Annals of Mathematics and Artificial Intelligence 9(1993)133–150

Top-down development of layered fault tolerant systems and its problems – a deontic perspective

J. Coenen*

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

Abstract

Although the top-down development paradigm has successfully been applied to master the complexity of large systems, it has not yet been accepted as a useful paradigm for fault tolerant system design. This is mainly due to a problem that is sometimes referred to as the "lazy programmers" paradox. The "lazy programmer" paradox was already present and solved in top-down development methods for non-critical systems. However, the problem has re-appeared in an even more serious variant for critical systems. A few "toy" examples concerning exception handling in an Ada-like language are used to explain and illustrate the paradox. One possible solution to the problem is to use a specification language in which one can express that certain behaviours of a system are preferred over others. This paper proposes deontic logic as such a specification language. Therefore, a short and rather informal introduction to deontic logic is included. A non-trivial example is included to illustrate how deontic logic can be used to solve the "lazy programmer" paradox.

1. Introduction

As computing systems are used more often for critical applications, the importance of formal design methods for fault tolerant systems becomes more apparent (cf. [12]). Such design methods should provide not only formal specification and verification methods, but also a *design methodology* which supports the structuring of the system under development *and* the development process itself. Formal methods that meet these requirements adopt the top-down development paradigm. Top-down development methods incorporate some refinement method which is used to gradually transform a high-level abstract specification into a low-level concrete implementation.

*Supported by NWO/SION Project 612-316-022: "Fault Tolerance: Paradigms, Models, Logics, Construction".

Each transformation step creates a new layer beneath the previously generated layers of the system, hence the name layered systems. One of the earliest descriptions of a layered system can be found in [6].

To overcome the complexity of its design, a fault tolerant system may, like most complex systems, be 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 [2], see fig. 1. The arc directed from "exceptional

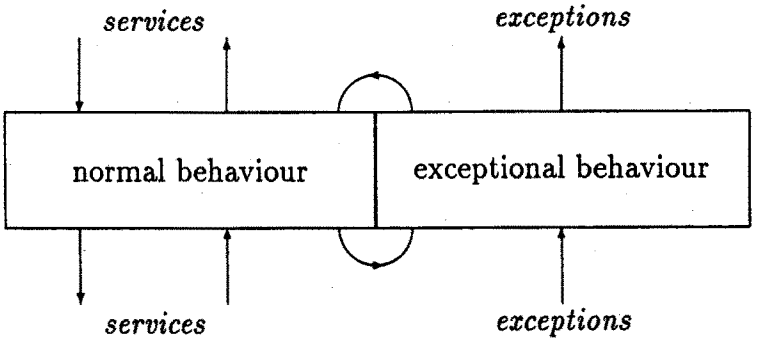

Fig. 1. Layer viewed as an ideal fault tolerant component.

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 "exceptional 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). Notice that in order to achieve a layered structure as described above, it must be possible to program a deliberately raised exception.

Any formal method that supports top-down development of layered fault tolerant systems has to solve the following two problems. Firstly, the method must provide a formal language to reason over faults and their effects. For example, Hoare's proof system as it was presented in [9] cannot deal with fault tolerance, because in this proof system, a program is considered correct if it behaves according to its specification under the assumption of a faultless execution mechanism.

In [5], Cristian extended Hoare's logic to deal with exceptions. However, 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 is a particular case of the second problem that has to be solved in any top-down development method for fault tolerant programs. The more general case of this problem is referred to as the "lazy programmer" paradox, and will be discussed in more detail in section 4.

This paper is a first step towards a deontic specification language for fault tolerant systems. It does not include a semantic model, nor does it include a complete proof theory. It merely discusses and illustrates the problems encountered when specifying the operations of fault tolerant systems when adapting a top-down development strategy. This is unlike the work in [11] where (monadic) deontic logic is reduced to dynamic logic, thereby obtaining a logic for specifying the behaviour of programs without considering faults.

