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

In most modern complex vehicles (e.g., spacecraft, aircraft, and automobiles) hardware such as central electrical, mechanical, and hydraulic components are monitored by Integrated Vehicle Health Management (IVHM) 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 techniques, appropriate mitigation strategies can be selected (replacement or repair, switch to redundant systems, etc.).

Also crucial is the reliability of the software employed by these vehicles. In fact, these complex systems rely more and more on software to a point where software failures have caused severe losses. Software failures during a manned mission can cause loss of life. So, there are severe requirements to make the software as 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 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:

- 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 in the software system (are dormant) and can show up during operation. Software errors also do not "go away" on their own.
- Failures in software 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 purpose[1], by accident during development (e.g., replacement of an 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 software errors). 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., decreasing 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 take swift actions in order to detect and recover from failures.

[1]In Ariane V some 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), which led to an uncaught overflow error, causing the rocket to behave erratically and required its destruction.

- 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]. We analyze these techniques along the following dimensions:

- **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 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 deal with faults differently: 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 manifests itself 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 some 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.** Some techniques can be executed fully automatically, while others require a high level of human intervention (manual), and some can be partially automated (semi-automatic). Although, automatic processing may be 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 model) and in computational resources during SWHM execution. There is a clear trade-off 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 manage certain faults. Again, other approaches provide statistical estimates and failure probabilities.

We use these dimensions to classify SWHM techniques in order to provide a map for dealing with software faults and failures. Table I summarizes our classification. In this table, considered SWHM techniques are presented according to the phases in the software life-cycle at which they are typically utilized. Columns 2-5 correspond to the rest of the dimensions discussed above.

All runtime techniques are automatic and usually require few resources (esp. relative to V&V techniques). Very few approaches can guarantee the completeness of fault handling. Note that many techniques cannot be classified according to FDIR, because they are meant to be applied before the faults become active. Also, we do not classify the resource requirement for design and programming paradigms, because their usages are different from the rest of the techniques.

These classifications provide valuable information on current approaches, allowing us to identify the gaps in existing SWHM technology and to develop new techniques to address various challenges unique to SWHM. Future techniques can be evaluated based on the proposed metrics and added to this map to make it more complete.

IV. Conclusions

We discussed challenges associated with different aspects of SWHM and analyzed 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.
- most approaches are tailored toward discrete software, like finite state machines, or mode logic. Monitoring of continuous calculations as they, e.g., occur in guidance, navigation, and control (GN&C), are seldomly addressed.

Technique	Fault handling	FDIR	Automation	Resources	Completeness
Design and programming methodologies (design/development phase)					
Model-based design	fault prevention	N/A	manual	N/A	No
Goal-based operations	fault prevention	N/A	manual	N/A	No
Aspect-oriented programming	fault prevention	N/A	semi-automatic	N/A	No
Recovery-based computing	fault prevention, fault tolerance	recovery	manual	N/A	No
Software configuration management	fault prevention	N/A	semi-automatic	N/A	No
Verification and Validation (V&V) (testing phase)					
Testing	fault removal	N/A	manual, semi-automatic	adjustable	No
Simulation	fault removal	N/A	automatic	moderate-high	No
Debugging	fault removal	N/A	semi-automatic	varied	No
Numerical analysis	fault removal	N/A	manual	low	No
Model checking	fault removal	N/A	automatic	high	In some cases
Theorem proving	fault removal	N/A	automatic	high	In some cases
Runtime techniques (post-deployment phase)					
Redundancy-based fault tolerance	fault tolerance	isolation,recovery	automatic	varied	No
Checkpointing and rolling back	fault tolerance	recovery	automatic	varied	No
Runtime monitoring	fault tolerance	detection	automatic	minimal	No
Trace analysis	fault tolerance	detection	automatic	varied	No
Built-in tests	fault tolerance	detection	automatic	minimal	No
Software rejuvenation	fault tolerance	recovery	automatic	minimal	No
Computer immunology	fault tolerance	detection,isolation	automatic	usually minimal	No
Self-healing software	fault tolerance	detection,isolation,recovery	automatic	varied	No

TABLE I
CLASSIFICATIONS OF SOFTWARE HEALTH MANAGEMENT TECHNIQUES.

