# Detection Of Errors Using Aspect-Oriented State Consistency Checks

James Thai, Barry Pekilis, Alexander Lau, Rudolph Seviora
Bell Canada Software Reliability Laboratory
University of Waterloo, Waterloo, Ontario, Canada

{jthai, bpekilis, alexlau, seviora}@swen.uwaterloo.ca

## 1. Introduction

Software systems have grown in size and complexity. It is not possible to exhaustively test them to a fault free condition before they are deployed. As a result, their execution may be marked by failures. Often, these failures are preceded by a deterioration of internal state of the program (errors) before an externally observable failure occurs. Operationally, it would be beneficial to provide an indication of the internal well-being or 'health' of the system. At minimum, the system operator could be provided with a warning, so that corrective action could be taken. An aspect-oriented approach for implementing health monitoring of software systems is presented in [1], along with a classification scheme for health indicators. This paper presents a health indicator which reflects the extent of corruption of the internal program state. The focus is on embedded real-time systems, whose top-level objects have finite state machine behavior.

## 2. Consistency Checking

A measurement of the extent of corruption in the program state would require an oracle to determine what the correct program state should be and an appropriate metric for the distance between the actual and the correct program state. However, such oracles are unavailable.

Since a health indicator is only a statistical measure, an absolute check of correctness is not necessary. Instead, the work presented relies on checking consistency between selected parts of the program state. The extent of inconsistencies is used as the base for the health indicator.

Several issues arise with state-consistency health indicators. The first is *what* should be checked. There are many consistency relationships, both direct and indirect, in a reasonably sized program. However, only a limited number of consistency checks can realistically be carried out. The second is *when* such consistency checks should be performed during program execution. The third is *how* to derive the value of the indicator from the set of failed checks.

The goal of the work presented is to devise a methodology for dealing with these three issues for the class of software systems considered. An additional requirement is that the health indicator be implementable with aspects. Two reasons underlie this requirement, software engineering considerations and the capability to retrofit the indicator code into an existing software product.

The work assumes that requirement specifications and design models are available and expressed in the form of class-association diagrams and dynamic (FSM) models of class behavior. Also, design models from both high-level and detail design phases must be available.

The methodology for *what* – which parts of the program state should be checked for consistency – relies on an opportunistic mix of several criteria. In particular:

a) 'importance' of class associations and of parts of internal state. The development phase in which a particular association or partial state appeared is taken as an indicator of importance – the earlier, the more important;
b) metrics derivable from design models. These include both static metrics (e.g., association multiplicity, cardinality of sources of association traversals, length of traversal chain) and dynamic metrics (e.g., frequency of traversal of association, depth of call tree). Some indication about the operational profile under which the program executes is needed for dynamic metrics. Weighted metrics may be employed, which take into account whether a software class or package is reused, and whether a technology boundary is adjacent or crossed (e.g., software-hardware boundary);
c) prior experience with the error modes of the product.

The methodology for *when* to execute checks is based on the (FSM) states in the dynamic model of behavior of top-level classes. The states are classified based on their occupancy into stable and transient states, and, for session-oriented systems, also active and inactive. Consistency checks are carried out upon entry in the stable and inactive states. For scalability reasons, a rolling scheme may be employed.

The methodology for *how* to compute the value of the health indicator from the outcomes of the consistency checks calls for fairly simple processing. Specifically, the value of the indicator is the ratio of checks that passed to the total number of checks executed over an observation window. More sophisticated algorithms could be devised and may be appropriate under some circumstances.

Copyright ISSRE and Chillarege Corp. 2001

## 3. Example

Consider the partial class model for the control program of a small telephone exchange (PBX) in Figure 1. The model is layered. The classes in the top layer were identified in the early development phases; lower level software classes only appeared later in the design cycle.

