Reflex and Healing Architecture for Software Health Management

Abhishek Dubey, Nagbhushan Mahadevan, Robert Kereskenyi
Institute for Software Integrated Systems
Vanderbilt University, Nashville, TN

Abstract—**This paper discusses the applicability of reflex and healing architecture for implementing software health management in complex 'system of systems', such as those used in interplanetary space missions.**

I. Introduction

Complex systems such as those used in space missions are built as composition of systems. Each sub-system is a collection of components which are implemented either in hardware or software or as a combination of both. A component can contain other components, giving the 'global level' system a recursive structure that has potentially several levels. Mission critical and safety critical nature of these systems require implementations that are resilient to failures. Ariane 5, Mars Climate Orbiter and Mars Polar Lander are a few examples of the catastrophic consequences of software and related system failures.

Delay in communications due to distance, especially in interplanetary missions, requires these systems to be autonomic i.e. they have to provide resilience to faults by adaptively mitigating faults and failures through self-management. In order to achieve this, a system must be able to detect occurrences of discrepancies that signify failures, diagnose and isolate the probable fault sources, take actions to contain the faults i.e. stop them from propagating outwards to the parent component, and mitigate their effects on the functionality of the sub-system where the fault occurred.

Rasmussen [1] suggests that one of the traditional problems with implementing fault-tolerance is that it is seen as an add-on that is orthogonal to the system development process. He argues that fault-protection behavior should be integrated with the control problem during the design process. This is the basis of goal-based control paradigm [2] that supports a deductive controller that is responsible for observing the plant's state (mode estimation) and issuing commands to move the plant through a sequence of states that achieves the specified goal. This approach inherently provides for fault recovery by using the control program to set an appropriate configuration goal that negates the problems caused by faults in the physical system.

However, these control algorithms are typically implemented in software and are therefore reliant on the fault-free behavior of related software components. This imposes stringent requirement on software dependability. Of current interest are tools and techniques that provide an integrated health management infrastructure for software components. We believe that this infrastructure will be complementary to the main control executive.

A 'software health management' infrastructure needs to monitor software components at various levels for discrepancies, which are deviations from the expected behavior. Upon detection of faults, an online diagnosis engine is required to isolate and identify the fault source. This should be followed with appropriate mitigation actions. In this paper, we identify an architecture that could potentially be extended and applied to software health management.

II. Reflex and Healing Architecture

Reflex and Healing (RH) [3], [4], [5] is a biologically inspired two stage mechanism for recovering from faults in large distributed real-time systems. In the first stage, the primary building blocks of fault management are components called reflex engines (also referred to as managers) that are arranged in a hierarchical management structure. They offer pre-specified reactive responses called reflexes to faults as they are discovered. In the context of this paper, reflex is considered synonymous to fault-mitigation. Next stage of RH architectures is associated with system healing. It refers to a planned reconfiguration of the system at the global level. This is required if no suitable reflex exists for a fault-event or to optimize system performance after several reflexes. This architecture has been successfully demonstrated for the BTeV real-time embedded systems project (http://www-btev.fnal.gov/public/hep/detector/rtes/) [6].

RH architecture is typically deployed in a three level hierarchy of local level, regional level and global level, where 'local' means a specific component, 'regional' means a groups of components that are in close proximity, and 'global' refers to the entire system (see Figure 1). Interactions between different reflex engines are restricted to four types:

1) $S \leftrightarrow M$: Interaction between an operator/system interface and a manager
2) $M \overset{up}{\leftrightarrow} M$: Interaction between a subordinate manager and a governing manager
3) $M \overset{down}{\leftrightarrow} M$: Interaction between a governing manager and subordinate managers
4) $M \leftrightarrow C$: Interaction between a manager and a managed unit component

A Global Manager G is defined as a manager, which is capable of both $S \leftrightarrow M$ and $M \overset{down}{\leftrightarrow} M$ behavior. A Regional

Manager R is defined as any manager that is capable of both $M \overset{up}{\leftrightarrow} M$ and $M \overset{down}{\leftrightarrow} M$ behaviors. There can be multiple levels of regional hierarchy. Finally, local managers L are managers which are only capable of $M \overset{up}{\leftrightarrow} M$ and $M \leftrightarrow C$ behaviors.

