# Diagnosing Misconfiguration with Dynamic Detection of Configuration Invariants

Dong Zhou
*DoCoMo USA Labs*
*zhou@docomolabs-usa.com*

## Abstract

*In software engineering, an invariant is a set of properties describing the value of a program variable. Dynamic invariant detection has been used to debug software errors. In this position paper, we propose to dynamically construct invariants for configuration settings, and to detect violations to configuration invariants for misconfiguration diagnosis. We also propose to use task-based suspect ranking to identify suspicious configuration changes made by less trustable programs to improve the diagnosis process. We describe our design of a misconfiguration diagnosis system based on dynamic configuration invariant detection and suspect ranking with task-related information.*

## 1. Introduction

Compared with closed computing platforms, users of open computing platforms are more likely to find themselves in situations like "The application just crashed! But yesterday it was fine!", or "why this game runs great on his device but doesn't work on mine?!" By open computing platforms, we mean platforms where users can extensively change the configurations of the hardware and software, including introducing new hardware and software into the platforms. Increasingly, personal computing systems are becoming open computing platforms, including the majority of PCs, as well as an increasing number of mobile devices, such as PDAs and cell phones.

The above described frustrating dependability related scenarios are more likely to happen on open computing platforms because such platforms are more suspicious to problematic configuration changes. For example, assigning wrong value to a configuration item (such as a port number) may cause problem for software that uses the configuration items. As another example, updating a library during the course of installing a new application may break old applications that depend on the library, if the new version of the library is not backward compatible.

Traditional approaches for diagnosing and solving such configuration problems involves first describing the symptoms of the problem, then from a problem database, matching described symptoms with a number of records, and finally, for each record, if the record indicates that the cause of the problem is a configuration item, then try to set the configuration item with suggested value. The process involves a large amount of human involvement, and taxes heavy cost for companies providing such customer services.

To reduce human involvement and other costs in the process, researchers have looked at approaches for automatically generating a list of suspected problematic configurations [1,2], approaches for ranking such suspects [1,2], and approaches for automatically recovering from misconfiguration [2,3]. Such existing efforts typically use the information that the value of a configuration item was changed to identify suspect configuration items. A limitation of such binary approach (i.e., the value was changed or not) is that, when the number of items that has changed is big, it is difficult to nail down the misconfigured item(s).

In Glean, Kicimen and Wang proposed to use correctness constraints on configuration classes. [4] A configuration class is defined as a group of Windows hierarchical registry keys with the same structure. A registry key violating any of the constraints of its configuration class is considered as possible misconfiguration. Four types of constraints are used: size constraint, which says that a subkey in a given configuration class has a fixed size; value constraint, which says that the value of a subkey in a given configuration class takes on one of a small set values; reference constraint, which says that a registry key should always reference instance of a particular configuration class; and equality constraint, which says that a group of registry keys should always have the same value.

Our work was motivated by Glean, as well as by dynamic invariant detection research in software engineering field [5,6,7,8]. An invariant is a set of properties describing the value of a program variable (or the values of a set of variables). Predicates describing properties can be flexibly defined. A sample predicate for an integer variable is whether certain bits of the variable are constant, or whether the value of the variable increases monotonically. Dynamic invariant detection has been proven useful in debugging software errors [5,7,8].

The first aspect of our work is thus to extend the concept and techniques of dynamic program variable invariant detection for software debugging to the field of misconfiguration diagnosis. That is, we dynamically detect invariants for configuration items, and we treat violations to invariants as potential misconfigurations.

Our work was also motivated from the observation that current diagnosis approaches, when faced with a large number of suspicious configuration changes, typically do not take into account information about who (which applications or processes) made such changes in ranking those suspected misconfigurations. The usefulness of such information in diagnosis is obvious. For example, assuming two applications, one had run successfully many times before and the other was just downloaded and installed, each made a suspicious configuration change, then it is empirically reasonable to say that the change made by the latter application more likely caused the misconfiguration. Thus in our system, we monitor the configuration accesses of each application, and use such access information for later diagnosis.

## 2. Overview

Figure 1 shows the components of the Fault Diagnosis Module in our design. The module is composed of a Read Access Table, a Write Access Journal, a Write Journal Manager, an Invariant Table, an Invariant Refinement Unit, a Task Violation Detection Unit, and a Suspect Ranking Unit.

The Read Access Table has one entry for each task. Each entry stores at least the following information; the identification of the task, a list of elements each having fields including the identification of the configuration item read, the time of the read access happened, and the value of the configuration item when the read access happened. Note that once a Task Success Notification message or a Fault Diagnosis Request message for a task has been processed, the task's entry in the Read Access Table can be removed.

The Write Access Journal is a log of all write accesses to all configuration items by all tasks. Each entry in the log contains at least following information: the identification of the configuration item, the identification of the task made the write access, the time of the write access, and the new value of the configuration item.