The merits of a dyadic deontic specification language is that it is possible to distinguish the behaviour in a perfect world (i.e. a computation without faults) from the one (preferred) in a less than perfect world. For example, if a program should satisfy a property φ but due to some fault it does not, we can specify a property ψ it should satisfy instead. Using dyadic deontic logic, this can be specified as follows:

$$O\varphi \wedge (\neg\varphi)O\psi.$$

The conjunct $O\varphi$ is used instead of simply φ, because φ is not always satisfied but it *ought* to be if possible. The second conjunct specifies that it φ is not satisfied, then ψ ought to be satisfied instead. If one would replace the second conjunct by an implication $(\neg\varphi) \rightarrow \psi$, the program that satisfies $\neg\varphi \wedge \psi$ would be a correct implementation, which was not intended. Replacing $(\neg\varphi)O\psi$ by $O(\neg\varphi \rightarrow \psi)$ or $\neg\varphi \rightarrow O\psi$ causes similar problems (see [8]).

The remainder of this paper 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 deontic logic specification language. Section 3 introduces deontic logic. The "lazy programmer" paradox is discussed in somewhat more detail in section 4. Section 5 includes an informal description of a non-trivial fault tolerance system. The application of deontic logic as a specification language to solve the "lazy programmer" paradox is illustrated in section 6 by specifying part of the example outlined in section 5. Finally, section 7 contains a comparison with related work and some suggestions for future work.

2. Program notation

In this section, a small subset of an Ada-like "programming" language [1], called $\mathcal{P}rog$, is defined. This programming language is also used in section 5 to describe some of the 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

$$S ::= \textbf{null} \mid x := exp \mid \textbf{raise } exc \mid \textbf{begin } S \textbf{ end} \mid S_1; S_2$$
$$\mid \textbf{if } b \textbf{ then } S \textbf{ fi} \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } S$$
$$\mid \textbf{begin } S_0 \textbf{ exception when } exc_1 \Rightarrow S_1 \ldots \textbf{when } exc_k \Rightarrow S_k \textbf{ end}$$

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 an 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 program S_1 and S_2. First S_1 is executed, and if S_1 terminates successfully, then S_2 is 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 construct **if** b **then** S **fi** is an abbreviation of **if** b **then** S **else null fi**.
- 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 is not 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 fig. 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 than or equal to K, and sets x to zero in case N is larger than K.

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

Fig. 2. Running examples.

3. Deontic logic

The specification language combines deontic logic with first-order predicate logic, and is inspired by the logic used in [7]. A systematic introduction to deontic logic in general is given in [4]. The basic modality of the deontic logic used in this paper is the *dyadic obligation* $\varphi O \psi$. A more philosophical motivation of dyadic deontic logic can be found in [14, 15]. The first-order predicates in the specification language are used to quantify over logical variables only.

Assume that the following sets are defined:

- $\mathfrak{E}xpr'$, the extended set of expressions over program variables, which may be decorated with a prime. Thus, $\mathfrak{E}xpr \subset \mathfrak{E}xpr'$.

- $\mathcal{L}var$, 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.

Given the above sets, 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 $g \in \mathcal{L}var$)

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

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

$$O\varphi \triangleq \mathbf{true}\, O\varphi,$$

$$\varphi F \psi \triangleq \varphi O \neg\psi, \qquad F\varphi \triangleq \mathbf{true}\, F\varphi,$$

$$\varphi P \psi \triangleq \neg(\varphi O \neg\psi), \quad P\varphi \triangleq \mathbf{true}\, P\varphi.$$

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.

A formula with primed and unprimed variables specifies a relation between the initial and final states of a program. Hence, it cannot distinguish between the individual actions of a program. The primed variables provide the specification language with the dynamic aspect needed to reason about programs. For instance, $x = x' + 1$ specifies an action that increases the value of program variable x by one.

Below, two standard derivation rules of deontic logic are given (see, for example, [4]):

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

The axioms below are more typical for the application discussed in the introduction. The first two are still quite common axioms, which 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 not, of course, intended to be a complete axiomatization. The axiomatization of the logic itself is part of ongoing research in which there are still many questions to be settled.

The most characteristic difference between the deontic logic defined above and the ones that can be found in the literature about system specification is that the above logic includes *dyadic* modalities. For example, Khosla [10] 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 [10] 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 [3]).

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

$$O(x' \neq 0) \wedge O(x' \neq 0 \rightarrow y = 1/x') \wedge (x' = 0 \rightarrow O(y = 0)). \tag{1}$$

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. Using the above axioms and proof rules only it is possible to derive $O(y = 1/x')$ from $O(x' \neq 0)$ and $O(x' \neq 0 \rightarrow y = 1/x')$, and $O(y = 0)$ from $x' = 0$ and $(x' = 0 \rightarrow O(y = 0))$. The problem is that the monadic modalities refer to perfect worlds only, which may lead to a conflict of duties once one finds oneself 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.