- most of these techniques are for software and for software only. Hence, their performance is weak with respect to the handling of faulty software-hardware interactions.
- there is not much integration between approaches from different phases of SW life-cycle. Getting techniques from different phases to collaborate (e.g., one technique may "set the stage" for the other) could result in increase coverage and efficiency.
- 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

[1] *Demonstration of autonomous rendezvous technology mishap investigation board review*, Tech. Report RP-06-118, NASA Engineering and Safety Center Technical Report, 2006.

[2] A. Avižienis, *The n-version approach to fault-tolerant software*, IEEE Trans. on Software Eng. **SE-11** (1985), no. 12, 1491–1501.

[3] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Trans. on Dep. and Sec. Comp. **1** (2004), no. 1, 11–33.

[4] Antonia Bertolino, *Software testing research: Achievements, challenges, dreams*, FOSE '07, IEEE Computer Society, 2007, pp. 85–103.

[5] G.V. Bochmann, R. Dssouli, and J.R. Zhao, *Trace analysis for conformance and arbitration testing*, Software Engineering, IEEE Transactions on **15** (1989), no. 11, 1347–1356.

[6] G. M. Bull, *Dynamic debugging in basic*, The Computer Journal **15** (1972), no. 1, 21–24.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled, *Model checking*, The MIT Press, January 2000.

[8] Daniel D. Dvorak, Michel D. Ingham, and J. Richard Morris, *Goal-based operations: an overview*, AIAA Infotech@Aerospace. Conf., 2007.

[9] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, *A survey of rollback-recovery protocols in message-passing systems*, ACM Comput. Surv. **34** (2002), no. 3, 375–408.

[10] Jacky Estublier, *Software configuration management: a roadmap*, ICSE '00, 2000, pp. 279–289.

[11] Donald Firesmith, *Testing object-oriented software*, TOOLS (11), 1993, pp. 407–426.

[12] Stephanie Forrest and Catherine Beauchemin, *Computer immunology*, Immunological Reviews **216** (2007), no. 1, 176–197.

[13] Bill Hetzel, *The complete guide to software testing (2nd ed.)*, QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[14] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, *Software rejuvenation: analysis, module and applications*, Proc. of 25th International Symposium on Fault-Tolerant Computing (1995), 381–390.

[15] Angelos D. Keromytis, *Characterizing self-healing software systems*, MMM-ACNS'07, 2007, pp. 22–33.

[16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean M. Loingtier, and John Irwin, *Aspect-oriented programming*, ECOOP, 1997, pp. 220–242.

[17] Mark Kordon, Steve Wall, Henry Stone, William Blume, Joseph Skipper, Mitch Ingham, Joe Neelon, James Chase, Ron Baalke, David Hanks, Jose Salcedo, Benjamin Solish, Mona Postma, and Richard Machuzak, *Model-based engineering design pilots at jpl*, IEEE Aerospc conf., 2007.

[18] J. Kramer and J. Magee, *Dynamic configuration for distributed systems*, IEEE Trans. on Software Eng. **SE-11** (1985), no. 4, 424–436.

[19] J. C. Laprie, *Dependable computing and fault tolerance: concepts and terminology*, (1985), 2–11.

[20] David A. Patterson, *Recovery oriented computing: A new research agenda for a new century*, HPCA, 2002, p. 247.

[21] B. Randell, *System structure for software fault tolerance*, Proc. of intl. conf. on Reliable software, 1975, pp. 437–449.

[22] B.A. Schroeder, *On-line monitoring: a tutorial*, Computer **28** (1995), no. 6, 72–78.

[23] Johann M. Schumann, *Automated theorem proving in software engineering*, Springer-Verlag New York, Inc., New York, NY, USA, 2001.