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

M. Carmen de Castro-Cabrera
Department of Computer Science
University of Cádiz
Cádiz, Spain
email:maricarmen.decastro@uca.es

Antonio García-Dominguez
Department of Computer Science
University of York
York, United Kingdom
email:a.garcia-dominguez@york.ac.uk

Inmaculada Medina-Bulo
Department of Computer Science
University of Cádiz
Cádiz, Spain
email:inmaculada.medina@uca.es

***Abstract*—In the context of advances in software testing, this paper is related to the generation of test suites and black box testing. Test coverage is a popular technique to evaluate test quality: a test suite that can exercise a large part of the paths in a program will provide a high confidence that the program will work as expected. One way to obtain test suites with high path coverage is through model-based testing, and specifically using a diagram, which represents the various states in the program and their possible transitions. Using model-based testing, it is possible to obtain a set of paths that achieves a set of coverage criteria. Constraint programming approaches can then be used to turn each path into a test case with its program inputs and oracle, by solving the input and output constraints collected along the path. In this paper, we present a case study on this combination of techniques, by developing a state diagram that represents a popular piece of software, and using it to derive a set of test cases that achieve path coverage through constraint programming. Specifically, two tools have been used: GraphWalker (to obtain a set of paths through the program) and MiniZinc (to obtain test inputs by solving the constraints along the paths). We have obtained good results on the `cat` GNU Coreutils utility, especially in terms of ease of understanding the logic of the software through the GraphWalker diagram and its visual execution, as well as the agility in obtaining the restrictions to be solved to achieve path coverage.**

***Index Terms*—Keywords–Model-based Testing; Path Coverage; Constraint Programming; GraphWalker; MiniZinc.**

I. Introduction

Model-Based Testing (MBT) is one of the most frequently used techniques in software testing, because it can represent the program in a schematic and simplified way, abstracting from the implementation details or the language used. As defined in [15]: "Model-based testing is an efficient and adaptable method of testing software by creating a model describing the behavior of the system under test". In addition, it is easier to check the coverage criteria on the program model. This technique is used at different levels of software testing, especially in system testing, and with a broad range of levels of automation with various tools [1] [10]. One of the tools is GraphWalker (GW) [11], which provides graphical model design and execution capabilities. On the other hand, in order to test a software, it is useful to have a set of test cases that meet some coverage criteria. To achieve this coverage, it is usual during test case generation to establish some restrictions on the inputs and expected outputs of the program. In order to solve the constraints needed to follow a specific path and identify the expected outputs, one option is to use Constraint Programming (CP). In [8], CP is described as "The basic idea in constraint programming is that the user states the constraints and a general purpose constraint solver is used to solve them. Constraints are just relations, and a Constraint Satisfaction Problem (CSP) states which relations should hold among the given decision variables". Like MBT, CP has tools to implement the constraint models: Kjellerstrand surveyed the available tools in [9].

In this paper, MiniZinc (MZ) [6] will be used to implement the constraint models of the paths obtained with GW. A combination of both techniques has the advantage of automating the testing process, simplifying the visual understanding of the logic of the program to be tested, the execution of the model itself, and the generation of a test suite that meets certain path coverage criteria [12]. We propose an example of using both techniques for the `cat` utility.

This paper is organized as follows: Section II introduces the proposal and the concepts of the techniques used, Section III develops the `cat` case study, explaining the tools used and the process followed to obtain the model and the set of test cases. Section IV summarizes related work and finally, Section V presents conclusions and future work.

II. Proposal

This section describes in a generic way the process followed to generate test cases suites with path coverage, starting from a model of the program under test, and then CP to implement the constraints of the paths to cover. We have based this only on the execution and specification of the program (without accessing the source code). The starting point is to create a model of the software's behaviour, such as a Finite State Machine (FSM). The FSM can *transition* from one state to another in response to some inputs. In this paper, states are represented by nodes (or vertices) and transitions by edges. The proposed combination follows these steps:

1) **Model-Based Testing**: create a model of the program behaviour as a finite state machine, by defining and labeling states and transitions; add input and output

 ISBN: 978-1-61208-997-3

constraints for each transition expressed by propositional logic; run the model with a given path generator and coverage criteria (e.g., full edge coverage and random search), achieving a set of paths that fulfil that coverage as output of this step.

