Integrating a Category-Partition Testing Tool with Combinatorial Interaction Testing Tool To Produce T-Way Adequate Test Frames

Andrew Graff

April 27, 2021

1 Introduction

Software is the beating heart of almost every company, which makes defects in that software to be potentially very costly. Some software defects result in printing funny characters to the terminal, while others can bring company production to a halt for hours or even days, costing thousands and even millions of dollars in losses. Another critical defect may result in a security breach, which exposes the company's intellectual property that could be fatal to business altogether. That is why testing is such an important part in the process of creating defect free software. Testing software is a difficult task, which depends on the complexity of the software itself, what kind of inputs it takes, i.e., the input domain, and how many functions it performs. Depending on software, testers can choose among different testing approaches to implement the actual testing: black-box, white-box, model-based testing, to name a few. In this project, we focus on black-box testing, and more specifically on the test-case optimization: how to generate defect-revealing test cases with a realistically executable number of them.

2 Problem Statement and Proposed Solution

It is difficult to engineer a set of tests that adequately tests a particular piece of software in an acceptable amount of time. Category partitioning is a useful black-box testing method that helps systematically design test cases.

It allows the user to divide the software's input space into categories of choices. *tsl* (testing specification language) is a tool that offers a good front-end interface for the category partitioning method with a formal descriptive language to specify categories and choices on the input domain partitions. In addition, *tsl* generates readable test frames in an output file, where each test frame corresponds to a test case. However, its weakness is that it generates many frames because it computes all possible permutations of the categories of choices. This results in a large test suite, which may not run in a feasible amount of time.

Combinatorial interaction testing (CIT) addresses this problem of the brute force approach of exhaustive test case generation by offering a method for analyzing and selecting a subset of test cases that cover specific combinations of category choices. The tool we use for CIT is *casa*, which uses a simulated annealing search to find the subset of test cases with desired interactions between choices. Given a model file and constraints file as input, *casa* then outputs to a file with a set of combinations of options, i.e., category choices, that should be tested together. The weakness for *casa* is that the constraints are expressed in the conjunctive normal form, which is not how a typical user would define or express them. It is much easier and more human readable if we could write constraint as a general formula in propositional logic. Thus, it can be challenging for users to write the constraints in *casa*. Additionally, the output is a set of numbers representing each choice. It's hard to map these numbers back to actual choices. So some remapping is required back to the original inputs (e.g. *tsl* "choices").

These two methodologies and tools are separate pieces of software that require the user to create the input files to each tool separately. **The objective of this project is to combine the user-friendly input/output components of *tsl* and integrates them into the powerful test case optimization algorithm of *casa***. Upon completion of the project, we will produce a tool where a user only needs to generate the category partition test specification input file, and an easy to use t-test adequate coverage set of test frames will be generated. This will eliminate the necessity for the user to generate the complicated and bug prone inputs to the *casa* tool and remapping of the output back into the *tsl* frames.

3 Background

This project builds on previous work that addresses to two different problems: systematic test case generation and test case optimization. In this section we explain approaches for solving each problem and the tools implementing those solutions.

3.1 Category-partition Method

The *category-partition method* uses a formal test specification language (TSL) to define test cases and then generates test case descriptions according to it specification. These test case descriptions are then used to create an executable software test suite. As with any formal specification, writing the formal test specifications is an iterative approach. Depending on the size of the software system to test, the engineer may need to break the program up into smaller testable blocks. For the purposes of this project, we will use a simple example of a browser's settings for how to treat tabs. See Figure 1.

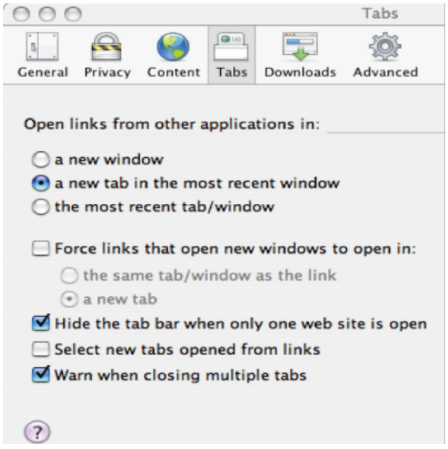

Figure 1: Sample of software feature to test tabs

As part of the process to translate this feature into the formal language of *tsl*, we need to follow a specific format for the test specification file. It specifies *categories*, *choices*, *properties*, and *selectors* where *selectors* are boolean expressions. See Figure 2 for an example.

