Lightweight Automatic Error Detection by Monitoring Collar Variables

João Santos and Rui Abreu

Department of Informatics Engineering
Faculty of Engineering
University of Porto
Portugal
joao.filipe.santos@fe.up.pt, rui@computer.org

Abstract. Although proven to be an effective way for detecting errors, generic program invariants (also known as fault *screeners*) entail a considerable runtime overhead, rendering them not useful in practice. This paper studies the impact of using simple variable patterns to detect the so-called system's *collar variables* to reduce the number of variables to be monitored (instrumented). Two different patterns were investigated to determine which variables to monitor. The first pattern finds variables whose value increase or decrease at regular intervals and deems them not important to monitor. The other pattern verifies the range of a variable per (successful) execution. If the range is constant across executions, then the variable is not monitored. Experiments were conducted on three different real-world applications to evaluate the reduction achieved on the number of variables monitored and determine the quality of the error detection. Results show a reduction of 52.04% on average in the number of monitored variables, while still maintaining a good detection rate with only 3.21% of executions detecting non-existing errors (false positives) and 5.26% not detecting an existing error (false negatives).

Keywords: Error detection, program invariants, automatic oracles, dynamic execution.

1 Introduction

An application's *development phase* is usually restricted by the budget allowed for development and/or time-to-market. These restrictions provide a trade-off with the reliability of the system, which leads to an increase in defects that can lead to catastrophic results. In these cases proper *error detection* is vital in order to ensure the recognition and recovery from faults during the *deployment phase* as soon as possible [1]. One possible way of implementing error detection on a system is with the use of generic invariants, also known as *fault screeners*. They may present a higher rate of false positives (faults detected when none exist) and false negatives (the non detection of an error) when compared to hard coded error detection methods (such as asserts), due to the latter detecting anticipated faults. Despite this, generic invariants have the great benefict of being generated and intrumented *automatically* into the code. This along with the fact that (1) the invariants can be trained automatically during the *testing phase* and (2) hard coded solutions are cumbersome and time consuming to implement, might give an edge to generic invariants. Having generated automatically the invariants and trained them during the *testing phase*, they are ready for being used during the *deployment phase*, where the invariant detects deviations from the learned behaviour [2]. Generic invariants have been subject of study for many years, spawning various types like range screeners, bitmask screeners, and screeners that leverage Bloom filters [2,3]. They are mostly used for fault localization [4] and error detection [3].

Despite the benefits of generic invariants, their use on real-world, large software applications is currently impeded by the overhead that monitoring all the system's variables requires. However, monitoring every variable may not be required, as only a subset of variables, known as *collar variables*, truly affect the outcome of a system in a meaningful way [5]. Applications like TAR3 and TAR4.1 have some algorithms that already experiment on the detection of collar variables [6], but the use of these *collar variables* has not been applied on the reduction the number of generic invariants needed to monitor a system effectively.

To tackle this, two algorithms were devised to detect exectution patterns of variables both during executions and between them. These algorithms, called variable evolution pattern detectors in this paper, are executed during the training phase of the invariants and collect information from successful executions. During the operational phase, when the impact of the instrumentation overhead needs to be minimized, the data collected from the pattern detectors allows variables deemed unimportant to be ignored.

This paper makes the following contributions:

- Proposes two methods to detect variables that do not require monitoring (in other words, methods to detect the *collar variables* of the program under analysis).
- Investigates the reduction achieved on the number of used invariants on real world applications.
- Evaluates the quality of the *error detection* when comparing with the results obtained using the test suite of the applications.
- Reports the increase in execution time with the use of the invariants.

The paper is organized as follows. Section 2 gives a quick overwiew of how a fault screener works, along with a more detailed explanation of the used screener for the study, the dynamic range screener. In Sect. 3 explains the functioning of the two variable evolution pattern detectors. The experimental setup and results are shown in Sect. 4. Section 5 presents work related to this paper. Finally Sect. 6 gives some final thoughts and some insight on future work.

