576-580, 1969.

[Khosla88] S. Khosla. *System Specification: a Deontic Approach.* Ph.D. Thesis University of London (Imperial College of Science and Technology), 1988.

[Koymans90] R. Koymans. *Specifying Real-Time Properties with Metric Temporal Logic.* Real-Time Systems 2, pp. 255–299, 1990.

[Maibaum86] T.S.E. Maibaum. *A Logic for the Formal Requirements Specification of Real-Time/Embedded Systems (FOREST).* Reader in Computing Science, Imperial College of Science and Technology, 1986.

[Meyer88] J.-J.Ch. Meyer. *Using Programming Concepts in Deontic Reasoning.* Report IR-161 Free University Amsterdam, 1988.

[Schepers91] H. Schepers. *Terminology and Paradigms for Fault Tolerance.* Report CSN-9108, Eindhoven University of Technology, 1991.

[vWright71] G.H. von Wright. *A New System of Deontic Logic.* In "Deontic Logic: Introductory and Systematic Readings", pp. 105–120. R. Hilpinen (Ed.), Reidel 1971.

[vWright81] G.H. von Wright. *Problems and Prospects of Deontic Logic: a Survey.* In "Modern Logic — a Survey: Historical, Philosophical, and Mathematical Aspects of Modern Logic and Its Applications", pp. 399–423. E. Agazzi (Ed.), Reidel 1981.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.

91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.