The methodology devised suggests a number of possible consistency checks. One static-metric based check applies to top layer objects. The association between the PhoneHandler class and the (exchange) ResourceManager class is many-to-one. Within the PhoneHandler state machine, there are a number of transitions in which the association is traversed (a ResourceManager method is called). This yields a consistency check between the parts of the PhoneHandler state and the ResourceManager state that cover allocated resources. The check is invoked when the PhoneHandler reaches an inactive state (phone is on-hook).

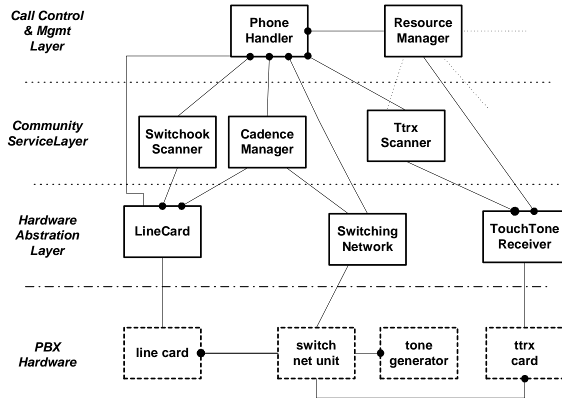


**Figure 1: Class Diagram of PBX Control Program**

Another consistency check is suggested by the presence of technology boundary between the hardware abstraction layer and the actual hardware (dotted outlines in Figure 1). Static metrics from the design model, weighted by the presence of this boundary, indicate a multi-way consistency check. The check determines whether the symbolic state of the PhoneHandler is consistent with the state of the switching network and the corresponding line card. This check is executed when the PhoneHandler object enters a symbolic state which is stable and in which a call progress tone is applied to the phone.

## 4. AOP Implementation and Experience

Until recently, it was difficult to insert consistency checking code into an existing program in a clean and safe manner. With aspect-oriented programming, such code can be implemented as an observer aspect and automatically woven into the program. The original source need not be modified, but only recompiled with an aspect preprocessor. By defining appropriate aspect advice (consistency checking code) and pointcuts (sets of program locations at which aspect advice is inserted), consistency checking is introduced into the program.

To evaluate the approach presented, the above and other health consistency checks were implemented. The target was the PBX control program mentioned earlier. The program was coded in Java, aspects in AspectJ [2].

Initial evaluation showed that the checks were generally implementable with aspects. However, the functional program had to contain 'get' methods to give the aspects access to the state of related objects. Alternatively, the aspect language had to allow direct access to the internal state of objects. Individual consistency checks were relatively small, typically some 100-150 lines of AspectJ. The supporting aspects were even smaller. Runtime overhead was ~5 μsec per advice invocation on Sun Ultra 1.

## 5. Concluding Observations

The paper presented one indicator of internal health of an executing software system, namely the extent of inconsistencies in its internal state. The paper outlined a methodology for determining what parts of the system state to check for consistency, when to execute the checks, and how to compute the value of the indicator from the set of failed and passed consistency checks. A requirement was that the checks be implementable with aspects.

Initial experience with AspectJ implementation of such checks for the control program of a small telephone exchange showed that the development cost and runtime overhead of each check were relatively low.

The consistency checks derived could be coded directly as a part of the system code. However, the aspect-oriented implementation provided a cleaner separation and almost eliminated the possibility of inserting new faults into the system code.


*Acknowledgements*

This work was supported by the Ontario Ministry of Energy, Science and Technology, Singapore-Ontario Joint Research Programme project UW2827601MEST.

The authors would like to thank our collaborators at the School of Computing, National University of Singapore and industrial partners in the project for their input.


## References


1. J.Thai et al., "Aspect-Oriented Implementation of Software Health Indicators", to appear, *Proc. Asia-Pacific Software Eng. Conf.(APSEC2001),* IEEE CS Press, Los Alamitos, Calif., 2001.
2. 2 G. Kiczales et al., "A Overview of AspectJ," *Proc. European Conf. Object-Oriented Programming (ECOOP 2001)*, Springer-Verlag, 2001.