A Case Study on Combining Model-based Testing and Constraint Programming for Path Coverage

M. Carmen de Castro-Cabrera, Antonio García-Dominguez e Inmaculada Medina-Bulo

Department of Computer Science, University of Cádiz, Spain
University of York, United Kingdom
UCASE Research group

ICSEA 2022,The Seventeenth International Conference on Software Engineering Advances October 16, 2022 - October 20, 2022

Presenter

Presenter's bio

- Researcher at the Computer Engineering Department, University of Cádiz (Spain) She belongs to the UCASE research group on Software Engineering (TIC-025), at the University of Cadiz in Spain. The group facilities are located at the School of Engineering.
- She is currently working as Associate Professor with the Department of Computer Science and Engineering, She has mainly centered her research on testing techniques,specially Metamorphic Testing, Model-Based Testing, mutation testing and Constraint Programming. She participates in some research projects, mainly involved in software engineering .

Research Profiles

ORCID: 0000-0003-4622-5275
SCOPUS: 57210792964

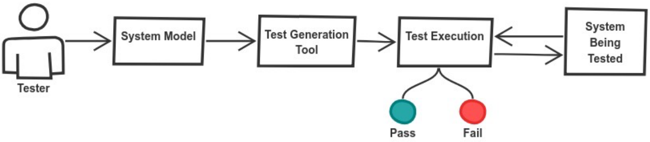
Table of contents

Introduction

Model-based Testing

Model-Based Testing, MBT

- Provides a representation of a program by abstracting from the details of implementation and the programming language used.
- Model of system behavior.
- Allows to **visually walk through the model**.
- Coverage criteria can be checked for a set of paths on the model.
- Tools that automate and facilitate model representation and execution. (For instance, Graphwalker).



Constraint programming

Constraint programming, CP

- Solving problems represented by: variables and constraints.
- Specifying conditions for a certain action to be carried out.
- It is used to implement constraints that define the paths.
- **Test case generation**.
- Tools to implement constraints and generate test cases.(For example, MiniZinc)

Proposal

MBT+CP: Generic steps

Steps

1. *Model-Based Testing*: Creation of a **model of the program behaviour as a finite state machine**, adding the **input and output constraints for each transition**. Execution (path coverage).
2. *Constraint Programming*: from the paths generated, collecting the input constraints, mapping to **implement as CP models covering the paths**; **execute the implemented models to obtain the test suite**; **run the program to be tested with the test suite**, using the output constraints from the relevant path as the test oracle.

Formalisation of input and output conditions as a CSP

Constraint Satisfaction Problem (CSP)

Definition 1. A constraint satisfaction problem (CSP) is an ordered triple (X, D, C) where:

- X is a set of n variables $x_1, ..., x_n$.
- $D = \langle D_1, \ldots, D_n \rangle$ is a vector of domains (D_i is the domain containing all the possible values that the variable x_i can take).
- C is a finite set of constraints. Each constraint is defined on a set of k variables by means of a predicate that restricts the values that the variables can take.

Definition 2. An assignment of variables, (x, a), is a variable-value pair representing the assignment of the value a to the variable x. An assignment of a set of variables is a tuple of ordered pairs, $((x_1, a_1), ..., (x_i, a_i))$, where each ordered pair (x_i, a_i) assigns the value a_i to the variable x_i.

Formalisation of input and output conditions as a CSP

Constraint Satisfaction Problem (CSP) II

Definition 3. A solution to a CSP is an assignment of values to all variables such that all constraints are satisfied.

`cat` utility

- X is represented by the variable x
- D represents all the possible values of the variable x, according to the defined domain (strings)
- C is formed by all the propositions applicable to the variables and which establish restrictions on their values (e.g.,: is word(x) belongs to C; it is true when x is a string that contains alphabet character.

Mapping intermediate code to MiniZinc constraint

TABLE I: MAPPING INTERMEDIATE CODE TO MINIZINC CONSTRAINT.

Proposition	MiniZinc constraint
$\mathit{input\_string}(x)$	`var x: string;`
$\mathit{contains}(x, s)$	`str_contains(x, s);`
$\mathit{is\_word}(x)$	`str_len(x) > 0; str_alphabet(x, \abcd...");`
$\mathit{is\_nonprinting}(x)$	`var x: string of {"\n","\b"," "}; str_len(x) > 0; str_alphabet(x, "\n\b...");`
$x > 0$	`var x: int; x > 0;`
$\mathit{output\_string}(x)$	`var x: string;`

Case study

MBT tool: GraphWalker

GraphWalker

- Open source MBT tool.
- **Studio** Editor: allows to create, edit and run models visually.
- The model has: **start, generator,** and **stop condition**.
- Allow to define *actions* and *guards* (with programming code).
- Configurable options (e.g. type of coverage and run mode: `random(edge_coverage(100))`).