Of course, one might argue that if in the above specification $O(x' \neq 0 \rightarrow y = 1/x')$ is replaced by $x' \neq 0 \rightarrow O(y = 1/x')$ or $(x' = 0 \rightarrow O(y = 0))$ by $O(x' = 0 \rightarrow y = 0)$, then there is no problem if x is initially zero. The philisiphical objections to do so (see, e.g. [8]) might be irrelevant to system specifications. The specification of fault tolerant systems is a difficult task even if one does not have to bother with such subtle paradoxes. Therefore, it is preferable to use a specification language in which such paradoxes can easily be avoided.

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 $x' = 0$ because there is no detachment rule which allows one to derive $O(y = 0)$ from $x' = 0$ and $(x' = 0)O(y = 0)$. Notice that in the intuitive meaning of $\varphi O \psi$ it is implicit 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 in section 4.

In this paper, the use of dyadic modalities is restricted 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 $\delta(exc)O\psi$. However, it is permitted to have predicates on the left side also. This is illustrated in the next section.

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 fig. 3.

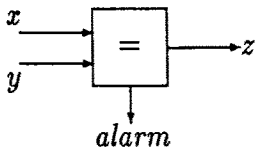

Fig. 3. Comparator.

The comparator may be specified by

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

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**.

4. The "lazy programmer" paradox

Lazy programmers were already a problem in Hoare's logic because it is a *partial* correctness formalism, which means that it is not possible to specify that a program must terminate. Hence, each *divergent* program is a correct implementation of every specification. This particular version of the "lazy programmer" paradox is solved in *total* correctness formalisms in which one can specify that a program must terminate.

The particular formulation of the "lazy programmer" paradox for fault tolerance has a striking similarity with the "Good Samaritan" paradox (see [3]). 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 fig. 2 serve to illustrate the previous discussion. Consider the following naive specification for a program that is to compute the factorial of N:

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

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 fig. 2. Because program *b* ought to raise the exception *ovg* and set x to zero, it satisfies

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

Unfortunately, (3) specifies a correct implementation of (2), which can formally be proved as follows:

1. $\vdash (x = 0 \wedge \delta(ovf)) \rightarrow ((\neg \delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0))$;
2. $\vdash O((x = 0 \wedge \delta(ovf)) \rightarrow ((\neg \delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)))$;
3. $\vdash O((x = 0 \wedge \delta(ovf)) \rightarrow ((\neg \delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)))$
   $\rightarrow (O(x = 0 \wedge \delta(ovf)) \rightarrow O((\neg \delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0)))$;
4. $\vdash O(x = 0 \wedge \delta(ovf)) \rightarrow O((\neg \delta(ovf) \rightarrow x = N!) \wedge (\delta(ovf) \rightarrow x = 0))$.

The individual steps of the above derivation are justified as follows:

1. is a valid predicate logic formula;
2. is obtained by the application of the Necessitation rule to 1;
3. is an instance of the first axiom listed in section 3;
4. is obtained by applying Modus Ponens to 2 and 3.

Using dyadic modalities, one can specify program *a* as follows:

$$O(x = N!) \wedge \delta(ovf)O(x = 0). \tag{4}$$

This specification expresses that it is preferred to set x equal to $N!$, and if this is not possible due to an overflow, x ought to be zero. Provided that the axiomatization of the deontic logic does not allow one to derive (4) from (3), it is not possible to

prove that program b is a correct implementation of (4). As a matter of fact, program b can be excluded more explicitly by adding the conjunct $F\delta(ovf)$ to (4). 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 fig. 2 is intuitively a correct implementation of (4). The specification of program c is, however, as follows:

$$O((N \leq K \wedge x = N!) \vee (\neg N \leq K \wedge x = 0)). \tag{5}$$

The only way to prove that (5) specifies a correct implementation, i.e. to prove that (5) implies (4), is by making the knowledge about the hardware limitation explicit. For instance, by including the following axioms:

$$\vdash O(N \leq K), \tag{6}$$

$$\vdash O(\neg N \leq K \rightarrow \varphi) \rightarrow \delta(ovf)O\varphi. \tag{7}$$

