AURORA: AUtomatic RObustness coveRage Analysis Tool

Angelo Gargantini,
Dip. di Ing. dell'Informazione e Metodi Matematici,
Università di Bergamo, Italy
angelo.gargantini@unibg.it

Marco Guarnieri,
Institute of Information Security,
ETH Zürich, Switzerland
marco.guarnieri@inf.ethz.ch

Eros Magri,
Dip. di Ing. dell'Informazione e Metodi Matematici,
Università di Bergamo, Italy
eros.magri@unibg.it

***Abstract*—Code coverage is usually used as a measurement of testing quality and as adequacy criterion. Unfortunately, code coverage is very sensitive to modifications of the code structure, and, therefore, we can achieve the same degree of coverage with different testing effort by writing the same program in syntactically different ways. For this reason, code coverage can provide the tester with misleading information.**

In order to understand how a testing criterion is affected by code structure modifications, we have introduced a way to measure the sensitivity of coverage to code changes by means of code-to-code transformations. However the manual execution of the robustness analysis is tedious, time consuming and error prone. In order to solve these issues we present *AURORA*, a tool that automates the robustness analysis process and leverages the capabilities offered from several existing tools. *AURORA* has an extendible architecture that concretely supports the tester in the execution of the robustness analysis. Due to this extendible architecture, each user can personalize the robustness analysis to his/her needs. *AURORA* allows the user to add new transformations by using TXL, which is a programming language specifically designed to support source transformation tasks. It performs the coverage evaluation by using existing code coverage tools and is based on the use of the JUnit framework.

***Keywords*-Code Coverage; Testing Criteria; Code Transformations; Coverage Robustness**

I. Introduction

The notion of code coverage and testing criteria dates back to the early sixties [1], [2]. Although, as Dijkstra claimed in 1972 [3], "program testing can be used to show the presence of bugs, but never their absence", coverage criteria are used to measure the *confidence* in the absence of errors in programs by testing them. The testing community has introduced, compared, and studied a large number of testing criteria, which have proved to be useful in finding faults in programs. However, there still exists some skepticism around the actual significance of coverage criteria. It is well known that testing criteria are very sensitive to the structure and to the syntax of the code, regardless its actual behavior. Rajan et al. [4] show that MCDC, required by FAA for software on commercial airplanes, and often considered a very tough criterion to achieve, can be easily cheated by simple folding/unfolding of conditions inside guards.

Despite their weaknesses, coverage criteria give an indicator of the quality of the testing that can be easily computed by running the code with the tests. Coverage is often used as acceptance threshold: if a test suite achieves a given coverage, it is considered adequate, and the tested software accepted as *good*. For this reason, reaching a given level of coverage becomes a critical factor during the testing activity.

We can state that the coverage data are easily obtained and are widely used as acceptance measure, but may have a questionable significance.

There are two main scenarios in which it is important knowing how the coverage offered by a test suite behaves with respect to the changes in the code structure.

A. The code has been transformed *in the past* before being tested and the coverage may depend on the transformations applied. In this way, testing criteria can easily be cheated, and hence an additional measure of the coverage fragility helps in identifying well-tested classes from poorly tested ones.

B. The code structure will change *in the future* without changing the input/output semantics of the program by applying some refactoring rules, by some automatic transformations, or by introducing particular patterns. This may influence the coverage after the application of these transformations. In this context, the tester would like to know if the level of coverage provided by the test suite will be preserved, i.e., if the coverage is robust with respect to future changes.

As highlighted in [5], a viable way for handling the behaviour of the coverage with respect to code transformations is to extend usual coverage criteria with an index that measures the losses of coverage that are caused by code transformations. We have proposed a new metric, called *extended coverage*, that extends the usual coverage measurement with an information on how much the coverage offered by the test suite is sensitive to transformations. The *extended coverage* consists of a pair of values (a, b) where a represents the usual coverage, whereas b is a fragility index that measures the sensitivity of the coverage to modifications in the code structure.

Extending existing coverage criteria instead of trying to define new ones that would be less sensitive to code transformations has the main advantage that the tester can reuse existing frameworks and tools for measuring code coverage: it is sufficient to transform the original code and apply the usual techniques to see how the coverage changes.

However measuring the *extended coverage* is a tedious task, especially when we have to consider a big program and an high number of transformations, because for each transformation we have to transform each file of the program and then we have to re-run the entire test suite over the transformed program in order to compute the loss of coverage. For this reason we

This work was partially done when Marco Guarnieri was with Dip. di Ing. dell'Informazione e Metodi Matematici, Università di Bergamo, Italy.

have developed the *AURORA* tool, which stands for *AUtomatic RObustness coveRage Analysis* tool.