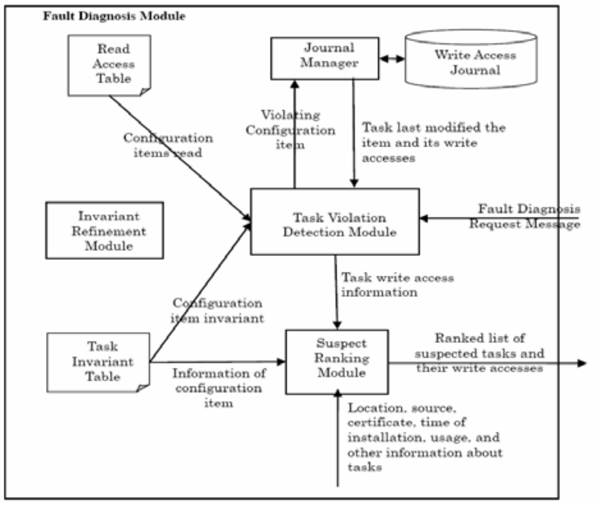
Fault Diagnosis Module
Read Access Table
Journal Manager
Write Access Journal
Violating Configuration item
Task last modified the item and its write accesses
Configuration items read
Invariant Refinement Module
Task Violation Detection Module
Fault Diagnosis Request Message
Task write access information
Configuration item invariant
Task Invariant Table
Information of configuration item
Suspect Ranking Module
Ranked list of suspected tasks and their write accesses
Location, source, certificate, time of installation, usage, and other information about tasks

**Figure 1. Component Diagram of the Fault Diagnosis Module**

The Write Journal Manager is responsible for maintaining the Write Access Journal. It adds new write access information, keeps the journal on persistent storage, and periodically removes old entries from the journal when they become useless.

An Invariant Table has one entry for each task. Each entry contains the identification of the task, and a pointer to a list for configuration accesses of the task. Each element in the list contains the identification of the configuration item, as well as a pointer to an invariant. Note that multiple elements from one or more lists can point to the same invariant. Figure 2 depicts the structure of the Invariant Table.

The Invariant Refinement Unit redefines an invariant for a configuration item accessed by a task with a new value. It first checks if the new value

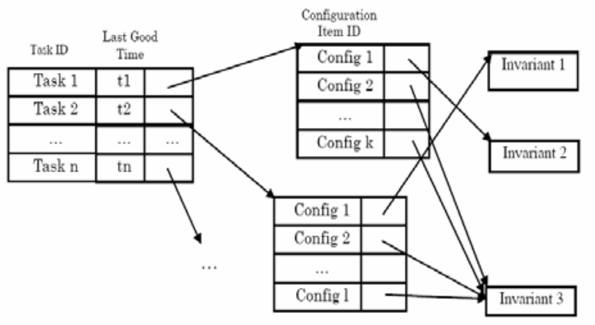

**Figure 2. Structure of Task Invariant Table**

violates the invariant. If it does, the invariant is redefined with the new value taken into consideration.

The Task Violation Detection Unit is responsible for determining the list of tasks that made write operations to configuration items and violated invariants for the failed task.

The Suspect Ranking Unit gets the list of suspect tasks from the Task Violation Detection Unit, ranks these tasks using an algorithm described in next section, and returns this ranked suspect task list with information about their write accesses to configuration items.

## 3. Configuration Invariants

There are three types of basic invariants: integer number invariants, real number invariants, and string invariants. Below is a sample implementation of the integer number invariant implemented in the C programming language:

```
typedef struct {
        int refineCount;
        char isConstant;
        char monotonic;
        ExpandableList *valueSet;
        int lowerBound, upperBound;
        int constBits;
} NumberInvariant;
```

The refineCount field is for recording the total number of refinements conducted on the invariant. The isConstant field is used to indicate whether the number value of the configuration item is constant. The monotonic field is used to indicate whether the number value of the configuration item increases or decreases monotonically. The valueSet field, if it is not set to null, contains all the previously assigned values of the configuration item. The lowerBound and upperBound fields indicate the lower bound and upper bound of the number value of the configuration item, respectively. The constBits field indicates the bits in the new value that should remain the same as in the same bits in the old value. A real number invariant is implemented similarly as a integer number invariant.

When a task accesses configuration items, the entry for the task in the Read Access Table is updated in case of a read access, otherwise, the Write Access Manager is notified to update the Write Access Journal.

When the success completion of a task is observed, the Invariant Refinement Unit gets the list of read accesses made by the task from the Read Access Table. For each configuration item that the task made a read access, the Invariant Refinement Unit retrieves the corresponding invariant (i.e., the invariant for the task and configuration item pair) from the Invariant Table, checks if the value of the read access violates the invariants. If a violation is detected, a copy of the invariant is first made if it is shared with others, the invariant is then refined with the value of the read access.

When the system is requested to diagnose the fault of a task,is received, the Task Violation Detection Unit gets the list of read accesses made by the task from the Read Access Table. For each configuration item that the task made a read access, the Task Violation Detection Unit checks if the value of the read access violates the invariant for the task and configuration item pair. If a violation is detected, the Task Violation Detection Unit hands the configuration item to the Write Journal Manager, which looks for the task that last made write access to the configuration item, and returns all the write accesses to all configuration items made by the task since the failed task has last reported success. The Task Violation Detection Unit collects all the violating tasks and their write accesses, and hands them to the Suspect Ranking Unit. The Suspect Ranking Unit ranks the violating tasks and sends out a Fault Diagnosis Result message.