IARIA

UCA Universidad de Cádiz

GraphWalker diagram steps

Steps

1. Modeling `cat` in GraphWalker.

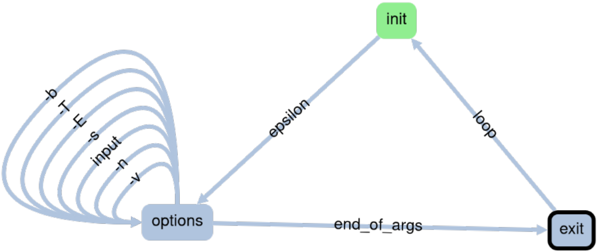


Figure: GraphWalker diagram for the `cat` program.

GraphWalker diagram steps II

Steps II

2. Adding the input and output constraints of each transition in propositional logic language.

Example: `cat` with option -E

- `cat` with the option `-E`, which displays $ character at end of each line,
- Action: input condition that is $input\_string(s) \wedge contains(s, "\backslash n")$,
- It means that must have at least two lines (with an end line character) in order to observe its effect on lines in the output.
- Action: output condition is $output\_string(o) \wedge contains("\$", o)$.

IARIA

UCA Universidad de Cádiz

GraphWalker diagram steps III

Steps III

3. Validating the GraphWalker model through execution

random(edge_coverage(100) && reached_vertex(exit))

- `random(edge_coverage(100) && reached_vertex(exit))`: a random walk that achieves full edge coverage and reaches the `exit` state.
- The model was then run and visited elements were colored: by the end of the execution, the entire diagram had been colored.
- Furthermore, by running the model in the GW CLI (GraphWalker Command-Line Interface), we obtain a set of paths that will result in a set of test cases that meets the coverage conditions.

CP: MiniZinc

MiniZinc

- Open source constraint modeling language.
- It can be used to model constraint satisfaction problems and high-level, solver-independent optimization problems
- Graphical interface, examples and documentation.
- MiniZinc model consists of variables and parameter declarations and a set of constraints.
- We used MiniZinc with the G-Strings solver, a variant of the Gecode solver which can solve constraints over strings.

CP: MiniZinc

MiniZinc steps

1. Implementing each path's input constraints as CP models in MiniZinc.
2. Running the implemented model in MiniZinc to obtain the test cases of the defined set.

```
%%% GENERIC INPUT VARIABLES (all options)
var string of {"a","b"..."z",":","/",".","\n","\b","␣","\t"}:
    input_string;
%%% CONDITIONS − INPUT CONSTRAINTS
constraint str_len(input_string) > 0; % must not be empty
%%% −T option input constraint: must have TAB
constraint str_contains(input_string, "\t");
%%% −E option input constraint: must have linebreak
constraint str_contains(input_string, "\n");

solve satisfy;
```

Listing 1: Excerpt of MiniZinc model for running `cat` with -T and -E.

Test cat against the obtained test cases

cat execution

- The output file of the previous execution (InputTEoptions.txt) is the input to execute `cat` with the indicated options: `cat -TE InputTEoptions.txt`
- The output conditions specified in the GraphWalker diagram model provide an oracle to check whether the execution of `cat` with the corresponding options and the file containing the input information generated by MiniZinc is met.
- The constraints on the outputs have been checked manually
- In addition, the source code coverage analysis and profiling tool, `gcov` was used to find out the percentage of executed code.

TABLE II: RESULTS OF cat TEST SUITE EXECUTION, INCLUDING LINE COVERAGE OVER ITS 281 LINES

Test suite	Options	Cumulative line coverage (%)
Tc_1	-E ,-s	38.79
Tc_2	(none)	38.79
Tc_3	-s ,-b	43.71
Tc_4	-E ,-T ,-n, -b, -s	46.62
Tc_5	-E ,-T, -v ,-b ,-s	51.25

Conclusions

Conclusions and future work

Conclusions

- MBT and CP techniques have been combined to obtain a test cases suite with path coverage.
- The process has been developed through a case study of `cat` GNU Coreutils.
- GraphWalker: to represent the state model and execute it visually.
- GraphWalker has produced a set of paths providing sets of input constraints.
- Input constraints will be solved through CP using the MiniZinc to generate the test cases covering the paths.
- Validating results by running cat with the corresponding options and the generated inputs, while measuring test coverage with gcov

Future work

- Testing this process with other larger programs and in other contexts.
- Introducing more formalisation and automation on the constraints.

Thanks

maricarmen.decastro@uca.es