In this paper we present the *AURORA*[1]tool that provides testers with the capability of computing the *extended coverage* achieved by a certain test suite ts over a program p in an automated way. The tool accepts code transformations defined by means of the TXL language [6], and uses standard coverage measurement libraries to compute the coverage achieved by ts on p, and using the transformations it automatically computes the fragility indexes.

Structure of the paper: Section II presents some examples that motivates the need of a way to measure the influence of code transformations on the coverage. In Section III, we present some theoretical background about the *extended coverage* metric, and we provide a simple introduction to the TXL transformation language. Section IV presents some code transformations and their formalization in TXL, whereas in Section V we describe the architecture of *AURORA*. In Section VI, a comparison is made with some related works, whereas in Section VII we present some considerations about the use of the extended coverage metric during the testing process. Finally, Section VIII draws a few concluding remarks and presents future work.

II. Motivation

Several works have already highlighted the fact that code coverage is very sensitive to the structure of the code, and thus it can be misleading as test adequacy criterion [4], [5], [7].

We motivate the need of quantifying how much the achieved coverage is sensitive to changes in the code by using four examples taken from [5]. These examples consider four different coverage criteria, namely MCDC, branch coverage, condition coverage and statement coverage. In each of these examples let p_1 be the code fragment on the left, and p_2 be the one on the right.

Example 1: Let `in_1` and `in_2` be two boolean variables.

p_1
```
boolean expr_1 = in_1 || in_2;
if (expr_1){
        ...
}
```

p_2
```
...
if (in_1 || in_2){
        ...
}
```

p_1 and p_2 are obviously semantically equivalent because they behave in the same way for each possible value of `in_1` and `in_2`. However, a test suite ts containing only two tests (in_1=**true**, in_2=**false**) and (in_1=**false**, in_2=**false**) achieves a full MCDC coverage of the *if statement* (with a simple variable as a guard) in p_1, whereas it does not achieve the full MCDC coverage of the *if statement* with the *inlined* condition in p_2. □

Example 2: Let x be an integer variable.

p_1
```
...
if (x>=2) x=x+1;
else x=x+2;
```

p_2
```
...
if (x==2) x=x+1;
else if(x>2) x=x+1;
else x=x+2;
```

[1]The *AURORA* Tool can be found at http://code.google.com/a/eclipselabs.org/p/aurora/. The same website contains also a tutorial and an installation guide.

Also in this case p_1 and p_2 are semantically equivalent. However a test suite ts containing only two tests (x=**0**) and (x=**5**) achieves 100% of branch coverage over p_1, but only 75% of branch coverage on p_2. □

Example 3: Let *a* and *b* be two boolean variables.

p_1
```
if(a && b){
   .... // body
}
```

p_2
```
if(a) {
  if (b){
      .... // body
  }
}
```

A test suite ts containing two test cases (a = **true**, b = **true**) and (a = **false**, b = **true**) achieves the full decision coverage of p_1, but it covers only the first decision of p_2, despite the fact that p_1 and p_2 have the same input/output behaviour. □

Example 4: Let *a* be a boolean variable and *i* be an integer variable.

p_1
```
if(a)
        i = i+1;
else
        i = i+2;
cout<<i;
```

p_2
```
if(a) {
        i = i+1;
        cout <<i;
} else {
        i = i+2;
        cout<<i;
}
```

Despite the fact that p_1 and p_2 are semantically equivalent, the test suite containing only one test (a = **true**) achieves 60% of statement coverage on p_1 but only 50% of statement coverage on p_2. □

The examples above show that also simple programs with the same input/output behaviour may achieve very different degrees of coverage with respect to the same test suite. This means that achieving the same level of coverage may require a different effort depending on the structure of the code itself (independently from the fact that the programs offer the same functionalities). However, it may be not advisable to force programmers to write programs in a particular way only to assure that a test suite achieves a particular degree of coverage. This could potentially diminish other qualities of the code, like readability or maintainability. A way of quantifying the influence of code structure on the coverage achieved by a certain test suite is thus needed to guide the testing process.

III. Background

In Section III-A we present some background needed to understand the concept of extended coverage and robustness whereas Section III-B provides some basic knowledge about the *TXL* language.

A. Theoretical Background

Due to space limitation here we give only a brief overview of the main concepts behind the *extended coverage* approach. A more detailed analysis of the problem can be found in [5].

We can formalize a *coverage criterion* as a function $C : TS \times P \rightarrow [0, 1]$ where TS is the set of all the test suites and P is the set of all the programs. We consider only program based testing criteria and thus a coverage criterion C takes as input a test suite ts and a program p and returns a number

that indicates to which degree a particular set of elements in the program p is covered by ts. For instance, the statement coverage criterion C returns the percentage of statements of the program p covered by the test suite ts.

