Recovery Oriented Programming*
(Extended Abstract)

Olga Brukman and Shlomi Dolev

Department of Computer Science, Ben-Gurion University of the Negev,
Beer-Sheva, 84105, Israel
{brukman, dolev}@cs.bgu.ac.il

Abstract. Writing a perfectly correct code is a challenging and a nearly impossible task. In this work we suggest the *recovery oriented programming* paradigm in order to cope with eventual Byzantine programs. The program specification composer enforces the program specifications (both the safety and the liveness properties) in run time using predicates over input and output variables. The component programmer will use these variables in the program implementation. We suggest using the "sand-box" approach in which every instruction of the program that changes a specification variable, is executed first with temporary variables and that is in order to avoid execution of an instruction that violates the specifications. In addition, external monitoring is used for coping with transient faults and for ensuring convergence to a legal state. The implementation of these ideas includes the definition of new instructions in the programming language with the purpose of allowing addition of predicates and recovery actions. We suggest a design for a tool that extends the Java programming language. In addition to that, we provide a correctness proof scheme for proving that the code combined with the predicates and the recovery actions is self-stabilizing and, under the restartability assumption, eventually fulfills its specifications.

Keywords: self-stabilization, autonomic computing.

1 Introduction

Writing a perfectly correct code is a challenging and a nearly impossible task. Paradigms, tools and programming environments, including structured programming, object oriented programming, design patterns and others, were created to assist the programmer in writing a manageable and correct code. Tools that ensure testing during the programming phase complement the above effort [2,5,13]. Still, in many cases the program specifications are not fulfilled [19] – a situation that can cause a great deal of damage. In our previous work on self-stabilizing autonomic recoverer [4], we suggested a formal framework for the recovery oriented paradigm [20]. The suggested approach fitted existing (black box) software

* Partially supported by the Lynn and William Frankel Center for Computer Sciences, by Deutsche Telecom grant, by IBM faculty award, the Israeli Ministry of Science, and the Rita Altura Trust Chair in Computer Sciences.

A.K. Datta and M. Gradinariu (Eds.): SSS 2006, LNCS 4280, pp. 152–168, 2006.

packages, which resulted in high overhead, as each IO action had to be intercepted to detect a faulty state.

Fault tolerance paradigms and eventual Byzantine software. Self-stabilization [9] is a strong fault tolerance property for systems that ensures automatic recovery once faults stop occurring. A self-stabilizing system is able to start from any possible configuration in which processors, processes, communication links, communication buffers and any other process-related components are in an arbitrary state (e.g., arbitrary variable values, arbitrary program counter). The designer can only assume that the system's programs are executed. Based on that assumption, he or she proves that the system converges to a legal state, i.e., to a state in which the system satisfies its specifications. If the system is started from a legal initial state, the execution will ensure that the system remains in a legal state. This is called "closure property". In case the system is started in an illegal state (possibly after encountering transient faults), the execution of the self-stabilizing program will ensure that eventually (within a finite number of steps) a legal state will be reached. This is called "convergence property". Again, once the program reaches a legal state, it will continue running and will remain in a legal state until a (transient) fault reoccurs. A self-stabilizing algorithm never terminates. The algorithm does not necessarily need to "identify" the failure occurrence and to recover, but rather it continues to be executed and brings the system into a legal state. The time complexity of a self-stabilizing algorithm is the number of steps required for an algorithm started in an arbitrary state to converge to a legal state. Note that when all the processors execute incorrect programs (programs with bugs), they may exhibit any kind of behavior and, therefore, there is no guarantee for convergence.

The Byzantine fault model [16,10] is used for modeling arbitrary (in fact, malicious) behavior of a program that contains bugs and, therefore, does not obey the specifications. Systems that tolerate a bounded number of Byzantine processors (typically, less than one third of the processors can be Byzantine) were designed and proved to be correct.

Research on self-stabilizing systems and systems that model faults through Byzantine behavior has not yet provided solutions for systems in which software packages contain bugs with a very high probability. We observe that software packages usually function as required for a long period of time after being started from an initial state. The initial correct behavior can be attributed to the testing done by the software manufacturer. Therefore, programs started from an initial state run correctly for bounded length executions. System administrators and users occasionally restart such software in order to cope with failures.

Our contribution. Our goal is to incorporate the program predicates and the recovery actions provided by the program specification composer with the program code so that the predicates violation would be detected and avoided during run-time. Both predicates and recovery actions will be an integral part of the program specification. In this scope we are interested in a new programming