2 Fault Screeners

First used by Ernst et al. [7], fault screeners, also known as program invariants, are fault tolerance mechanisms that use historical data recovered from previous executions to determine the expected behaviour from a system's variables, issuing a warning when the expected behaviour is not met [2]. Hence, the use of fault sceeners is a possible way to achieve automatic error detecting by monitoring the system's variables. However, for the detection to be effective, a training phase is required. During this phase the spectrum of valid variable values is determined. This constitutes the expected behaviour for a variable that should raise a warning in case a value that does not fit the spectrum is detected [8]. Formally, screeners are not effective at detecting errors that involve the use of random values, or variables that store things like current timestamp.

There are various types of invariants, each with its own algorithms for training and error detecting. In this paper it is focused on the dynamic range invariants [2], due to its simplistic nature, reduced overhead, and known to work in practice [4]. The dynamic range invariant stores the bounds of valid variable values. During the training phase, when a new value is found, the range of values allowed by the screener is extended according to the following equations:

$$l := min(l, v) \tag{1}$$

$$u := max(u, v) \tag{2}$$

If the new value is lower than the lower bound l, the lower bound is updated. Likewise, if the value is greater than the upper bound u, that bound is updated. Table 1 shows an example of how the training works for the dynamic range screener. At first the invariant does not consider any value valid since no observation was made yet. After the first observation, in this case 5, both bounds need to be updated leading to a valid range of $[5, 5]$. The second observation is a 72. This value is greater then the upper bound of the range and not lower then the lower bound, so the upper bound is updated. With an updated range of $[5, 72]$, the new observed value 6 is compared to both bounds. It is between the upper bound and lower bound so no change is made. Lastly, the value 5004 is observed, again greater then the upper bound. This bound is updated leading to a final valid range of $[5, 5004]$.

Table 1. Dynamic Range Screener training

New Result Value	**Range Point**
5	$\emptyset$
72	$[5, 5]$
6	$[5, 72]$
5004	$[5, 72]$
	$[5, 5004]$

When on error detection phase, every observed value is checked against the range of values allowed by the invariant. If the value goes outside the range of permitted values, a violation to the expected behaviour is detected:

$$violation = \neg(l < v < u) \tag{3}$$

The dynamic range invariant can use a larger number of ranges in order to restrict the allowed spectrum [2]. While the concept is the same, additional ranges require more memory and more execution time. When using more then one range, the objective during the training phase is: when a new value is observed, the updated range is the one that increases the valid spectrum by the least amount of values. Table 2 shows an example of a dynamic range invariant with two ranges. The invariant begins with two empty ranges. Once it observes the value 5, one of the ranges becomes $[5, 5]$. On the second observed value, 72, since there is still one range that is empty, that range becomes, $[72, 72]$. Now that both ranges, when new values are observed, the invariant tries to make the ranges as short as possible to learn the least amount of unseen values. When 6 appears, there would be two range choices, $[5, 6]$ and $[72, 72]$ or $[5, 5]$ and $[6, 72]$. Since the first has the smaller ranges, this is the selected option. The last value 5004 provides an interesting twist. At first glance it would seem that this update would lead to $[5, 6]$ and $[72, 5004]$, however that is not the case. The ranges are actually updated to $[5, 72]$ and $[5004, 5004]$. This happens because the amount of values that is learnt is a lot smaller (from 6 to 72 compared to from 72 to 5004) and it still guarantees both the acceptance of the values from the values before the update and the new value observed. In this paper, the only version of the dynamic range invariant used is the single range one.

Table 2. Dynamic Range Screener training with two segments

New Result Value	**Range Point 1**	**Range Point 2**
5	$\emptyset$	$\emptyset$
72	$[5, 5]$	$\emptyset$
6	$[5, 5]$	$[72, 72]$
5004	$[5, 6]$	$[72, 72]$
	$[5, 72]$	$[5004, 5004]$

One of the challenges for using generic invariants is the accuracy of the *error detection*, as the more training the invariants suffer, the number of false positives, errors detected that do not exist, tends to decrease, while the number of false negatives, the non detection of existing errors, increases [9]. This happens because of the increase of accepted values by the invariant.