A key concept of our approach is the definition of transformation, which represents a possible change to the source code of the program. A *code-to-code transformation* is a function $t : P \rightarrow P$ that takes as input a program and returns the transformed version of the program itself. We focus our attention on *Semantic Preserving Transformations* (SPTs) [8] which are transformations that change the structure of the code but not the input/output behaviour of the programs to which they are applied (i.e., a transformation t is a SPT iff for any $p \in P$ then p and $t(p)$ have the same input/output behaviour). In the following we refer only to SPTs and thus we call them simply transformations. Given a sequence of transformations T, we define the transformation t_{seqT} as the application of the transformations in T in sequence, i.e. $t_{seqT} = t_n \circ t_{n-1} \ldots \circ t_1$ where $t_1, \ldots, t_n \in T$. Given a transformation t, we define the transformation $\tilde{t}$ as the iterative application of t until the program is no longer modified by t. Given a transformation t, we can define the inverse transformation t^{-1} by exchanging the input pattern and the output pattern. A more detailed description of the transformations currently implemented in *AURORA* can be found in Section IV.

Definition 1: Given a program $p \in P$, a test suite $ts \in TS$, a set of transformations T and a coverage criterion C, we say that ts *fragilely covers* p iff there is at least one transformation $t \in T$ such that $C(t(p), ts) < C(p, ts)$.

The fact that a program is covered in a fragile way by a certain test suite ts may reduce the confidence in the measurement of the coverage because the obtained degree of coverage may be due to the particular structure of the code. On the one hand, a fragile coverage means that the developer may have written the code in a particular way in order to achieve an higher degree of coverage with a lower testing effort. On the other hand, fragile coverage is not robust with respect to future transformations. This may be a problem because usually after the application of a SPT (e.g., a refactoring transformation) the developer does not feel the need to change the test suite because no new functionalities were added, and thus the original test suite may achieve a lower coverage on the resulting code than expected. For this reason it is important to automatically perform a robustness analysis that can identify fragilely covered programs because they may need more testing, regardless of the level of coverage achieved so far.

In [5], we have defined the extended coverage as follows:

Definition 2: The *Extended Coverage* is a pair of values (a, b) where $a = C(p, ts)$ represents the usual coverage obtained by applying ts to the program p, whereas b is a fragility index such that $b \in [0, 1]$, and it measures the sensitivity of the coverage to modifications in the code structure.

If the fragility index b has a value of 0 this means that the coverage is robust. The closer b is to 1, the more fragile the coverage is.

Let p be a program, ts a test suite, and C a coverage criterion. We define $\Delta(t) = C(p, ts) - C(t(p), ts)$ where t is a transformation in a given set T. Let $pos(x)$ be a function defined as $max(0, x)$ and let $\rho(t)$ be a function that defines the weight of each transformation $t \in T$, such that $\sum_{t \in T} \rho(t) = 1$.

We have defined three different *fragility functions* that given a distribution $\Delta(t)$ compute the fragility index in the following ways:

- **Averaged fragility**: $b_{af} = pos\left(\frac{\sum_{t \in T} \Delta(t)}{|T|}\right)$
- **Weighted fragility**: $b_{wf} = \sum_{t \in T} \rho(t) * pos(\Delta(t))$
- **Worst case loss of coverage**: $b_{wc} = pos(max_{t \in T}(\Delta(t)))$

B. TXL in a Nutshell

In this section we briefly present the *TXL* language. For an extensive treatment of the subject the reader may refer to [6], [9]. *TXL* is a programming language that can be used to define code-to-code transformations. The *TXL* toolkit takes as input a textual file and a set of *TXL* files defining the grammar and the transformations, it parses the textual file according to the grammar, it produces a parse tree and then it applies query rewriting rules over the parse tree. Finally, the textual output is generated from the modified parse tree.

A *TXL* program consists usually of two different parts. We shows these parts by using an example taken from [9].

- The program contains a *grammar file* which describes the grammar of the input language. The grammar is defined by means of the usual extended BNF notation. The example below shows how the usual expression grammar can be encoded in *TXL* (in the example `[n]` represents the token of a numerical value).

```
define program
  [e]
end define

define e
    [t]
  | [e] + [t]
  | [e] - [t]
end define

define t
    [p]
  | [t] * [p]
  | [t] / [p]
end define

define p
    [n]
  | ( [e] )
end define
```

  Note that *TXL* already comes with grammars for most of the programming languages like Java, C, and C++.
- Each *TXL* program contains also a set of transformation rules. Each transformation rule consists of a target type to be transformed, a *pattern* (i.e., an example of the input that we want to transform) and a *replacement* (i.e., an example of how the transformed input will look). *TXL* defines two kinds of transformation rules, namely *rules* and *functions*. While *functions* are applied only once to their scope, *rules* are applied repeatedly on their scope until they reach a fixpoint.

