PhD Proposal: Automatic Detection of Software Errors

Rui Maranhão
Software Engineering Lab.
Faculty of Engineering
University of Porto
rui@computer.org — www.fe.up.pt/~rma

1. Introduction

From the international monetary system and power plants to TVs and cell phones, almost everything runs on (embedded) software. In office buildings, the elevators, the lights, the water, the air conditioning are all controlled by software. In cars, the ignition timing, the air bag, and the infotainment system is controlled by software. Almost every written communication that's more complicated than a postcard depends on software.

Software is currently measured in terms of millions of lines of code, the complexity of which makes software development cost-intensive and error-prone. Software developers already spend approximately 80% of development costs on identifying and correcting defects [3], and yet few products of any type other than software are shipped with such high levels of faults.

With the pervasion of software at all levels of society the impact of software defects is becoming increasingly problematic. A 2002 landmark study on this problem [3] estimated that software defects cost the US economy some $60 billion per year (around 0.6% GDP). High-profile incidents caused by malfunctioning software include several Mars landers/orbiters, Ariane 5, USS Yorktown, Denver Airport baggage-handling system, LA Airport air-traffic control system, etc.

Apart from economic loss during deployment, software defects also account for an important part of the development costs. With respect to development costs residual defect density is very much dependent on the effort invested in testing, defect finding and fixing. For embedded software development cost estimates are quoted around $15-30 per line. The US Defense Dept. and Carnegie-Mellon's Software Engineering Institute estimate that defect density ranges from 5 to 15 bugs per KLOC. In the defense realm costs can range up to $100, while for highly critical applications, such as the Space Shuttle, the cost per line approximates $1,000. This higher effort results in significantly lower defect density. For example, the NASA Shuttle code (420 KLOC) only had 17 faults [20]. Although not all defects manifest themselves as fatal (NASA JPL data suggest that only some 20% are fatal), typically all defects have to be diagnosed in order to locate the most fatal ones.

Fault diagnosis is a major cost factor in both the development and the deployment of software. Based on the assumption that localizing defects takes on average 75 minutes per defect, at the development stage, the above data implies that fault diagnosis already accounts for more than $3,000,000/MLOC. At the deployment stage the costs of diagnosing and fixing residual defects is not known. However, a significant fraction of the annual $60 billion economic loss is attributed to (1) system down time (or degraded performance) while the problems are being diagnosed and repaired, and the effort spent on diagnosis and repair (in some cases diagnosis accounts for 60% of the downtime). Given the fact that the economic loss is in the same order of total development cost, and the fact that diagnosis and repair is usually more costly when applied later in the software life cycle, it is therefore reasonable to assume that diagnosis costs at deployment phase may even exceed the costs at the development stage.

The above data clearly suggests that even a large investment in development (e.g., $1,000 per line of code (amounting to a staggering $1,000,000,000 per MLOC) does not yield zero-defect code, despite recent advances in software engineering (e.g., formal methods). In fact, for many applications the optimum investment set point (minimizing development and deployment costs due to defects) will correspond to a much higher residual defect density. For instance, in consumer electronics it is accepted practice that time-to-market and developer (testing) cost often take precedence over low-defect-density software. The fact that in many domains it is more pragmatic to simply cope with failures, rather than invest asymptotic effort at the development phase, is increasingly being recognized. In this new paradigm fault diagnosis is a central focus, since providing the system (and/or operator) with the crucial insight what part(s) of the system cause(d) the failure(s) is key to timely recovery and/or repair. As the earlier data suggests that a large fraction of the costs incurred at development and deployment is related to fault diagnosis, investments in better fault diagnosis (less effort, less time, more precision) have a dramatic effect on development cost and costs incurred at deployment due to residual defects.

2. Fault Diagnosis

Fault diagnosis aims at identifying the root cause(s) of system malfunction based on external observations. For the purpose of diagnosis the system is broken down in a number of components (subsystems) in terms of which the diagnosis is expressed. The diagnostic process takes observations (values, events) as input, and produces a list of possible diagnoses, where each single diagnosis may involve multiple component faults to be a likely explanation for the observations (e.g., observations can be explained by the combination of component 13 and component 99 at fault).