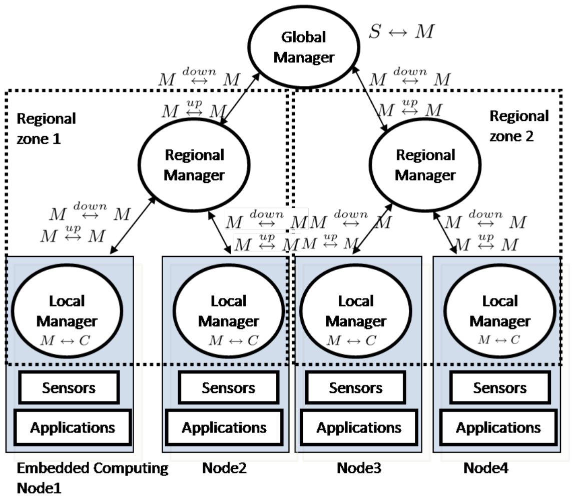

Fig. 1. The layout of management entities involving global (G), regional (R) and local (L) managers.

A. Reflex Engine

A reflex engine is composed of three distinct processes: a sensor process, an observer process and a mitigator process (see Figure 2). Monitoring information is generated by instrumenting embedded computing node with software sensors (scheduled using a sensor scheduler) that generate traces of value-events - a structure containing a value and a corresponding global time stamp. An example of these sensors is the 'heartbeat' sensor that periodically generates 'I am alive' messages. Other example is a CPU utilization sensor. It is important for the sensors to be least intrusive in terms of performance. They are critical component of this architecture and should be therefore certified rigorously.

Reflex engine integrates several fault management devices expressed as timed state machine models. While some of these state machines are used as observers that generate 'fault-events' (discussed in next section), others are used as mitigators to perform bounded time reactive actions upon occurrence of certain fault-events. Fault-events are propagated up in the hierarchy from local to global until a suitable mitigation strategy is found. A database (not shown in figure) stores all event exchanges for statistical investigations that require historical information.

An incoming event is forwarded to a scheduler, which maintains a lookup table for the event-type and the state machine that can use the event. The scheduler uses the lookup table to forward the event to the appropriate state machine. A challenge in this architecture is to verify that none of the state machines will ever be deadlocked. A potential verification procedure for this has been described in [5].

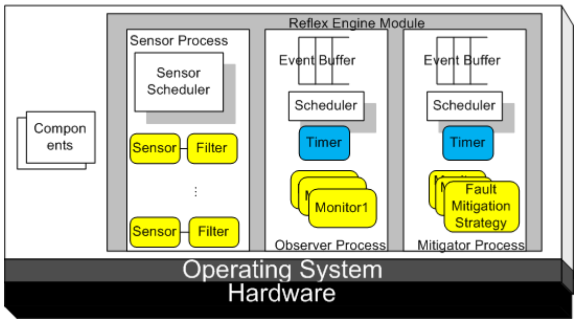

Fig. 2. A computing node in the real time reflex and healing framework.

B. Observers: Detecting Discrepancies

In the RH architecture, faults are detected through the observation of discrepancies that are deviations from the expected behavior. Expected behavior is specified as a Timed Computation Tree Logic (TCTL) formula. This formula is used to manually construct a timed automaton observer, which uses the monitoring information generated by software sensors. Violation of specified condition leads the observer to a faulty-state, where the entry action generates a fault-event. Presence of such an event indicates a discrepancy.

C. Distributed Diagnosis

A discrepancy could be potentially caused by many fault sources. It is useful to employ a diagnosis engine that can progressively use the discrepancy information and identify the fault source(s). We are investigating the use of distributed Timed Fault Propagation Graphs (TFPG) reasoner [7] for diagnosis. A TFPG-model is a directed acyclic graph (DAG) in which nodes depict faults (root-nodes) and discrepancies (non-root nodes). Edges represent propagation of failure between the nodes (faults to discrepancies, discrepancy to other discrepancies). Edges specify additional properties with regards to failure propagation - minimum/maximum time limits & permissible mode(s) for failure propagation.

When a discrepancy event (alarm) is generated, the reasoner (diagnosis engine) tries to identify the possible causes (or sources) for the alarm using the current configuration of the graph. This would most likely generate a set of valid hypotheses (faults) as the possible sources. Upon generation of further alarms, the valid hypothesis set is refined to include only those that can explain all the alarms. Once, the fault is suitably identified after the diagnosis, the mitigation strategy can be triggered by sending out the appropriate fault-event. Further, based on the model of the system, the TFPG diagnosis engine can also forecast / predict an impending failure / discrepancy, which could be extremely useful information, especially with regards to preventing catastrophic failures. The TFPG diagnosis engine employs a robust reasoning algorithm that can account for false alarms, missing alarms, intermittent alarms. Also, a single TFPG-model can be configured to enable multiple modes of operations for various components.