  The example below shows two transformations. `rA` is a *rule* that replace two numerical values with the sum of the values. If we apply `rA` to the expression `1 + 2 + 3` we obtain `6`. `rFA` is a *function* that does exactly the same transformation as `rA`, however due to the fact that

`rFA` is a function, if we apply `rFA` to `1 + 2 + 3` we obtain `3 + 3`.

```
rule rA                          function rFA
 replace [e]                      replace * [e]
  N1 [n] + N2 [n]                  N1 [n] + N2 [n]
 by                               by
  N1 [+ N2]                        N1 [+ N2]
end rule                         end function
```

The fact that *TXL* allows the definition of transformation by using an example-driven approach is the main reason behind our choice of using *TXL* for representing source code transformations.

IV. Transformations in TXL

In [5] we have defined several transformations that can influence the robustness of the coverage achieved by a test suite; in this Section we introduce some of these transformations.

Externalized Complex Flag: The transformation t_{ecf} has the following schema (*complexBoolExpr* is a Boolean expression that contains at least one Boolean operator, and the statements between the point *A* and *B* do not change the value of *x*, and of the variables referenced in *complexBoolExpr*):

```
boolean x;                          boolean x;
...                                 ...
x = complexBoolExpr;//A             x = complexBoolExpr;
...                    t_ecf        ...
if(...x...){//B        ⇒            if(...complexBoolExpr...){
   ...                                 ...
}                                   }
```

This transformation was already identified by by Rajan et al. [4] and by Harman et al. [10]. Furthermore several refactoring patterns [11] can be partially mapped on this transformation or its inverse , i.e. *Inline Temp Variable* (in case the variable is boolean and it is inlined in an *if statement*), *Remove Control Flag*, *Introduce Explaining Variable*). This transformation was applied in Example 1.

Boundary extraction: The transformation t_{be} splits an if statement containing a condition that performs comparisons over numerical values into several *if statements* (we assume that $a \leq b$). It has the following schema:

```
                                    ...t0...
                                    if(x==a) {
...t0...                               ...t1...
if(a<=x && x<=b){                   } else if(x==b) {
   ...t1...              t_be          ...t1...
} else {                 ⇒          } else if(x>a && x<b) {
   ...t2...                           ...t1...
}                                   } else {
                                       ...t2...
                                    }
```

A simplified version of this transformation was applied in Example 2.

Flattening Conditional Expression: The transformation t_{fce} splits all the expressions used as guards in *conditional statements* until every *if statement* has only an atomic Boolean expression as a guard. It can be modeled by the following two transformation schema, the first one for conjunctions and the second one for disjunctions. In both schema $cond1$ and $cond2$ represent boolean expressions.

```
if(cond1 && cond2){                 if(cond1){
   ...t1...                            if(cond2){
} else {                  t_fce             ...t1...
   ...t2...               ⇒             } else {
}                                           ...t2...
                                         }
                                    } else {
                                       ...t2...
                                    }
```

```
if(cond1 || cond2){                 if(cond1){
   ...t1...                            ...t1...
} else {                  t_fce     } else if(cond2){
   ...t2...               ⇒             ...t1...
}                                   } else {
                                       ...t2...
                                    }
```

This transformation is a generalization of some well known transformations (for instance, the inverse of the *Consolidate Conditional Expression* refactoring pattern [11] and some transformations performed during compilation to assembly code or bytecode [12] can be represented as special cases of this transformation). It was used in Example 3.

Remove Consolidate Conditional Fragment: The transformation t_{rccf} moves the first statement after an *if statement* into the *then block* and the *else block* of the *if statement* itself . It has the following schema:

```
if(cond){                           if(cond) {
   ...t1...                            ...t1...
} else {                  t_rccf        statement;
   ...t2...               ⇒          } else{
}                                      ...t2...
statement;                             statement;
                                    }
```

t_{rccf} is the inverse transformation of the *Consolidate Conditional Fragment* refactoring pattern [11], and it was used in the Example 4.

The definition of the transformations as done above, by means of an input pattern and an output pattern, can be directly formalized by means of *TXL* programs. Currently *AURORA* is restricted only to the *Java* language, and therefore we have used the *Java 1.5 TXL* grammar[2]. Listing 1 shows how the t_{ecf} transformation can be implemented in *TXL*, whereas Listing 2 shows the implementation of the t_{fce} transformation for disjunctive expressions.