Figure 1 displays a possible setup for a dynamic range invariant. During the *training phase* the invariant learnt that the values between -2 and 2 were the valid set of possible values. However the real case is that the values should be valid between -3 and -1, as 1 and 2, as well as between 3 and 4. This leads to some false positives and false negatives. Values observed that withing the ranges $[-3, -2[$ or $]3, 4]$ issue a detected error warning, hence they are false positives. Likewise, observations between -1 and 1 do not issue any warnings when they should.

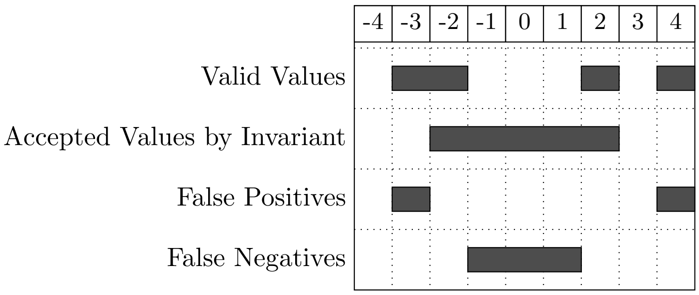

Fig. 1. False positive and false negative example

In the same scenario, if the invariant had been subject to more training, then more values would be added into the accepted range. On Fig. 2 the number 4 was such a value (even though 4 should not appear during executions if). This led to the values ranging from 3 to 4 to become valid, eliminating those false positives, but the ones from 2 to 3 also became valid, becoming new false negatives. In other words, there was an increase of false negatives and decrease of false positives. With more training the false positive rate tends to lead to 0 because the entire possibility of values become valid.

On the other side, the number of false negatives increases because since it accepts a lot more values then it should, it does not detect any values outside the huge accepted range.

Note that there are other types of invariants, each with their own behaviour regarding accuracy of error detection and performance [4]. Among them are bitmask invariants, which use a bitmask with the bits that were changed during the training when compared with the first observed value. Another one is the Bloom filter, an invariant that saves the entire history of values observed during the training phase. In this paper, the results were obtained by only using the dynamic range invariant. However the approach proposed is easily extensible to other invariant types.

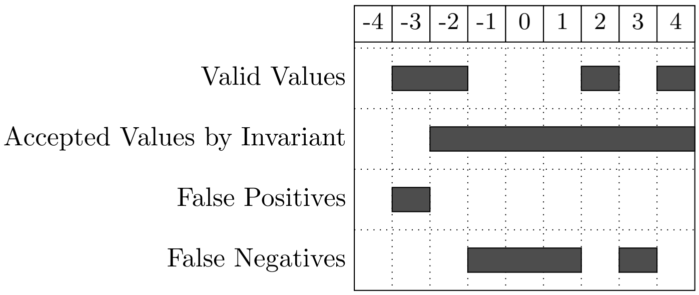

Fig. 2. False positive and false negative example with increased training

3 Variable Evolution Pattern Detectors

In this section, the two methods created to detect patterns on the variable values are presented. These patterns were designed to be as simple as possible, while still detecting constants and other variables, like counters. It is important to note that a variable is never classified as not important to monitor if it was only used on one execution of the system.

3.1 Delta Oriented Pattern Detector

The Delta Oriented Pattern Detector is the first of two algorithms created to detect *collar variables*. With this detector, the main objective is to discover variables that throughout its life cycle evolve in a constant fashion. These variables are then deemed not essencial since during every execution its value increases or decreases in the same manner, no matter what the input is, in other words variables with such detected pattern do not need to be monitored. This is accomplished by using a delta value (Δ), that is the difference between the last value observed and the current one:

$$\Delta := \text{current value} - \text{last value} \quad \text{if last value} \neq \emptyset \tag{4}$$

$$\Delta := 0 \quad \text{if last value} = \emptyset \tag{5}$$

Algorithm 1 demonstrates how this detector can determine which variables are important to monitor. Every variable in the system has a Δ associated to it. During the training phase, when the first value is observed, Δ is given the value 0 and the last value is updated to the observed one. On the next observation, Δ will be updated accordingly, using the current value and the last value, as seen in Line 8. After this, the pattern detection begins. With each observation, an updated Δ is generated (Δ_2) and is compared to the current Δ. If the new Δ is equal to the current one, the pattern detection continues as the evolution of the variable remains the same. In case the Δ is different, since the pattern is broken, a flag is stored indicating that this pattern does not exist for the variable being evaluated. There is, however, an exception to this. When the new Δ is 0, then it is not compared to the previous Δ (Line 12). This is done because variables can be accessed without their values being changed.

