Using Relative Refinement for Fault Tolerance *

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

Institut für Informatik und Praktische Mathematik II
Preußerstr. 1-9
Christian-Albrechts-Universität zu Kiel
D-2300 Kiel 1, Germany

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

1 Introduction

Current formal methods are far from solving the problems in software development. The simplest view of the formal paradigm is that one starts with a formal specification and subsequently decomposes this specification in subspecifications which composed together form a correct refinement. These subspecifications are decomposed into “finer” subspecifications. This refinement process is continued until one gets subspecifications for which an implementation can easily be given. This view is too idealistic in a number of respects. First of all, most specifications of software are wrong (certainly most informal ones unless they have been formally analyzed) and contain inconsistencies [9]. Secondly, even if a formal specification is produced, this is only after a number of iteration steps because writing a correct specification is a process whose difficulty is comparable with that of producing a correct implementation, and should therefore be structured, resulting in a number of increasingly less abstract layers with specifications which tend to increase in detail (and therefore become less readable [8]). Thirdly, even an incorrect refinement step may be

* Partially supported by ESPRIT/BRA project 6021 (REACT)
** E-mail: ac@informatik.uni-kiel.dbp.de
*** E-mail: wpr@informatik.uni-kiel.dbp.de
useful in the sense that from this incorrect refinement step one can easier derive the correct refinement step. This is especially the case with intricate algorithms such as those concerning specific strategies for solving the mutual exclusion problem. An interesting illustration of this third view is provided by E.W. Dijkstra's "Tutorial on the split binary semaphore" [5] in which he solves the readers/writers problem by subsequently improving incorrect refinement steps till they are correct. If this master of style prefers to approximate and finally arrive at his correct solution using formally incorrect intermediate stages, one certainly expects that a formally correct development process for that paradigm is difficult to find! The strategy described in [5] is necessarily informal, reflecting the state of the art in 1979. We have formalized this strategy in [3].

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

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