Axiom (6) expresses that it ought to be the case that $N \leq K$. Axiom (7) 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 (5) is equivalent to

$$O(N \leq K \rightarrow x = N!) \wedge O(\neg N \leq K \rightarrow x = 0). \tag{8}$$

This can be proved as follows. Let ψ, φ_1, and φ_2 be defined by

$$\psi \triangleq (N \leq K \wedge x = N!) \vee (\neg N \leq K \wedge x = 0),$$

$$\varphi_1 \triangleq N \leq K \rightarrow x = N!,$$

$$\varphi_2 \triangleq \neg N \leq K \rightarrow x = 0.$$

We give the major steps of the derivation of (8) from (5):

$\vdash \psi \leftrightarrow (\varphi_1 \wedge \varphi_2)$,	Predicate logic.
$\vdash O(\psi \leftrightarrow (\varphi_1 \wedge \varphi_2))$,	Necessitation.
$\vdash O\psi \leftrightarrow O(\varphi_1 \wedge \varphi_2)$,	Axioms and Modus Ponens.
$\vdash O\psi \leftrightarrow (O\varphi_1 \wedge O\varphi_2)$,	Axioms and Modus Ponens.

Because it is easily seen that (8) implies

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

we may conclude from (6) and (7) that program c is a correct implementation of (4), *provided* the above assumptions hold. Thus, only if the hardware limitations are such that the axioms are justified does the above reasoning hold.

It is possible to think of clever variations on the programs in fig. 2, e.g. the ones in fig. 4, for which the correct arguments to accept or reject them as correct implementations of (4) are not so easily found. For example, program d should be rejected, but just including $F\delta(ovf)$ in the specification would also exclude program

```
(d) begin x := N!; raise ovf exception when ovf ⇒ x := 0 end
```

```
(e) begin
      if N ≤ K
        then x := N!
        else raise ovf
      fi
    exception
      when ovf ⇒ x := 0
    end
```

Fig. 4. The lazy programmer strikes back.[1)]

e which might be acceptable. 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, if provided with adequate semantics, can be expressive enough to distinguish deliberate errors from unintentional ones.

5. 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 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

[1)] Or is it a too diligent programmer?

stable storage and focuses on the implementation of the read operation. A more complete description of the stable storage described below is given in [13].

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.

5.1. 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 section might include a recursive call of *read_sector*.

The *logical_disk_crash* exception is handles simply by deleting the corresponding disk number from the set *operational_disks*. If the reliable disk layer runs out of operational logical disks, it raises a *reliable_disk_crash* exception. See also fig. 5.

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.

5.2. 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, in its turn, achieves a

```
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
      fi
  end
end
```

Fig. 5. *read_sector*.

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 fig. 6. The logical disk layer simple calls the *read_physical_disk* operation with the converted address. If

```
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
```

Fig. 6. *read_logical_disk*.

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* is raised.

5.3. 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 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 the faults like damaged disk drives, etc. are not considered relevant in this example. The *read_physical_sector* operation is listed in fig. 7.

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

Fig. 7. *read_physical_sector.*

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

6. Deontic logic specifications 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, for example, 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 (fig. 7) contains only monadic modalities. The *read_physical_sector* operation (for physical disk i) is specified by

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

The first conjunct expresses that if the underlying execution mechanism functions correctly, then *sector* is set equal to the contents of physical disk i at location *sectornr*. 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 instance:

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

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 did not pass the cyclic redundancy check.

Notice that if the second conjunt in the specification of *read_physical_sector* is replaced by

$$\delta(\textit{invalid\_crc}) \rightarrow O \neg \textit{cyc\_red\_check}(\textit{sector}'),$$

then an *invalid_crc* exception ensures that *sector* did not pass the cyclic redundancy check regardless of whether the exception was raised by the *read_physical_sector* operation itself or by another operation.

