# Learning About Software Errors Via Systematic Experimentation

**Terrance Goan  Oren Etzioni**

Department of Computer Science and Engineering, University of Washington
Seattle, WA 98195
{goan, etzioni}@cs.washington.edu

Classical planners assume that their internal model is both correct and complete. The dynamic nature of real-world domains (e.g., multi-user software environments) makes these assumptions untenable. Several new planners (e.g.,XII [2]) have been designed to work with incomplete information, and strides have been made in planning with potentially incorrect information. But, efficient operation in the presence of incorrect information is highly dependent on a planner's ability to detect errors. Failing to recognize errors can result in unexpected and potentially destructive effects, as well as further corruption of the world model.

This abstract describes ED (the Error Detective) which automatically generates error detection functions for a software robot (softbot). In addition to error detection, the functions generated by ED accurately *diagnose* the cause of the errors. The *automatic* generation of these functions is important due to the large number of conditions that can affect the success of command execution. In addition, the ability to diagnose the cause of an error can greatly reduce the number of preconditions which need to be checked/resatisfied prior to a successful execution of the operator. In tackling this problem we utilize three key insights:

(1) If an operator is completely specified, every error is due to some subset of the preconditions being unsatisfied.

(2) Software error messages generally signal only one error. For example, in UNIX, if you execute the `diff` command on two files x and y, where both files are **not** readable, the error message would be "`diff: x: Permission denied.`" It provides no information about the status of file y. More formally: error-msg(~p1 **and** ~p2) = (error-msg(~p1) **or** error-msg(~p2)), where ~p1 and ~p2 represent unsatisfied preconditions.

(3) Since software errors do not interact, errors can be fixed incrementally (the decomposable fault assumption). This means we need not assume just a single fault has occurred, rather we assume that if error-msg(~p1 **and** ~p2) = error-msg(~p2) and we re-execute the operator (after achieving p2), we will now get error-msg(~p1) which can be handled in turn.

ED "learns" the error diagnosis functions via a decision tree. The attributes which compose the training instances are: error message length (number of tokens) and a binary (present or not present) attribute for every token seen in the collected error messages. And the classes are sets of preconditions which may be at fault.

We compared the accuracy of the decision tree with two other methods: (1) our current hand-crafted error detection functions which rely on the presence of colons in error messages and makes no attempt to diagnose the cause of the error and (2) a simple string match method which looks for an exact match (except for command arguments).

The rate of correct diagnosis for the three methods were hand-crafted (57%), string match (71.6%), and decision tree (90.4%). In addition, both the string match and decision tree methods resulted in a significant decrease (50.5 and 63.6 respectively) in the number of preconditions which must be checked/resatisfied prior to re-executing an operator (i.e., the number of preconditions which can be rejected as the cause of the given error).

ED is complementary to "The Operator Refinement Method" presented in [1]. Where we assume correct operator models while developing error detectors, Carbonell and Gil assume accurate error detection while augmenting operator models.

Accurate error diagnosis allows yet another set of potentially fruitful experiments—finding the "optimal" set of operator preconditions. By "optimal" we mean the set of preconditions which best balance the cost of operator execution with the cost of error recovery. For example, it seems reasonable to execute `pwd` without verifying that all ancestor directories are readable. Simply execute the operator and handle the errors which will occasionally occur.

[1] Carbonell and Gil. Learning by Experimentation: The Operator Refinement Method. *Machine Learning: An Artificial Intelligence Approach* vol. III. 1990.
[2] Golden et al. XII: Planning for Universal Quantification and Incomplete Information. *AAAI* 1994.