After each execution, the value of Δ is saved along with a flag indicating whether the pattern was broken or not. Subsequent executions use the Δ from the first execution and starts the pattern detection after the first two values, instead of after the third like the first run.

With this detector it is possible to detect constant values ($\Delta = 0$), as well as counters and loop variables that always increment/decrement with the same pace. A good example of this is the `Java` code presented on Fig. 3. Of all the variables from this small code sample, `j` is the one that has the least impact on the outcome. It only serves as an auxiliary variable for the loop.

Algorithm 1. Delta Oriented Pattern Detector

```
1: $pattern := true$
2: for all Execution $\bigwedge$ pattern do
3:     for all Observation $\bigwedge$ pattern do
4:         if first observation then
5:             $\Delta := 0$
6:             $LastValue := ObservedValue$
7:         else if second observation $\bigwedge nRuns = 0$ then
8:             $\Delta := ObservedValue - LastValue$
9:         else
10:            $\Delta_2 := ObservedValue - LastValue$
11:            $LastValue := ObservedValue$
12:            if $\Delta \neq \Delta_2 \bigwedge \Delta_2 \neq 0$ then
13:                $pattern := false$
14:            end if
15:        end if
16:    end for
17:    $nRuns++$
18: end for
```

The delta oriented pattern detector can be used to mark this variable as not essencial. It does not matter what the input of this function is, because `j` will always increment in the same manner. Δ will always be 1 (`j` always starts with the value 0 and increments by one on every access), so the pattern is never broken. Since this pattern is never broken, the variable will not be monitored during the error detection phase.

3.2 Range Oriented Pattern Detector

One of the main differences between this pattern and the previous one is that the range oriented pattern detector requires one full execution before it can determine a broken pattern. The basis of this detector is that if the range of values that a variable has between every run is the same, then it is not important to monitor. This is the reason why one full execution is required. The detector only has the range of the full execution at the end of it.

The functions of updating the bounds of the range are the same as the dynamic range invariant:

$$l := min(l, v) \tag{6}$$

$$u := max(u, v) \tag{7}$$

The main difference between the dynamic range invariant and the range oriented pattern detector is that the bounds of the detector are only updated on the first

```
public int funcExample(int i) {
   int accumulator = i;
   for(int j = 0; j < 3; j++) {
     if(accumulator == 1)
       break;
     accumulator *= accumulator;
   }
   int result = accumulator * 3;
   return result;
}
```

Fig. 3. Delta Detector code example

execution that a variable appears in. On the following executions, every time a new value is observed, it is determined if it is within the range of the first execution:

$$broken = \neg(l < v < u) \tag{8}$$

Algorithm 2 shows how the detector works. During the first execution (Lines 4 and 5) the range is constantly updated with every observation of a given variable. Once the first execution is over, the pattern detector is ready to discover a pattern. Hence, on the following executions, each observed value is compared to the pattern detector range, as seen in Line 7. If the new value is not within the range determined by the first execution, then the pattern was broken. If this never happens then it is determined that there is a pattern in the execution and the variable will not be monitored during the error detection phase.

With this detector it is possible to detect variables that although do not evolve in a linear way that can be detected by the delta oriented pattern detector, are restricted in some way during the execution. This is the case of loop variables that are affected within the cycle. This can be seen in the example shown on the example shown on Fig. 4. In this case, variable `j` is not a very important variable to be monitored. Taking into account the previous, detector, it is easy to understand that it would not be marked as not essencial (as Δ can be both 1 or 2). However the range oriented detector can find a pattern. On every execution, despite what input is received, the range of values `j` takes is always $[0, 5]$. During the first execution, this range would be given to the pattern detector and the following runs would follow the pattern, so the variable would not be monitored.