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 (fig. 6). The *read_logical_sector* operation (for logical disk i) is specified by

$$O(\textit{sector} = \textit{logical\_disk}_i'(\textit{address}')) \wedge$$
$$\delta(\textit{invalid\_crc})O((\delta(\textit{logical\_disk\_crash}) \wedge \textit{spare\_sectors}_i' = \emptyset) \vee$$
$$(\delta(\textit{logical\_sector\_lost}) \wedge \textit{spare\_sectors}_i' \neq \emptyset)).$$

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 are no 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 $F(\delta(\textit{logical\_disk\_crash}) \wedge \delta(\textit{logical\_sector\_lost}))$, 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 (fig. 5) of the reliable disk layer is specified by

$$O\exists_i(i \in \textit{operational\_disks}' \wedge \textit{sector} = \textit{logical\_disk}_i'(\textit{address}'))$$
$$\wedge\ \delta(\textit{logical\_disk\_crash})O(\delta(\textit{reliable\_disk\_crash}) \rightarrow \textit{operational\_disks} = \emptyset).$$

Suppose that it is forbidden to raise the *reliable_disk_crash* exception deliberately, which may be accomplished by adding the conjuct $F\delta(\textit{reliable\_disk\_crash})$ to the specification above. Then it follows that a *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 does not have the appropriate information.

7. Conclusions

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 of the system explicit. The possibility to express the preference of some behaviours 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 to system specification before, for example in [10], the application to fault tolerant systems seems to be new, which partly explains the differences between the specification language used in this paper and those appearing in the literature about system specification.

The deontic logic described in this paper differs from the deontic logics for system specification in the existing literature mainly in two ways. Firstly, the logic in this paper is a *dyadic* deontic logic, whereas the logics in, for example, [11] and [10], are *monadic* deontic logics. Secondly, primed and unprimed variables are used to capture the dynamic aspect of programs in the specification language, whereas Meyer [11] and Khosla [10] 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 favour of a more important one if a fault occurs. This corresponds in a natural way with deontic logic specification of the format $\varphi_1 O(y_1 \wedge \ldots \varphi_n O \psi_n$ that specifies the behaviour of 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 4. Also, although the examples used to illustrate this paradox may be regarded as "toy" examples, it should be evident from the example in section 5 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 primes variables or only unprimed variables. Action predicates and predicates with both primed and unprimed variables. 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 [7].

The next step which must be taken is the definition of an adequate formal semantics for the deontic logic discussed in this paper. A first study shows that a Kripke semantics can be obtained by introducing residuals of reachability relations.

Acknowledgements

The author is grateful to Henk Schepers for providing the stable storage example, and Tijn Borghuis and Wim Koole for many helpful discussions.

References

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

[2] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, 2nd revised ed. (Springer, 1990).

[3] L. Åqvist, Good Samaritans, contrary-to-duty imperatives, and epistemic obligations, Noûs 2(1967) 361–379.

[4] L. Åqvist, Deontic logic, in: *Handbook of Philosophical Logic*, Vol. II, ed. D. Gabbay and F. Guenthner (Reidel, 1983) pp. 605–714.

[5] F. Cristian, A rigorous approach to fault-tolerant programming, IEEE Trans. Software Eng. SE-11 (1985)23–31.

[6] E.W. Dijkstra, The structure of the "THE"-multiprogramming system, Commun. ACM 11(1968) 341–346.

[7] J.A. van Eck, A system of temporally relative modal and deontic predicate logic and its philosophical applications, Logique et Analyse 100(1982)249–381.

[8] D. Føllesdal and R. Hilpinen, Deontic logic: an introduction, in: *Deontic Logic: Introductory and Systematic Readings*, ed. R. Hilpinen (Reidel, 1971) pp, 1–35.

[9] C.A.R. Hoare, An axiomatic base for computer programming, Commun. ACM 12(1969)576–580.

[10] S. Khosla, System specification: a deontic approach, Ph.D. Thesis, Imperial College of Science and Technology, University of London (1988).

[11] J.-J. Ch. Meyer, Using programming concepts in deontic reasoning, Report IR-161, Free University Amsterdam (1988).

[12] W.-P. de Roever, Foundations of computer science: leaving the ivory tower, Bull. EATCS 44(1991) 455–492.

[13] H. Schepers, Terminology and paradigms for fault tolerance, Report CSN-9108, Eindhoven University of Technology (1991); to appear in: *Formal Techniques in Real-Time and Fault Tolerant Systems*, ed. J. Vytopil (Kluwer, 1993).

[14] G.H. von Wright, A new system of deontic logic, in: *Deontic Logic: Introductory and Systematic Readings*, ed. R. Hilpinen (Reidel, 1971) pp. 105–120.

[15] 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*, ed. E. Agazzi (Reidel, 1981) pp. 399–423.