V. Tool Architecture

The *AURORA* tool is an Eclipse plugin that allows developers to execute automatically the robustness analysis of their code given a test suite. We have chosen to implement it on the basis of the Eclipse framework for three main reasons: (a) Eclipse is widely used by developers as IDE, (b) Eclipse allow us to integrate *AURORA* in an environment that already supports several aspects of the testing process (e.g., unit testing, coverage measurement, test generation), (c) a developer can add to *AURORA* new transformations, new fragility indexes, and new coverage criteria using the Eclipse Plugin mechanism and the *AURORA* extension points.

[2]The grammar can be downloaded at http://www.txl.ca/nresources.html.

```
rule main
 replace $ [repeat declaration_or_statement]
  A [repeat declaration_or_statement]
 by
  A [ecf_decl] [ecf_assign]
end rule

rule ecf_assign
 replace [repeat declaration_or_statement]
  A [id] '= B [assignment_expression] ';
  'if '( C [id] ') IfBranch [statement]
  ElseBranch [opt else_clause]
  Rest [repeat declaration_or_statement]
 where
  A [= C]
 by
  A '= B;
  if '( B ') IfBranch ElseBranch
  Rest
end rule

rule ecf_decl
 replace [repeat declaration_or_statement]
  'boolean A [id] '= B [expression] ';
  'if '( C [id] ') IfBranch [statement]
  ElseBranch [opt else_clause]
  Rest [repeat declaration_or_statement]
 where
  A [= C]
 by
  'boolean A '= B;
  if '( B ') IfBranch ElseBranch
  Rest
end rule
```

Listing 1. t_{ecf} transformation

```
rule main
 replace $ [statement]
  A [if_statement]
 by
  A [extractPar] [flat]
end rule


rule extractPar
 replace [if_statement]
  'if '( '( A [expression] ') ')
  IfBranch [statement]
  ElseBranch [opt else_clause]
 by
  'if '( A ')
     IfBranch
  ElseBranch
end rule

rule flat
 replace $ [if_statement]
  'if '( A [conditional_and_expression]
    '|| B [conditional_and_expression]
        C [repeat or_conditional_and_expression] ')
  IfBranch [statement]
  ElseBranch [opt else_clause]
 by
  'if '( A ')
    IfBranch
  'else
     'if '( B C ')
        IfBranch
      ElseBranch
end rule
```

Listing 2. t_{fce} transformation

The robustness analysis process is shown in Algorithm 1. The process takes as input a program p, a coverage criterion C, a set of transformations T, a test suite ts and a fragility function f and returns the value of the extended coverage E (i.e., the pair of the usual coverage $C(p, ts)$ and the value of the fragility index). The process can be easily extended to handle the robustness analysis over more than one coverage criterion/fragility function (we simply iterate the execution of the Algorithm 1 over the set of coverage criteria/fragility functions).

Algorithm 1: Robustness Analysis process

Input : Program p, Criterion C, Transformations T, Test suite ts, Fragility Function f
Output: Extended Coverage E
begin
  $n = |T|$;
  $\Delta[n]$;
  $c = C(p, ts)$;
  for $t \in T$ **do**
    $p' = \bar{t}(p)$;
    $c' = C(p', ts)$;
    $\Delta[i] = c - c'$;
  return $(c, f(\Delta))$;

The process is very simple, however executing it manually would be tedious and error prone for several reasons:

- if the transformations are done manually they are time-consuming and error prone (especially on large programs),
- if the transformations are done by means of tools like *TXL*, the transformed files have to be imported each time in the IDE because they need to be built/compiled again,
- code coverage tools usually require that the code under test is instrumented, and thus the transformed code should be instrumented each time after the transformation phase,
- tools like *TXL* and code coverage tools are not usually integrated in the same environment and thus moving source files from one environment to the other is time consuming,
- the test suite ts has to be executed for each triple of transformation, coverage criteria and fragility index,
- the results of the analysis are not immediately understandable because we only obtain a bunch of coverage results. The results have to be aggregated and processed in order to obtain the extended coverage.

We have created *AURORA* in order to automate the robust-

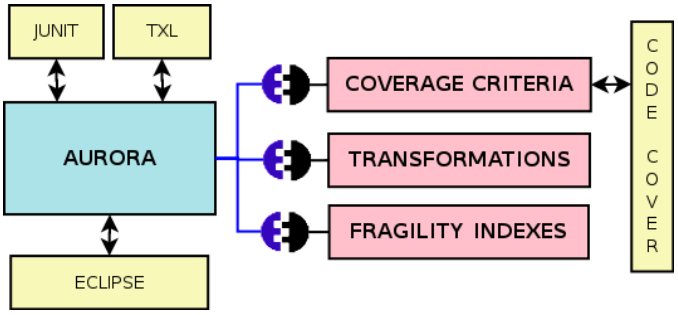

Fig. 1. Architecture of *AURORA*

ness analysis process and to solve the issues highlighted above. The aim of *AURORA* is to provide an environment that testers can easily use for executing the robustness analysis of their projects with a couple of clicks. In our opinion only tools like *AURORA* can lead to a concrete adoption of the robustness analysis outside the research environment.

