Software Health Management: A Short Review of Challenges and Existing Techniques
—Extended Abstract—

Knot Pipatsrisawat, Adnan Darwiche
UCLA
Email: thammakn@cs.ucla.edu
Email: darwiche@cs.ucla.edu

Ole J. Mengshoel
CMU SV
NASA ARC
Email: Ole.J.Mengshoel@nasa.gov

Johann Schumann
RIACS/USRA
NASA ARC
Email: Johann.M.Schumann@nasa.gov

I. INTRODUCTION

Modern spacecraft (as well as most other complex mechanisms like aircraft, automobiles, and chemical plants) rely more and more on software, to a point where software failures have caused severe accidents and loss of missions. Software failures during a manned mission can cause loss of life, so there are severe requirements to make the software as safe and reliable as possible. Typically, verification and validation (V&V) has the task of making sure that all software errors are found before the software is deployed and that it always conforms to the requirements. Experience, however, shows that this gold standard of error-free software cannot be reached in practice. Even if the software alone is free of glitches, its interoperation with the hardware (e.g., with sensors or actuators) can cause problems. Unexpected operational conditions or changes in the environment may ultimately cause a software system to fail. Is there a way to surmount this problem?

In most modern aircraft and many automobiles, hardware such as central electrical, mechanical, and hydraulic components are monitored by IVHM (Integrated Vehicle Health Management) systems. These systems can recognize, isolate, and identify faults and failures, both those that already occurred as well as imminent ones. With the help of diagnostics and prognostics, appropriate mitigation strategies can be selected (replacement or repair, switch to redundant systems, etc.).

In this short paper, we discuss some challenges and promising techniques for software health management (SWHM). In particular, we identify unique challenges for preventing software failure in systems which involve both software and hardware components. We then present our classifications of techniques related to SWHM. These classifications are performed based on dimensions of interest to both developers and users of the techniques, and hopefully provide a map for dealing with software faults and failures.

II. CHALLENGES

In principle, there is no reason why software components could not be “hooked up” to an IVHM system tailored for software monitoring. Such an IVHM system could operate in a similar way to a traditional IVHM system, but would focus its attention on the software. However, there are substantial differences between physical systems and software systems. These differences call for special approaches for preventing software failure, which is the ultimate goal of SWHM. In particular, the challenges stemming from differences between physical and software systems include the following issues:

- Software errors do not develop over time, they are introduced as flaws and errors in all stages of the software life-cycle. Requirements errors, design flaws, and coding errors are just a few examples. If errors are not detected and removed during testing, they remain (dormant) in the software system and can show up during operation. Software errors also do not “go away” on their own.

- Failures in software most often occur due to problematic interoperation with the hardware. Hardware systems (and their sensors) might behave differently than expected, and thus could cause software failure. Such a different behavior could be on purpose1, by accident during development (e.g., replacement of a sensor or instrument shortly before launch [1]), as a result of a hardware failure (broken sensor cable), disabled sensor (e.g., broken capacitor on Deep Space 1’s star tracker), or gradual degradation (e.g., signal noise increases beyond the specified level and causes the software to behave erratically). Since physical systems can misbehave in various ways, it is extremely difficult to maintain software health in the presence of physical anomalies.

- In contrast to many hardware failures, which occur gradually (e.g., decrease in oil pressure due to a leak), most software failures occur instantaneously. The reason for this is that most of the software is discrete (state machines, decision logic) and usually cannot be described or reasonably approximated by continuous modeling techniques. So, systems dealing with software failure are under more pressure to predict potential failures and to

1In Ariane V several software modules from the smaller Ariane IV had been re-used. However, the range of certain sensor values was larger (due to different physical dimensions and construction), which led to an uncaught overflow error, causing the rocket to behave erratically and required its destruction.
take swift actions in order to detect and recover from failures.

- Fault detection and monitoring systems, as well as any SWHM system, are implemented as software themselves. Safety analysis has to ask: "Quis custodiet ipsos custodes?" (Juvenal) "Who guards the guardians?" This means that SWHM systems must be at least as safe and dependable as the software components they monitor.

All these differences (and commonalities) between IVHM of physical systems and software systems must be taken into account when developing novel techniques for unified software and hardware IVHM.

III. CLASSIFICATIONS OF EXISTING APPROACHES

Of course, the idea of monitoring a piece of software and reacting if something goes wrong is not new. Even basic error-handling (“if error then abort”) could be considered as an extremely simple—and usually not desirable—way of monitoring the health of a piece of software. In this section, we consider a number of software engineering techniques, which try to address issues similar to SWHM. These techniques are model-based design [17], goal-based design [8], aspect-oriented programming [16], recovery-based computing [20], software configuration management [10], software testing [4], [13], model checking [7], theorem proving [23], redundancy-based fault tolerance techniques [2], [21], checkpointing and rolling back [9], runtime monitoring [22], trace analysis [5], built-in tests [11], software rejuvenation [14], computer immunology [12], and self-healing software [15], [24]. We analyze these techniques along the following axes of concepts:

- **Software Life-cycle.** Different techniques are used during different stages of the software life-cycle. Although SWHM generally is active after code deployment, there are many tasks, which can and should be performed during earlier stages of the software life-cycle to prevent software failure during actual operation. As with humans, preventive care (i.e., finding and removing software bugs early) is an important prerequisite for an effective health management system.

- **Fault Handling.** Different approaches are supposed to deal differently with faults: there are techniques for fault prevention, fault removal, and fault tolerance [3]. Whereas design techniques primarily help prevent the occurrence of faults even before the system is built, typical V&V tasks are used to remove faults. Traditional fault tolerant approaches aim at keeping up functionality of the original software in the presence of faults (e.g., by using redundancy); this notion, however, can easily be extended to cover approaches like dynamic debugging [6] or dynamic reconfiguration [18], where the software is modified after the fault to avoid further problems.