Algorithm 2. Range Oriented Pattern Detector

```
1: $pattern := true$
2: for all Execution do
3:     for all Observation do
4:         if $nRuns = 0$ then
5:             $updatePatternRange(ObservedValue)$
6:         else
7:             if $ObservedValue \notin PatternRange$ then
8:                 $pattern := false$
9:             end if
10:        end if
11:    end for
12:    $nRuns++$
13: end for
```

```
public int funcExample(int i) {
    int accumulator = i;
    for(int j = 0; j < 5; j++) {
        if(accumulator == 1 && j < 3)
        j=j+2;
        accumulator *= accumulator;
    }
    int result = accumulator * 3;
    return result;
}
```

Fig. 4. Range Detector code example

4 Empirical Results

In this section the experimental setup is presented, along with the workflow of the experiments themselves. After that the experimental results are discussed.

4.1 Experimental Setup

Application Set. During the experimentation, three real world applications were used:

- `NanoXML` [1] - a XML parser.
- `org.jacoco.report` [2] - a report generator for the `JaCoCo` library.
- `XML-Security` - a XML signature and encryption library from the Apache Santuario [3] project.

In Table 3 some details of the applications used are shown. These details include the number of lines of code and the number of test cases.

[1] NanoXML – `http://devkix.com/nanoxml.php`

[2] JaCoCo – `http://www.eclemma.org/jacoco/index.html`

[3] Apache Santuario – `http://santuario.apache.org/`

Table 3. Application details

Subject	LOC	Test Cases
`NanoXML`	5393	9
`org.jacoco.report`	5979	235
`XML-Security`	60946	462

`NanoXML` is a free, easy to use and non-GUI based and non-validating XML parser for Java. It has three different components:

- `NanoXML/Java`, the main standard parser.
- `NanoXML/SAX`, an `SAX` adapter for the standard parser.
- `NanoXML/Lite`, an extremely small version of the parser with limited funcionality.

`NanoXML` is available under the zlib/libpng license, which is Open Source compliant.

`JaCoCo` is an open source code coverage library for Java, being developed by EclEmma. The current goal of `JaCoCo` is to provide a code coverage library that is able to provide coverage reports. To do this there is a bundle called `org.jacoco.report`. This bundle is able to provide reports in three formats:

- `HTML`, for end users.
- `XML`, to be processed by external tools.
- `CSV`, suitable for graph creation.

`XML-Security` is one of the libraries available on the Apache Santuario project, a project that aims at providing security standards for XML. It is distributed under the Apache Licence Version 2.0 which is compatible with other open source licenses. The `XMLSecurity` data format provides encryption and decryption XML payloads at different levels, namely Document, Element and Element Content. XPath can be used for multi-node encryption/decryption. There exist two versions of `XML-Security`: a Java one and a C++ one. The Java version is used for the experiments.

Workflow of Experiments. In order to determine if the pattern detectors were effective at reducing the number of instrumented points and if the error detection maintained a good quality, the system's variables is subject to training first. Each application is instrumented in order to train the fault screeners. This training is achieved by executing a random number of test case (roughly 50% of the tests in the original suite) of the target program. We did not use the complete suite in order not to influence the results positively.

Once the training of the fault screeners is complete, the error detection phase begins. To evaluate the quality of the error detection, each application is executed five times. On each execution a different bug is inserted into the code and the number of false positives and false negatives are collected. An additional execution is performed without any inserted bug to determine the execution time in a regular scenario.

Each application's test suite was executed without any instrumentation as well to determine the increase of time the instrumentation brings.

Figure 5 shows the different phases of the experiments. First, during the training phase, the test are executed with the instrumented code. Everytime a variable is used, the update function of the screener is called in order to update the accepted values. In addition, the screener uses the pattern detectors to detect broken patterns. At the end of the execution, both the invariant and the data collected from the detector are saved. On the operational phase the test cases are executed with the instrumented code once again. However, this time instead of monitoring every variable, only the variables that did not have a detected pattern are observed. On each observation the value is then validated by the screener using information gathered during the training.