The architecture of *AURORA* is shown in Figure 1. Currently *AURORA* supports only the Java language and it makes use of several existing tools:

- *TXL* is used to automate the transformation of the program under test. Each transformation is thus formalized as one or more *TXL* programs (some examples of transformations were presented in Section IV). By using *TXL*, we can concentrate only on the formalization of the transformations, without caring about how they are actually performed.
- We require that the test suite is provided as a set of *JUnit*[3] tests. We then use *JUnit* to automate the execution of the test suite.
- The *Eclipse* framework is used to automatically rebuild the source code just after each transformation.
- *CodeCover*[4] is used to automatically measure the coverage achieved by a certain test suite.

A key point of the *AURORA* architecture is its extendibility, which is achieved by using the concept of *extension point* provided by the *Eclipse* framework. We have defined three kind of extension points. First of all, we have defined an extension point *AURORA.transformation* that allows the tester to add new transformations to the set of transformations used by *AURORA* in the analysis. This is very important because the fact that a coverage is fragile or robust strongly depends on the set of transformations T one considers. With a small set T any coverage is likely to be robust, but with a large T only the best test suites will provide the required degree of coverage and robustness. Each tester should personalize the set of transformations used in the analysis with transformations that will likely be applied on the program in the future or with transformations that was likely applied on the program in the past. *AURORA* currently implements the four transformations presented in Section IV.

[3]JUnit - http://www.junit.org/
[4]CodeCover - http://codecover.org/

AURORA allows also the definition of new fragility indexes by using the extension point *AURORA.index*. Currently *AURORA* implements the three fragility functions presented in Section III-A, namely *Averaged fragility*, *Weighted fragility* and *Worst case loss of coverage*.

Another important extension point is *AURORA.criterion* that allows the tester to add new coverage criteria to *AURORA*. We implemented four coverage criteria, namely *Statement Coverage*, *Condition Coverage*, *Decision Coverage* and *MCDC*. The coverage measurement is done by leveraging the capabilities offered by the *CodeCover* tool. New coverage criteria may be added by using tools different from *CodeCover* or by manually writing components that measure the coverage.

Figure 2 shows the results of the robustness analysis of the WBS Java program. WBS is a Java implementation of the wheel brake system example found in ARP 4761 [13] and it has already shown fragility problems in [5] (some results are different from [5] because in this case we have considered only three transformations).

VI. Related Work

The concept of code coverage was introduced by by Miller and Maloney in 1963 [1], although also Senko introduced a similar concept [2]. Since then, various notions of code coverage (i.e., *coverage criteria*) have been proposed as a measure for test suite quality, including statement coverage, branch coverage, method coverage, MCDC, and others [14].

Although these criteria cannot guarantee the correctness of the program under test, they can be used as a measure of the adequacy of the testing activity and as an indication of how much of a program is tested by the current test suite. Test suites that satisfy certain coverage criteria may be required in order to accept commercial software. For instance, the Modified Condition Decision Coverage (MCDC) [15], is required for safety critical aviation software by the RCTA/DO-178B standard.

The assumption behind the use of coverage criteria as a adequacy measurement is that a test suite can reveal a fault only if it executes the portion of code that contains the fault, and thus an higher level of coverage (which indicates that a bigger portion of the code is exercised by the test suite) should correlate with a higher number of revealed faults, although other factors may influence the actual outcome [16]–[19].

It is well known that coverage criteria can be very sensitive to code structure both if they are used for measuring test adequacy and if they are used for test generation. Regarding the adequacy, there are several works arguing that code coverage is not robust to code structure transformations. In [7], Marick et al. show how very simple transformations (like adding a new empty line) can confuse code coverage tools. More severe issues are presented in [4]. In that paper, Rajan et al. show that MCDC metric is highly sensitive to the structure of the implementation and can therefore be misleading as a test adequacy criterion. They present six programs in two versions each: with and without expression folding (i.e., inlining). They find that the same test suites performed very diversely on