2) **Constraint Programming**: from the paths generated previously, collect 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 obtained in the previous step, using the output constraints from the relevant path as the test oracle.

A. Formalisation of input and output conditions as a CSP

This paper considers the example given as a CSP (Constraint Satisfaction Problem) by declaring constraints on the problem area (the space of possible solutions) and consequently finding solutions that satisfy all the constraints. In order to formally represent the input and output conditions included in the GW model diagram, the notation of a CSP proposed by Barber et al. in [7] will be followed. The focus in this paper is on problems with finite domains, which consist of a finite set of variables, a finite domain of values for each variable and a set of constraints that limit the possible values that these variables can take in their domain. The resolution of a CSP, as usual with representation systems, consists of two phases:

1) Modelling the problem as a constraint satisfaction problem (CSP).
2) Processing the constraints by means of search algorithms: this step in our approach is achieved by MiniZinc and the solvers integrated in it.

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.

Definition 3. A solution to a CSP is an assignment of values to all variables such that all constraints are satisfied. So, a solution is a tuple containing all the variables of the problem.

Specifically for our example `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), and 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 characters). Table I shows the notation to express our CSP logic propositions in GW and their mappings to MZ.

III. Case study

To show our approach, the GNU Coreutils `cat` tool has been selected [4]. Its manual page [5] shows that it takes a number of optional flags, followed by zero or more file paths, and it will be used as reference material to design a set of black-box tests. Based on the options, inputs and outputs when executing `cat`, we will draw the diagram of the program model. The rest of this section will explain how the steps in Section II will be applied to `cat`.

A. GraphWalker diagram

GW is an open-source MBT tool [11] available in three versions: web-based, console-based, and Eclipse plugin.

From a model, GW will generate a walk through it. A model has a start element, a generator that determines how the next step in the path is chosen, and an associated stop condition that tells GW when to finish its execution. This is formulated through an expression such as `random(edge_coverage(100))`, which represents a random walk aiming for 100% edge coverage. GW models can contain arbitrary JavaScript-based code for the *actions* and *guards* of their transitions. A transition will only activate and will execute its action if its guard evaluates to a true value. For example, in `cat`, a guard would be, if the `-b` option overrides the `-n` option, in order to execute the `-n` option, it must be checked that the `-b` option has not been previously executed. This is carried out by checking whether the output condition of option `-b` is already in the condition list of the path.

1) Modeling `cat` in GW: Figure 1 shows the GW model for `cat`. Three states have been considered: an initial state (`init`), a state that includes the options (`options`) and an exit status (`exit`). There is an empty (epsilon) transition from `init` to `options` that represents the start of the inputs for the test case, an `end_of_args` transition from `options` to `exit` that marks the end of the list of arguments, and a `loop` transition that returns to `init`, signalling the start of the next test case. The `loop` transition is needed to allow a single execution of GW to produce multiple tests that all together meet the required coverage criteria: without it, only one test case would be generated. There is a transition for each of the `cat` input arguments (`-b, -E,-n, -s, -T, -v, input`), that loop back to the same `options` state, allowing multiple options and inputs to be accumulated. In addition to these options, other options appear in the manual such as `-A`,

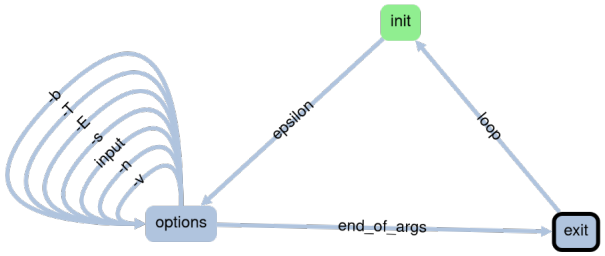

Fig. 1: GraphWalker diagram for the `cat` program.

TABLE I: MAPPING INTERMEDIATE CODE TO MINIZINC CONSTRAINT.

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

`-e` or `-t`, but they have been simplified for clarity and because they are shorthand combinations of the previous ones.