In our example, we need to choose *categories* and *choices* within those *categories*. For example, one *category* could be 'Link', and the choices as-

```
Parameters:
        <Category 0>:
                <Choice 0>.             [property <X>]
                <Choice 1>.             [property <Y>]
                <Choice 2>.

        <Category 1>:
                <Choice 3>.             [if <X>] [property <Z>]
                <Choice 4>.             [if <X> and <Y>]
                <Choice 5>.             [single]
```

Figure 2: Format for tsl input file

sociated with 'Link' are 'new window', 'new tab', and 'current tab' pulled from the 'Open links from other applications in:' portion of the settings. We complete this step for all the available selections in this browser's 'Tabs' options. Next, we assign some *properties*, which are useful for defining constraints that should be imposed on the valid *choices* a program may accept. And then the *selectors* are defined as prepositional logic statements that select the option that is associated with the *property* that is referenced in the *selector* statement. See Figure 3 for an example.

```
Parameters:
        Link:
                new window.
                new tab.
                current tab.

        NewWindowOption:
                force links that open new windows to open in.           [property LinkOption]
                don't force links that open new windows to open in.

        NewWindowOptionOption:
                the same tab window as the link.                        [if LinkOption]
                a new tab.                                              [if LinkOption]
                n/a.

        Hide:
                hide when 1 tab.
                don't hide.

        SelectTab:
                select new tab opened from link.
                don't select new tab opened from link.

        Warn:
                warn when closing multiple windows.
                don't warn when closing multiple windows.
```

Figure 3: An input file for *tsl* using our 'Tabs' example.

After *tsl* processes this input file, it produces 96 total test frames for all possible combinations of our 'Tabs' example, restricted by the set of constraints. This is a small example with very few choices. Another, more complex system such as an airplane cockpit with many switches and inputs can produce a large amount of possible combinations, which is not feasible to execute and hence to test this system completely at a certain level of details.

Removing categories and or excluding some values might result in smaller but less powerful test suite. Thus, we look for another way to reduce the number of test sets down to a feasible number, yet maintaining the desired level of detail. The original *tsl* tool is written in C and consists of 12 files and 1,125 lines of code.

3.2 Combinatorial Interaction Testing

Combinatorial interaction testing is a test case selection approach that relies on the principle that a large portion of faults can be detected with a smaller subset of test cases that insure interaction of different choices for different categories. *casa* is a tool that finds test cases with a *t-way* covering combinations of choices. The tool uses the simulated annealing search algorithm in order to find the minimal set of such test cases.

The tool takes two files as input: one that specifies the *t-way* interaction level desired along with all possible inputs and another that encodes constraints on those inputs. The output is the subset of combinations that satisfy the selected *t-way* interaction level. This allows to identify the subset of tests necessary to get adequate t-way testing coverage. In *casa*, the options must be manually written by the user as well as the constraints file, which accepts constraints in a conjunctive normal form (CNF). The user can find it difficult to translate from the logical formula of *tsl* to its CNF representation. In addition, the output needs some manipulation to translate it into a readable testing specification.

The *casa* tool's *.citmodel* input file has the following format. The first line specifies the strength of the T-way pairs to cover, which usually is limited to 2-way or 3-way interaction. The second line specifies the number of *categories* we have broken the program into. And the third line specifies how many *choices* within each category can be chosen. The enumeration starts from the first *category* to the last, e.g., Figure 4.

Next, the *.constraints* file contains the constraints written as a conjunction of disjunctions, i.e., CNF. The first line denotes how many disjunctive clauses (constraints) are defined for this model. Each conjunctive clause consists of a pair of lines, the first of which specifies how many terms are in the disjunctive clause. The second line of the pair is the disjunctive clause itself. See Figure 5 for a demonstration.