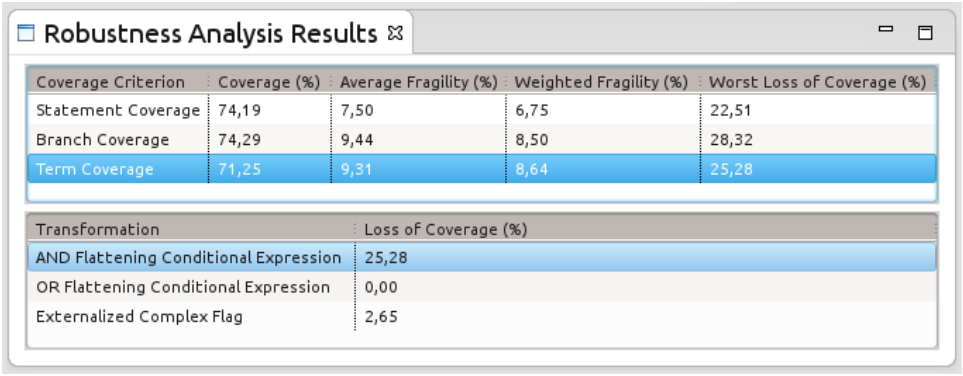

Fig. 2. Robustness Analysis Results

the two versions. In [5] we have proposed a framework that can be used to evaluate the sensitiveness of the coverage to code structure changes. We have also shown that several structural coverage criteria suffer from this problem. We have experimented how fragility affects existing code and test suites. Furthermore we have found that even industrial projects suffer from this kind of problems.

Regarding test generation, Staats et al. [20] show that test suites generated specifically to satisfy coverage criteria achieve poor results in terms of effectiveness, whereas the use of coverage criteria as a supplement to random testing provides an improvement in the effectiveness of the generated test suites.

So far, the main solution in the literature to overcome coverage criteria weaknesses has been trying to introduce more powerful and tough testing criteria [21].

Transformations and code coverage is studied by Weissleder [22]. In this case the transformation is used to obtain information of the coverage over the original code from the information about the coverage over the transformed code. The goal is to find a transformation such that if a test suite achieves the coverage C_1 over the transformed code, than the same test suite achieves the coverage C_2 over the original code. In this case C_1 *simulates* the coverage C_2.

The fact that transformations can disrupt coverage is also tackled by Kirner [12], [23], he addresses the challenge of ensuring that the structural code coverage achieved for a program P is preserved when P is transformed. If the transformation fulfill certain properties then we know that the coverage achieved on the original program is preserved on the transformed program.

VII. Final Remarks about Extended Coverage

Once the tester has measured the extended coverage achieved by a test suite ts over a program p with respect to a set of transformations T and a coverage criterion C, what are the information he/she can infer from the resulting value (a, b)? And, moreover, how should he/she react?

In this section, we are going to explain some intuitive considerations about these two topics and the meaning of the extended coverage metric. We do not claim that this section provides a thorough and extensive treatment of the subject, nor that these considerations are supported by empirical results. Indeed, we plan to extend our work on extended coverage with an accurate study of these considerations based on empirical data.

We assume in this section that the client has fixed a coverage threshold k, and has given a fixed set of transformations T. For simplicity's sake, we consider the *worst case loss of coverage* fragility index.

If the obtained fragility b is equal to 0, this means that the achieved coverage a is completely robust. This means that no matter which transformation $t \in T$ is applied to the program p, the coverage achieved over the transformed program $t(p)$ is always greater than a. So if $a \geq k$, then we are sure that our test suite has met the coverage requirements requested by the client, and, thus, we can safely stop the testing. On the other hand, if $a < k$ we know that we have to continue the testing process, since the test suite ts does not meet the coverage requirement on the program p.

Everything becomes more complex in case the fragility b is more than 0, i.e., the coverage a is fragile. Regardless the level of coverage a achieved by ts on the program p, this means that there is at least a transformation $t \in T$ such that the coverage achieved by ts on $t(p)$ is lower than a. This may cause some problems since the client's request is that all the possible transformed programs have to satisfy the coverage requirement. So it is easy to see that if $a - b \geq k$, then we are sure that every possible transformation $t \in T$ generates a transformed program that satisfies the coverage requirement, and so we can safely stop the testing process. On the other hand, if $a - b < k$, then there is at least a transformed version of the program (or the program itself) that does not reach the coverage threshold requested by the client, and thus we have to continue the testing process either by increasing the coverage or by decreasing coverage fragility.

In order to increase the coverage robustness the tester can act in two ways:

1) He/she could extend the test suite with new test cases, maybe generated from a transformed version of the program. In order to help the tester in this activity, AURORA reports the loss of coverage for every single

transformation.

2) He/she could change the structure of the code in order to remove all the points that introduces fragility issues.

However, restructuring the code and removing fragility points may be not straightforward nor possible every time (this fact highly depends on the transformations in T). Code changes may decrease other quality factors like readability and maintainability. Moreover transforming the code would increase the robustness at the expenses of the coverage, which would diminish. To maintain the same level of coverage, the tester must add new tests in any case.

If we consider other fragility indexes, it is more difficult to intuitively give similar considerations, because the mathematical structure of the indexes is more complex. However in this situation, a realistic assumption is that the client may fix a desired maximum level of fragility f. In this case, we can stop the testing process safely when both the coverage and the fragility requirement are met (i.e., when $a \geq k$ and $b \leq f$).