Due to the fact that the observations are typically limited in time and space (e.g., lack of measurement time, lack of sensors), and the fact that information on system behavior is limited, the size of the list of possible diagnoses can be large, while only one diagnosis represents the true system fault state. Next to computational complexity, this limited diagnostic accuracy is a key performance metric. Fault diagnosis is typically triggered by the detection of an error (or errors), i.e., the deviation of a system value from its nominal value (pass/fail information), also referred to as a symptom.

Traditionally, in diagnosis of software faults (defects, bugs) a symptom-based approach has been adopted, which is based on the existence of a mapping from symptoms to diagnoses. The implementation of this mapping can range from a symptom table to an expert system, and is typically compiled by humans. While such a mapping offers split-second diagnosis, the mapping is usually not complete, introducing the risk of bad diagnostic performance in unforeseen situations. Moreover, deriving the mapping is an intensive and error-prone process given its complexity. Especially in software, implementing symptom-based diagnosis is virtually impossible, as illustrated by the fact that even simple exception handling (where diagnosing the root-cause of the exception is typically even left out) requires significant coding effort. The large cost associated with traditional diagnosis has lead to a number of contemporary approaches, which aim for completeness and automation. The state-of-the-art techniques in software fault diagnosis are model-based diagnosis (MBD, based on reasoning) and spectrum-based fault localization (SFL, based on statistics) (due to lack of space, no detailed information is give ; refer to [1] for detailed information).

3. Work Package: Automatic Error Detection

While SFL's inherent diagnostic precision is lower than MBD, related work as well as the abovementioned industrial experience suggests that SFL is highly usable in the software domain [1]. Moreover, the fact that in SFL no compositional modeling (behavior, structure) is required makes SFL a particularly attractive candidate for fault diagnosis technique that requires minimal modeling effort, i.e., that aims at maximum automation of the entire fault diagnosis process. We believe the latter to be a crucial success factor for adoption of (computerized) fault diagnosis in an industrial context.

At development time information whether a run has passed/failed is typically derived from existing test oracles. However, at deployment time no information exists on nominal system behavior unless models are introduced. While at a high behavioral level the use of models (and/or specifications) cannot be avoided, much information on (impending) errors can be inferred from low-level errors within the code. For instance, an out-of-bound variable value (e.g., NULL pointer, segmentation violation, index overflow, response timeout) is a clear indication that a fault has been triggered, while the downstream effect may not yet be manifest (or may even go unnoticed). Recent research on error detection as well as preliminary research by the advisor [2] has indicated that automatic instrumentation with simple, generic invariants provides useful pass/fail information to SFL such that diagnostic precision can be achieved comparable to the use of test oracles. Eliminating the need for (error) modeling paves the way for fully automatic software diagnosis.

In this work package we propose to investigate the use of various generic invariants in the value and time domain, their effect on diagnostic precision, their relation with existing test oracles, and their run-time overhead, in particular, the density required (trading off overhead against precision). Evaluation will be based on standard test suites, while the associated input dataset is extended (using, e.g., genetic algorithms) to generate more observations.

4. References

[1] R. Abreu, **Spectrum-based Fault Localization in Embedded Software**. PhD Thesis, Delft University of Technology, November 2009.
[2] R. Abreu, A. Gonzalez, P. Zoeteweij, and A.J.C. van Gemund, **Automatic Software Fault Localization using Generic Program Invariants**. In Proc. of Annual ACM Symposium on Applied Computing (SAC'08) - Software Engineering Track, pp. 712–717, Fortaleza, Brazil, March 2008.
[3] RTI Health, Social, and Economics Research, **The economic impacts of inadequate infrastructure for software testing**. Planning Report 02-3, RTI Project Number 7007.011, May 2002.

Note: The work will be in close collaboration with Prof.dr.ir. A.J.C. van Gemund, Delft University of Technology, the Netherlands.