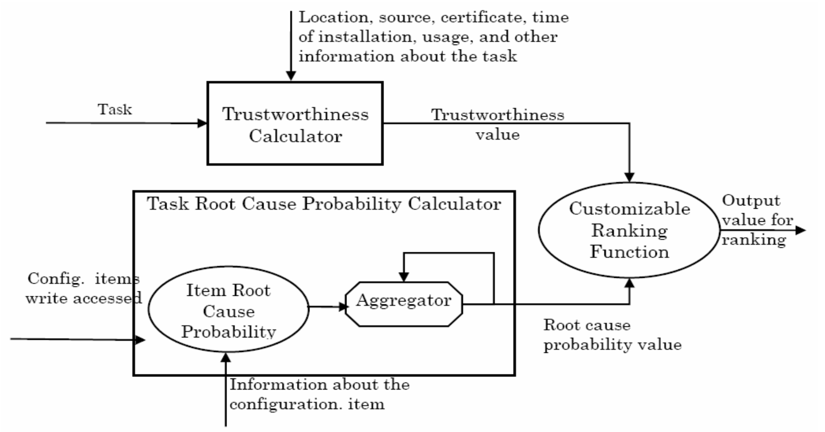
Location, source, certificate, time of installation, usage, and other information about the task
Task
Trustworthiness Calculator
Trustworthiness value
Task Root Cause Probability Calculator
Customizable Ranking Function
Output value for ranking
Config. items write accessed
Item Root Cause Probability
Aggregator
Root cause probability value
Information about the configuration. item

**Figure 3. Component and Data Flow Diagram of the Fault Diagnosis Module**

## 4. Ranking Invariant Violations

Figure 3 shows the components and data flow inside the Suspect Ranking Unit. The Suspect Ranking Unit ranks tasks based on their trustworthiness and the probabilities of their violation of configuration invariants being the root cause for the failed task. A customizable ranking function (F) is used for calculating the ranking of each task:

Ri = F(Ti, Pi)

where Ri is the ranking for task i, Ti is its quantified trustworthiness, and Pi is the probability that its violation of configuration invariants being the root cause of the failure. An instance of F is as follows:

F(Ti, Pi) = Ct / Ti + Cp * Pi

where Ct is the coefficient for trustworthiness, and Cp is the coefficient for root cause probability.

Trustworthiness value for each task is computed by the Trustworthiness Calculator. The calculator takes following information into consideration:

- Location of the binary executable or script of the task. A task started by binary executable or script residing in ROM has higher trustworthiness value.
- Source of the binary executable or script of the task: A task started by binary executable originated from, e.g.,, the wireless operator of the device has higher trustworthiness value. Those downloaded through Web browser has lower trustworthiness value.
- Certificate: A task started by binary executable or script with certificate has higher trustworthiness value.
- Time of installation: A task started by binary executable or script installed earlier has higher trustworthiness value than those installed later.
- Number of uses: A task that has been executed more frequently has higher trustworthiness value.

Root cause probability is computed by the Root Cause Probability Calculator. For task Ti, which made write operations to a list of configuration items (CI(i,1), CI(i,2), … CI(i,n)) and violated invariants of these configuration items, the calculator first calculates the root cause probability of each configuration item, then uses a customizable aggregation function to calculate the aggregated probability of the task based on the root cause probability of each configuration item.

The calculator computes the root cause probability of each configuration item with following information taken into consideration:

- The number of other tasks accessing this configuration item: a larger number indicates lower probability.
- The number of tasks succeeded after this configuration item was last write accessed: a larger number indicates lower probability.
- The number of times current value was assigned to the configuration item: a larger number indicates lower probability. (i.e., same value used by other tasks in the past)
- The number of write accesses to the configuration item made by the task being ranked over total number of write accesses: a higher ratio indicates higher probability.

At the end of its processing, the Suspect Ranking Unit outputs the order list of suspected tasks, each with its ranking and write accesses it made to all configuration items.

## *References*

[1] Jiahe Helen Wang, John C. Platt, Yu Chen, Ruyun Zhang and Yi-Min Wang, Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings OSDI 2004.*

[2] Andrew Whitaker, Richard S. Cox and Steven D. Gribble, Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of OSDI 2004.*

[3] Emre Kiciman and Yi-Min Wang, Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC),* 2004.

[4] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang, STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th USENIX conference on System administration, 2004.*

[5] Michael D. Ernst, Dynamically Discovering Likely Program Invariants. Ph.D. dissertation, University of Washington Department of Computer Science and Engineering, Aug. 2000.

[6] Michael D. Ernst, Adam Czeisler , William G. Griswold and David Notkin, Quickly detecting relevant program invariants. In *Proceedings of ICSE 2000.*

[7] Sudheendra Hangal and Monica Lam, Tracking Down Software Bugs Using Automatic Anomaly Detection. In Proceedings of ICSE 2002.

[8] Orna Raz, Philip Koopman and Mary Shaw, Semantic Anomaly Detection in Online Data Sources. In *Proceedings of ICSE 2002.*