However, there are two challenges with adapting TFPG for use in online diagnosis in a RH architecture: (i) System components can operate in different modes, which leads to different software dependency chains during runtime. This might require large-scale rewiring of models based on the concepts of modes supported by TFPG. Or, it might require choosing a different TFPG model itself. (ii) Search space of a global TFPG model could be very large. Hence, we will need a component-based distributed reasoning approach where local reasoners generate local hypothesis in individual components, which could subsequently be improved by a global reasoner. A distributed and hierarchical TFPG reasoning scheme is currently being developed to address this problem. The TFPG model is split into a global model and multiple regional models where each region model accounts for a specific sub-system. This distribution of the model, allows individual regional reasoners to reason about the events in their sub-system. The failure interactions between the regions is captured in the Global model, which is responsible for ensuring that the regional reasoners are kept in a consistent state relative to one another (in terms of the effects of their failure propagation interactions).

D. Mitigation

Once the discrepancy source is identified (or faults are diagnosed), an appropriate mitigation strategy could be employed. This is done in the first stage of RH architecture using reflex strategies. These strategies are specified as timed state machines. They react to fault-events that are generated by other state machines in the architecture. The strategy can be reactive or proactive. In the reactive case, it is triggered by a discrepancy condition that indicates a system state after the failure has occurred and defines recovery actions. In proactive case, the condition describes a system state that can lead to a failure with a high probability and defines preventive actions.

E. Healing

Next stage of RH architectures is associated with system healing. It refers to a planned reconfiguration of the system at the global level, which is required if no suitable reflex exists for a fault-event or when system performance needs to be optimized after several reflexes actions. This involves a planning step that searches over a set of candidate models obtained by using a set of prespecified reconfiguration operations. This step is usually multi-objective in nature and is dependent on several factors such as system goals, resilience to future faults and performance. Development of such techniques that result in an optimal and healthy configuration remains an active research challenge.

III. Reflex and Healing Architecture in ARINC-653 Health Management

ARINC-653 specification describes the standard Application Executive (APEX) that should be supported by safety-critical real-time operating system (RTOS). It has been proposed to be used as the standard operating system interface on space missions [8]. Support for health monitoring and management services is an integral part of this specification. It should be noted that this specification provides a place for implementing fault-management strategies without favoring any particular technique.

TABLE I
Mapping RH hierarchy to ARINC-653 specification

RH Specification	ARINC Specification
Local-level	Process-level
First Regional-level	Partition-level
Second Regional-level	Module-level
Global-level	System-level

The Health Monitoring (HM) service, as described in the ARINC-653 specification is responsible for monitoring and reporting hardware, application and OS software errors. The errors could be classified as process level, partition level or module level errors. A process level error impacts one or more processes in the partition, or the entire partition. Partition level errors impact a partition. A module level error impacts all the partitions within the module. Table I maps the RH hierarchy to the ARINC specification.

Specification of HM service supports recovery actions by using call-back functions, which are mapped to specific error conditions in configuration tables at the partition/ module/ system level. We can extend the functionality of call-back functions by executing them as actions during the state transition of a reflex fault-management strategy.

IV. Conclusion

Implementing software health management in large safety-critical systems requires the presence of a scalable architecture that supports detection, diagnosis and mitigation of software faults. We believe that the Reflex and Healing architecture has a good potential to support hierarchical software health management, if necessary in conjunction with ARINC-653 compliant RTOS.

References

[1] R. Rasmussen, “GN&C fault protection fundamentals,” in *American Astronautical Society 31st Annual AAS Guidance and Control Conference, Breckenridge*, Feb 2008.

[2] B. C. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan, “Model-based programming of fault-aware systems,” *AI Magazine*, vol. 24, no. 4, pp. 61–75, 2004.

[3] S. Neema, T. Bapty, S.Shetty, and S.Nordstrom, “Developing autonomic fault mitigation systems,” *Journal of Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Grids*, 2004.

[4] R. Sterritt and D. Bantz, “Personal autonomic computing reflex reactions and self-healing,” *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 36, no. 3, pp. 304–314, May 2006.

[5] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai, “Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems,” *Innovations in Systems and Software Engineering*, vol. 3, pp. 33–52, March 2007.

[6] S. A. et al., “RTES demo system 2004,” *SIGBED Rev.*, vol. 2, no. 3, pp. 1–6, 2005.

[7] S. Hayden, N. Oza, R. Mah, R. Mackey, S. Narasimhan, G. Karsai, S. Poll, S. Deb, and M. Shirley, “Diagnostic technology evaluation report for on-board crew launch vehicle,” NASA, Tech. Rep., 2006.

[8] N. Diniz and J. Rufino, “Arinc 653 in space,” in *Data Systems in Aerospace*, European Space Agency. European Space Agency, May 2005.