Recovery Oriented Programming
Olga Brukman, Shlomi Dolev∗
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
{brueman, dolev}@cs.bgu.ac.il
Marcelo Sihman
Department of Computer Science, Bar-Ilan University, Ramat Gan, 52900, Israel
{sihman@cs.biu.ac.il

Computerized management of critical systems makes the issues of correctness and faultless flow of long-lived and continuously-running programs extremely important. Complex systems cannot be fully verified because their verification may require an unreasonable amount of time and space. The software industry tests software products extensively in order to eliminate bugs as much as possible. Normally, software is tested by executing a set of large, but length-bounded and non-exhaustive scenarios starting from a predefined initial state while each scenario is defined by a set of input/output sequences. Undesired and unplanned behavior (bug) may occur due to scenarios that were not tested prior to the software release. Software malfunctions may cause damage that can outweigh the software cost. Keeping all this in mind, a consumer of a critical system would like to have a warranty that such a system will operate properly.

We suggest to model software package flaws (bugs) by assuming eventual Byzantine behavior of packages [1]. We assume that if a program is started in a predefined initial state, it will exhibit a legal behavior for a period of time but may eventually become Byzantine. This behavior pattern could be attributed to the fact that the manufacturer had performed sufficient package tests for limited time scenarios. Therefore, rebooting is a legitimate practical recovery action that can be used in such a case [6].

In our framework each critical process and subsystem (collection of processes) has a dedicated monitor. Monitors are generic and are given a process (subsystem) specific information in the form of invariants on a state of a process (or a subsystem) and recovery actions. Each invariant describes some safety or liveness requirement. A process is regarded as a black box and its monitor follows the process state by recording the process IO calls. The monitor continuously checks the given invariants and triggers the respective recovery action when needed. The monitor is self-stabilizing [3], starting in an arbitrary state it converges to legal behavior. Namely as soon as it accumulates the correct history log and completes full round of checks, the monitored process can detect safety and liveness violations. The self-stabilizing processor [4] and self-stabilizing operating system [5] support existence and correctness of the monitors. The combination of self-stabilizing operating system and monitors form our self-stabilizing recovery framework.

A complimentary technique is suggested for writing recovery oriented code (as opposed to given black box package), the Recovery Oriented Programming paradigm is suggested in [2], where the program specifier may enforce the program specifications during run time. The specifier includes the specifications description into the program code and the compiler automatically produces code that verifies and enforces its specifications in run time. Specifications are given in a form of invariants and recovery actions, where invariants are logic expressions on the program variables and the recovery actions are functions that need to be executed upon invariants unsatisfiability. Wherever an invariant variable assignment takes place, the compiler inserts code for: checking the invariant consistency, and for calling the respective recovery action if the consistency check fails. We want to avoid the execution of instruction that violates the specifications. We suggest to use the “sand box” approach: every instruction of the program that changes a variable of the specifications will be executed first using temporary variables. If the invariant is not violated, the instruction will be executed. Otherwise, the execution is stalled and a recovery action takes place.

When executing portions of the code with no invariant variables updates, the program can encounter an inconsistent state. Thus, external self-stabilizing monitoring is required. The external monitor checks the invariants periodically for safety and liveness and triggers recovery actions if required.

References

∗Partially supported by the Lynn and William Frankel Center for Computer Sciences, IBM faculty award, NSF grant 0098305, the Israeli ministry of defense, the Israeli ministry of Trade and Industry, and the Rita Altura trust chair in computer sciences.