- **FDIR.** System Health Management distinguishes its approaches into fault detection, fault isolation, and fault recovery [19]. Fault detection is the identification of the presence of fault. Fault isolation is the process of identifying the fault source and isolating it from the rest of the system. Based on the fault detection and/or fault isolation steps, fault recovery takes corrective actions to restore the system back to an operational state.

- **Automation.** Whereas several techniques can be executed fully automatically, others require a certain amount of human interaction. Although, in general, automatic processing is preferred (esp. in time-critical applications), SWHM applications with humans in the loop can be important, as such an architecture could lower the certification threshold (“the human is still making the critical decision”).

- **Resources.** The surveyed technologies require a wide spectrum of resources, both in setting up (e.g., developing a fault model) and in computational resources during the execution of the software. There is a clear trade between the capability of the health management system and the amount of CPU/memory it requires during execution of the software.

- **Completeness.** Some of the methods can provide guarantees (e.g., absence of deadlock or NULL-pointer dereference), whereas others can produce false positives or can fail to detect/manage certain faults. Again, other approaches provide statistical estimates and failure probabilities.

We use these dimensions to classify different SWHM techniques in order to provide a map for dealing with software faults and failures. Table I summarizes our classifications.

In this table, considered SWHM techniques are presented according to the phases in the software life-cycle at which they are typically utilized. The second column shows the purpose of each technique in terms of fault handling and, when appropriate, FDIR. The third column indicates the level of automation usually associated with the techniques. The forth column shows the amount of resources typically required by different techniques. Finally, the last column addresses the completeness of these approaches.

IV. CONCLUSIONS

We discussed challenges associated with different aspects of SWHM and analyzed a large number of different software engineering approaches, which can address some of the SWHM issues according to the framework discussed above. Despite the wide spectrum of available technologies, none of those addresses all requirements for an SHWM system. The most critical areas are:

- most approaches deal with faults as they occur or process them in a post-mortem fashion, but they are not able to perform any prognostic function or fault forecasting.
- many of these approaches are tailored toward discrete software, like finite state machines, statecharts, or model logic. Monitoring of continuous calculations as they, e.g., occur in guidance, navigation, and control (GN&C), are seldomly addressed.
- most of these techniques are for software and for software only. This means that their performance is weak
TABLE I
CLASSIFICATIONS OF SOFTWARE HEALTH MANAGEMENT TECHNIQUES.

<table>
<thead>
<tr>
<th>Technique</th>
<th>FDIR</th>
<th>Automation</th>
<th>Resources</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design and programming methodologies (development phase)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model-based design</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Goal-based operations</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Aspect-oriented programming</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Recovery-based computing</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Software configuration management</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Verification and Validation (V&amp;V) (testing phase)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td>fault prevention</td>
<td>N/A</td>
<td>manual, semi-automatic</td>
<td>adjustable</td>
</tr>
<tr>
<td>Simulation</td>
<td>fault prevention</td>
<td>N/A</td>
<td>automatic</td>
<td>moderate-high</td>
</tr>
<tr>
<td>Debugging</td>
<td>fault prevention</td>
<td>N/A</td>
<td>semi-automatic</td>
<td>varied</td>
</tr>
<tr>
<td>Numerical analysis</td>
<td>fault prevention</td>
<td>N/A</td>
<td>manual</td>
<td>low</td>
</tr>
<tr>
<td>Model checking</td>
<td>fault prevention</td>
<td>N/A</td>
<td>automatic</td>
<td>high</td>
</tr>
<tr>
<td>Theorem proving</td>
<td>fault prevention</td>
<td>N/A</td>
<td>automatic</td>
<td>high</td>
</tr>
<tr>
<td>Runtime techniques (post-deployment phase)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redundancy-based fault tolerance</td>
<td>fault tolerance</td>
<td>isolation, recovery</td>
<td>automatic</td>
<td>varied</td>
</tr>
<tr>
<td>Checkpointing and rolling back</td>
<td>fault tolerance</td>
<td>recovery</td>
<td>automatic</td>
<td>varied</td>
</tr>
<tr>
<td>Runtime monitoring</td>
<td>fault tolerance</td>
<td>detection</td>
<td>automatic</td>
<td>minimal</td>
</tr>
<tr>
<td>Trace analysis</td>
<td>fault tolerance</td>
<td>detection</td>
<td>automatic</td>
<td>varied</td>
</tr>
<tr>
<td>Built-in tests</td>
<td>fault tolerance</td>
<td>detection</td>
<td>automatic</td>
<td>minimal</td>
</tr>
<tr>
<td>Software rejuvenation</td>
<td>fault tolerance</td>
<td>recovery</td>
<td>automatic</td>
<td>minimal</td>
</tr>
<tr>
<td>Computer immunology</td>
<td>fault tolerance</td>
<td>detection, isolation, recovery</td>
<td>automatic</td>
<td>usually minimal</td>
</tr>
<tr>
<td>Self-healing software</td>
<td>fault tolerance</td>
<td>detection, isolation, recovery</td>
<td>automatic</td>
<td>varied</td>
</tr>
</tbody>
</table>

with respect to the handling of faulty software-hardware interactions.

- only few techniques can be demonstrated to be correct and reliable, addressing the issue that the SWHM-software is a safety-critical piece of software itself.

We hope this work will shed light on some strengths and weaknesses of SWHM approaches proposed in the literature of related areas of study. The presented classifications should also allow researchers and users to gain better understanding of the current state of this new and exciting field.

V. ACKNOWLEDGEMENTS

This work is supported by a NASA NRA grant NNX08AY50A "ISWHM: Tools and Techniques for Software and System Health Management".

REFERENCES