VIII. Conclusion and Future Work

In this paper we have presented the *AURORA* tool, that allows testers and developers to automatically quantify the robustness of a test suite with respect to code structure changes.

In this paper we have shown how existing tools, i.e., *CodeCover* and *TXL*, can be integrated in order to automate the process needed to compute the extended coverage. We have also shown how several semantic preserving transformations can be effectively implemented by using *TXL*.

Although *AURORA* is still a prototype, it has proved to be very effective in ease the robustness analysis of programs.

In the future we plan to extend the set of *TXL* transformations and we aims at extending the tool also to languages different from *Java*. We also plan to use *AURORA* to study the correlation between the fragility of a test suite and its fault detection capability.

References

[1] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, pp. 58–63, February 1963.

[2] M. E. Senko, "A control system for logical block diagnosis with data loading," *Commun. ACM*, vol. 3, pp. 236–240, Apr. 1960.

[3] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic Press, 1972.

[4] A. Rajan, M. W. Whalen, and M. P. Heimdahl, "The effect of program and model structure on mc/dc test adequacy coverage," in *Proc. of ICSE*, 2008, pp. 161–170.

[5] A. Gargantini, M. Guarnieri, and E. Magri, "Extending coverage criteria by evaluating their robustness to code structure changes," in *23rd International Conference on Testing Software and Systems (ICTSS 2012)*, 2012.

[6] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190 – 210, 2006, special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA).

[7] B. Marick, J. Smith, and M. Jones, "How to misuse code coverage," in *Proc. of ICTCS'99*, Jun. 1999.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[9] J. R. Cordy, "Excerpts from the txl cookbook," in *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, ser. GTTSE'09. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 27–91.

[10] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 3–16, January 2004.

[11] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

[12] R. Kirner, "Towards preserving model coverage and structural code coverage," *EURASIP J. Emb. Sys*, vol. 2009, pp. 1–16, 2009.

[13] *ARP 4761,Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. Society of Automotive Engineers, Detroit, USA: Aerospace Recommended Practice, 1996.

[14] G. J. Myers, *The art of software testing (2. ed.)*. Wiley, 2004.

[15] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[16] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proc. of ISSTA 2009*. ACM, 2009, pp. 57–68.

[17] L. Briand and D. Pfahl, "Using simulation for assessing the real impact of test coverage on defect coverage," in *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, 1999, pp. 148 – 157.

[18] J. R. Horgan, S. London, and M. R. Lyu, "Achieving software quality with testing coverage measures: Metrics, . . ." *IEEE COMPUTER*, vol. 27, pp. 60–69, 1994.

[19] F. Del Frate, P. Garg, A. Mathur, and A. Pasquini, "On the correlation between code coverage and software reliability," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, oct 1995, pp. 124 –132.

[20] M. Staats, G. Gay, M. W. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Proc. of Fundamental Approaches to Soft. Eng. (FASE)*, 2012.

[21] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 4, pp. 367 – 375, april 1985.

[22] S. Weißleder, "Simulated satisfaction of coverage criteria on uml state machines," in *Proc. of ICST 2010*. IEEE Computer Society, 2010, pp. 117–126.

[23] R. Kirner and W. Haas, "Optimizing compilation with preservation of structural code coverage metrics to support software testing," *Software Testing, Verification and Reliability*, 2012.

Appendix

The plan for the live demonstration is the following:

1) Firstly we present some of the examples of Section II to clarify the kind of problems addressed by *AURORA*.
2) Then we show a practical example that consists of a Java program and a JUnit test suite. In this case we execute the robustness analysis manually. We first instrument the Java project with CodeCover, then we execute the test suite to obtain the initial coverage. We transform the code by directly invoking the TXL process. Then we import again the transformed code into Eclipse, we show the differences w.r.t. original one, we execute again the test suite and we compute the coverage over the transformed version of the program. The goal of this part of the demonstration is to show that, although executing the robustness analysis manually is feasible, there is a need of an automated way of executing the robustness analysis process because manual approaches are time-consuming and error prone.
3) Then we show how *AURORA* can execute the robustness analysis on the same example in an automated way and

we describe how *AURORA* presents the results of the analysis.

4) Finally we show how *AURORA* behaves with some real examples that require to apply several transformations on several Java classes. These examples, like WBS (the is a Java implementation of the wheel brake system example found in ARP 4761 [13]), TCAS (the Java implementation of a Traffic Collision Avoidance System) and ASW (a Java implementation of an altitude switch), have shown to suffer from fragility problems [5]. The manual analysis of these examples is very time-consuming, whereas the execution of *AURORA* can be done in an automated way in a reasonable amount of time.