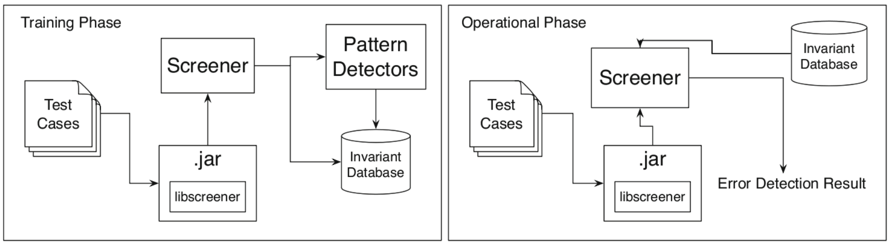

Fig. 5. Workflow of experiments

Injected bugs are of different types to guarantee a more varied input. Some examples of inserted bugs are:

- Change an operator when assigning values (i.e. change + to −).
- Change a random numeric value.
- Change comparation operator of a conditional clause (i.e. change a > to < on an `if` clause).
- Change the value of an argument of a function call.

With this setup the expected results are:

- Value of the reduction obtained in the number of used invariants.
- Comparison of execution times between executions with and without instrumentation.
- Accuracy of the error detection with the use of pattern detectors.

4.2 Results

Table 4 shows the number of variables that were trained and the number of variables that are considered *collar variables* by the pattern detectors. It is important to note that only numerical variables are subjected to training, in other words, only variables of the types `int`, `long`, `double` and `float`.

Table 4. Variable reduction

Subject	Variables trained	Collar Variables	Reduction
`NanoXML`	40	17	57.5%
`org.jacoco.report`	55	28	49.09%
`XML-Security`	325	164	49.54%

On average, a reduction of 52.04% is achieved with the use of the two pattern detectors. However the execution time of the program with instrumentation is also important to take into consideration. Table 5 presents the execution times of the test suites both with and without instrumentation. This instrumentation uses only *collar variables*.

Table 5. Execution time increase

Subject	Execution time with instrumentation (ms)	Execution time without instrumentation (ms)	Increase
`NanoXML`	270	827	206.3%
`org.jacoco.report`	3469	5162	48.8%
`XML-Security`	25005	63088	152.3%

The average increase in the execution time is 135.8%. Although this seems like a high value, it is greatly impacted by the increase noticed on `NanoXML` that is only a few miliseconds.

Having the data on the reduction of variables monitored and execution time increase, the quality of the error detection is what remains. To test the quality of the detection using these *collar variabes*, the number of false positives (N_{fp}) and false negatives (N_{fn}) was determined. A false positive is considered when the fault screener detects an error in the execution that does not exist. Likewise, a false negative is counted when a faulty execution has no objections raised from any fault screener.

The results shown on Table 6 were obtained by comparing the total number of false positives (N_{fp}) and false negatives (N_{fn}) with the number of tests on the test suite of the target program (N_t):

$$f_p := \frac{N_{fp}}{N_t} \tag{9}$$

$$f_n := \frac{N_{fn}}{N_t} \tag{10}$$

Table 6. False positive (f_p) and false negative rate (f_n)

Subject	**Bug 1**		**Bug 2**		**Bug 3**		**Bug 4**		**Bug 5**	
	f_p %	f_n %	f_p %	f_n %	f_p %	f_n %	f_p %	f_n %	f_p %	f_n %
`NanoXML`	0	0	0	0	0	0	0	66.67	0	0
`org.jacoco.report`	0	0	3.4	2.13	3.4	0	3.83	2.13	5.96	0
`XML-Security`	2.81	0.21	13.64	7.14	1.95	0.22	12.99	0.22	0.22	0.22

With an average of 3.21% rate of false positives and 5.26% rate of false negatives, the rate of these false results is considerably low, especially on the smaller applications. On the largest application, `XML-Security`, although having a higher rate of false results, the worst case scenario detected was a 13.64% f_p and 7.14% f_n.

In conclusion, with only the use of two pattern detectors, the decrease of used invariants is quite significant and the error detection quality remains very high, appart from some special cases. In terms of execution time, it may still not be enough to allow their use on real world markets, but perhaps the creation of even more detectors could be a solution.