Finally, let us consider the output format from the *casa* program. The first line contains the number of combinations generated for the given *.cit-*

model and *.constraints* input followed by that number of lines, each a different input combination. Each digit in the combination represents the *choice* from the *category* in order of their defined input. For example, with an input of '2 3 4 5' in our *.citmodel*, the first entry of the output line could contain '0' or '1'. The second entry could contain '2', '3', or '4'. The third entry would contain '5', '6', '7' or '8', etc. This output could be used to generate test frames by associating each entry in the line with its corresponding *choice*. See Figure 6. The *casa* tool consists of 87 files with 9,139 lines of code and is written in C++.

```
2
6
3 2 3 2 2 2
```

Figure 4: Example of *casa* .citmodel input file

```
2
3
- 3 + 5 + 6
2
- 4 + 7
```

Figure 5: Constraints file for the 'Tabs' example with 2 rules. If option 3 is chosen, then choose 5 or 6. If option 4 is chosen, then choose 7 only.

```
9
2 3 6 8 11 12
0 4 5 8 10 12
0 3 7 8 10 13
0 3 6 9 11 13
1 4 6 8 10 12
2 4 5 9 11 13
2 4 7 9 10 12
1 3 5 9 10 13
1 4 7 9 11 13
```

Figure 6: Output from *casa*

4 Preliminary Work

Some preliminary work has been done on this project in order to determine the feasibility of the idea to combine the *tsl* and *casa* tools into a single, easy to use tool that combines the *tsl*'s front-end and *casa*'s powerful search engine. A goal of the project is to simplify and reduce the effort a user needs to put in to use the new tool. Source code was acquired for both tools and some initial analysis of the code was done in order to understand how they work and where some work may be needed to combine them. As it turns out, the *tsl* tool is written in basic C and *casa* is written in C++. Converting *tsl* to C++ would be beneficial for combining the code bases into a single program. So *tsl* was updated to support compilation with the g++ compiler.

As part of the simplification of the tool, only the category partition test specifications should be created by the user as a *.tsl* file. The tool will do the rest to generate the adequate coverage set of test frames. Since *casa* takes as input a *.citmodel* file, this preliminary version of *tsl* generates one based on the specification file passed in. The support for generation of the *.constraints* file will be added at a later time as part of the proposed work. In order to create the *.citmodel*, we need to be able to keep track of the parsed information from the specifications file.

In order to achieve this, a new struct was added to *tsl/structs.h* called `container` that keeps track of the list of non-single `Choices` as a vector of `Choice` struct pointers. This vector is used to look up the choices and categories later for printing the test frames. In addition, a `parent` pointer was added to the `Choice` struct so that the `Category` information can be referenced when performing the lookup into the `Choice*` vector.

Rather than output directly all possible test frames, a new function called `make_citmodel()` was written in *tsl/output.c* to write a *.citmodel* file used as an input to *casa*. The function `generator(Flag flags)` was also modified to call `make_citmodel()`, and then make a system call to casa passing in the *.citmodel* file for the input. Then, another function created called `process_output_file(string filename)` processes the file output by *casa* to generate the test frame final output. Ideally there should be no file required, but this is a preliminary draft of the final solution and this will be addressed in the final version of the project.

5 Proposed Remaining Work

The goal is to simplify the work of a user who needs to use these tools to generate test frames for testing. Having a simple user interface and easy to understand input file is key. *tsl*'s input format is easy to configure and more user-friendly, so the new tool will inherit this format. Our preliminary work has demonstrated feasibility of *tsl* to interact with *casa*.

The remaining step is to convert the logic formula from the *properties* and *selectors* into CNF accepted by *casa*. One converter in particular, Vojkan Cvijovic's CNF converter is available at GitHub (https://github.com/Vojkan-Cvijovic/CNF-Converter/tree/master/src/converter, and it's written in C++ which matches the language for *tsl* and *casa*. If this tool does not serve the purpose, another CNF conversion tool will need to be found or a new one using well know algorithms for converting a propositional formula into its CNF will need to be coded in the *tsl* tool to output the *.constraints* file. The final output from the combined tools should be a set of test frames whose number has been reduced in size to provide adequate t-way coverage testing of the system described in the input of the program that can be quickly actionable from the user to use.

My project can be found at the following url https://github.com/Panik-Kontrol/MastersProject. The top-level directory contains my *.tex* files for this proposal as well as a place holder for the final paper (not started yet). It will also be the location where I store the slides I will present for the proposal next week. The `src/` directory contains the source code for both the *casa* tool and the *tsl* tool that I have and will be modifying. The `images` directory contains all the figure images used in this proposal.

5.1 Timeline

Here is my tentative timeline

- May, 2021 – Study Cvijovic's CNF conversion tool and implement converter component.
- June, 2021 – Finish Master's Project description document and defend Master's Project