4.3 Threats to Validity

The main threat to the validity of these results is the fact that only three test subjects were used during the experimentation. Despite these subjects being real world applications being diverse in both the size of the application (lines of code) and size of the test suite, the limited number of subjects implies that not all types of system's are tested. This means that a system with characteristics that are completly different might present different results.

Another threat is that the number of injected bugs is not enough to lead to accurate results, as these bugs might simply be "lucky bugs" that intercept a *collar variable*.

Naturally, there are also threats that are based on the implementation of the invariants, the instrumentation or the pattern detector algorithms themselves. The reduce these threats, additional testing was made prior to the experimentation to guarantee the quality of the experimental results in this regard.

5 Related Work

Since being introduced, generic invariants have been subject of study along the years with very different goals in mind. These goals range from study of program evolution [7,10], fault detection [2] and fault localization [3,11]. Invariants have also been used as an alternative way of error detection on a fault localization technique known as SFL [4,9].

Daikon [10] is a tool that reports likely invariants. It runs a program and then reports the properties observed during the executions. Besides storing pre-defined invariants like constants, range or linear relationships, it can be extended by the user with new invariant types. It is compatible with various programming languages, including `C`, `C++`, `Java` and `Pearl`.

Carrot [11] is a tool created with the purpose of using generic invariants for fault localization. It uses a smaller set of invariants than Daikon. The results obtained were negative which lead to the belief that invariants alone are insuficient as a means of debugging. However, in [9] the use of invariants for fault localization was successful when used as the input for the fault localization technique SFL.

DIDUCE [3] is yet another tool that uses dynamic bitmask invariants. Although the results appear to be good on four real world applications, the error that is detected is on a variable that is constant during the training phase and changed when it was on error detection mode (an error that is easily detected by a bitmask invariant, an invariant that detects differences on the allowed active bits of a variable value).

IODINE [12] is a framework for extracting dynamic invariants for hardware designs. It has been shown that accurate properties can be obtained from using dynamic invariants.

Zoltar [13] is a tool that applies a fault screener on every occurrence of a variable and tries to detect errors by finding perturbations on their behavior. In addition to detecting errors, Zoltar uses the errors detected to help debugging using SFL.

Another tool that works with fault screeners is PRECIS [14]. PRECIS introduces a different type of invariant based on pre- and post-conditions. The results obtained suggest the existance of some advantages over Daikon.

`iSWAT` [15] is a framework that uses invariants for error detection of a hardware level. It uses `LLVM` to instrument the source code to monitor the store values.

In [2] various invariants were subjected to performance evaluations. Among the tested invariants were dynamic range, bitmask, Bloom filters and TBL. Although the results show that bitmask outperforms Bloom filters and dynamic range, the errors used on the experimentation consisted of random bit switching, which is better suited for bitmask invariants and are not very common.

On the topic of *collar variables*, this term was used by Tim Menzies to describe the subset of variables that affect the output of an application in a meaninful way [5].

In [6] the algorithms of `TAR3` and `TAR4.1` are explained. These algorithms allow to obtain a ranking of "usefulness" of the different components of an application. `TAR3` uses the concepts of *lift*, the change that a decision makes on a set of examples, and *support*. `TAR4.1` uses Naive Bayes classifiers for the scoring heuristic in order to obtain an overall better performance in comparison to `TAR3.1`.

`KEYS` [16] is yet another algorithm that tries to discover the *collar variables*, called *keys* by the author. It is used to optimize requirement decisions and is faster then the `TAR3` algorithm. In [17], an improved `KEYS` algorithm is shown called `KEYS2`. It outperforms the original version by four orders of magnitude in terms of speed.

In [18], the concept of *collar variable* is once again used, this time by the name of *back doors*. They were using these *back doors* to solve CSP/SAT search problems and suggest by formal analisys the potencial improvement of some hard problems from an exponential to polynomial time.

6 Conclusions and Future Work

In this paper two simple detectors were used to evaluate what were the *collar variables* in each of the systems. Experimenting on real world applications led to a more accurate take on the impact of the use of invariants for error detection. By only using two detectors, the reduction of number of invariants used was above 50% while still maintaining good quality detection. Still the increase in execution time might still be too severe for use and the inability of the detectors to view patterns on non numeric values is still an obstacle.

In this regard, for future work in order to reduce the overhead, a further decrease in the number of invariants used is necessary. The study of additional detectors that would filter even more variables would be a possibility to achieve such a decrease. Another option is the use of an algorithm similar to the one used on TAR4.1 [6] to make the decision of what variables to monitor. There are also plans to combine this method with static analysis in orther to try to achieve better results.

Futher work will also be invested in tackling one of the main issues of the current approach, the ability to only evaluate numeric variables. Efforts will be made to use invariants and create detectors that would evaluate patterns for other variable types like `String` or `char`. These variables may prove invaluable to increasing the effectiveness of this method.

Acknowledgements. This work is financed by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PTDC/EIA-CCO/116796/2010.

References

1. Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N.: Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Computer Science Technical Report UCB//CSD-02-1175, 1–16 (2002)
2. Racunas, P., Constantinides, K., Manne, S., Mukherjee, S.S.: Perturbation-based Fault Screening. In: Proceedings of HPCA 2007, pp. 169–180 (2007)
3. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: Proceedings of ICSE 2002, pp. 291–301 (2002)

4. Abreu, R., González, A., Zoeteweij, P., van Gemund, A.J.: Automatic software fault localization using generic program invariants. In: Proceedings of SAC 2008, pp. 712–717 (2008)
5. Menzies, T., Owen, D., Richardson, J.: The strangest thing about software. Computer 40(1), 54–60 (2007)
6. Gay, G., Menzies, T., Davies, M., Gundy-Burlet, K.: Automatically finding the control variables for complex system behavior. Automated Software Engineering 17(4), 439–468 (2010)
7. Ernst, M.D., Cockrell, J., Griswoldt, W.G., Notkin, D.: Dynamically Discovering Likely Program to Support Program Evolution Invariants. In: Proceedings of ICSE 1999, pp. 213–224 (1999)
8. Dimitrov, M., Zhou, H.: Anomaly-Based Bug Prediction, Isolation, and Validation: An Automated Approach for Software Debugging. In: Proceedings of ASPLOS 2009, vol. 44, pp. 61–72. ACM (2009)
9. Abreu, R., González, A., Zoeteweij, P., van Gemund, A.J.: Using Fault Screeners for Software Error Detection. In: Maciaszek, L.A., González-Pérez, C., Jablonski, S. (eds.) ENASE 2008/2009. CCIS, vol. 69, pp. 60–74. Springer, Heidelberg (2010)
10. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1-3), 35–45 (2007)
11. Pytlik, B., Renieris, M., Krishnamurthi, S., Reiss, S.P.: Automated Fault Localization Using Potential Invariants. In: Proceedings of AADEBUG 2003, pp. 273–276 (2003)
12. Hangal, S., Chandra, N., Narayanan, S., Chakravorty, S.: IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. In: Proceedings of DAC 2005, pp. 775–778 (2005)
13. Janssen, T., Abreu, R., van Gemund, A.J.: Zoltar: A Toolset for Automatic Fault Localization. In: Proceedings of ASE 2009, pp. 662–664 (2009)
14. Sagdeo, P., Athavale, V., Kowshik, S., Vasudevan, S.: PRECIS: Inferring invariants using program path guided clustering. In: Proceedings of ASE 2011, pp. 532–535 (2011)
15. Sahoo, S.K., Li, M.L., Ramachandran, P., Adve, S.V., Adve, V.S., Zhou, Y.: Using likely program invariants to detect hardware errors. In: Proceedings of DSN 2008, pp. 70–79 (June 2008)
16. Jalali, O., Menzies, T., Feather, M.: Optimizing Requirements Decisions with KEYS. In: Proceedings of PROMISE 2008 (ICSE), pp. 1–8 (2008)
17. Gay, G., Menzies, T., Jalali, O., Feather, M., Kiper, J.: Real-time Optimization of Requirements Models. Jet Propulsion, 1–33 (2008)
18. Williams, R., Gomes, C.P., Selman, B.: Backdoors To Typical Case Complexity. In: Proceedings of IJCAI 2003 (2003)