2) Adding the input and output constraints of each transition in propositional logic language: For each of the `cat` options, input and output restrictions were modelled through transition guards and actions. These guards and actions were defined so that valid combinations of options are tested, and unnecessary repetitions of options are avoided to keep the length of the list of arguments within a readable size. For each option's transition, a guard is used to ensure that it is not already included in the path, and an action records the additional constraints for the current path. Input and output constraints in the GW model are expressed as conjunction of propositions. For example, for the option `-E`, which displays $ character at end of each line, the action records the input condition that is $input\_string(s) \wedge contains(s, "\backslash n")$, meaning that must have at least two lines (with an end line character), in order to observe its effect on lines in the output. The added output condition is $output\_string(o) \wedge contains("\$", o)$.

3) Validating the GW model through execution: GW was given the expression `random(edge_coverage(100) && reached_vertex(exit))` to generate a walk over the graph: 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.

B. Constraint programming with MiniZinc

MiniZinc is an open source constraint modeling language. It can be used to model constraint satisfaction problems and high-level, solver-independent optimization problems. The default distribution of MiniZinc includes a graphical interface for ease of use, as well as examples [6]. The data file is separate from the model file. Both are translated to obtain a FlatZinc model, which depends on the specific solver used. A MiniZinc model consists of a set of variables and parameter declarations, followed by a set of constraints. Given that `cat` is a text-centric program, we used MiniZinc with the G-Strings solver, a variant of the Gecode solver which can solve constraints over strings [13] [14].

1) Implementing each path's input constraints as CP models in MZ: A base template for all constraint models in `cat` was created, containing the variables and generic constraints that needed to be met for all paths. This was followed by a set of constraint model fragment for each option of `cat` achieved from GW output and mapping to MZ language, as shown in Table I. For instance, Listing 1 shows the model for the input constraints when using both `-T` and `-E` as options.

```
%%% GENERIC INPUT VARIABLES (all options)                          1
var string of {"a","b"..."z",":","/",".","\n","\b","␣","\t"}:      2
    input_string;
%%% CONDITIONS – INPUT CONSTRAINTS                                 3
constraint str_len(input_string) > 0; % must not be empty          4
%%% -T option input constraint: must have TAB                      5
constraint str_contains(input_string, "\t");                       6
%%% -E option input constraint: must have linebreak                7
constraint str_contains(input_string, "\n");                       8
                                                                   9
solve satisfy;                                                     10
```

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

2) Running the implemented model in MZ to obtain the test cases of the defined set: The MZ models can then be solved from the command line, using `mzn-gstrings modelName.mzn`. The execution of the model in Listing 1 can be redirected to a file (e.g., a text file named `InputTEoptions.txt`) as a test case of the previously obtained suite with GW. It will generate a test case to check the execution of `cat` with the options `-TE`. Therefore, it must have at least one tab character and two lines (an end-of-line character).

C. Test `cat` against the obtained test cases

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 GW diagram model (step 1) provide an oracle to check whether the execution of `cat` with the corresponding options and the file containing the input information generated by MZ is met. Continuing with the example above, the output conditions were $output\_string(x) \wedge contains("\backslash t", x) \wedge contains("\$", x)$. It has been verified that the inputs are indeed generated correctly by meeting the input constraints for the combined options of each path. It is important to note that our configuration of GW produces random walks, and therefore each execution may produce a different set of tests that achieves the same coverage criteria. The following results are from a specific execution of GW, using

 ISBN: 978-1-61208-997-3

the command-line version to execute the JSON file produced by the web-based modeling tool (GW Studio). The execution produced input and output conditions for 5 test cases, which were then solved with MiniZinc to generate the specific inputs to be used. Table II shows the characteristics of the test cases. Four combined MZ models have been implemented to

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

generate the inputs for each of these test cases (https://nube.uca.es/index.php/s/Z3lHZFAueE6FV4G) (Tc_2 represents when no option has been applied). Afterwards, `cat` has been run applying the options and inputs generated by the combined MZ models, obtaining the expected results. The constraints on the outputs have been checked manually. In addition, to find out the percentage of executed code, we have also used the source code coverage analysis and profiling tool, `gcov` (https://gcc.gnu.org/onlinedocs/gcc/Gcov.html), which generates accurate counts of the number of times each statement in a program is executed and annotates the source code to add instrumentation. Over 281 executed lines of source code, every test case has executed the cumulative percentage indicated in the Table II. We can observe that when these test cases are run consecutively, the tc_1 covers 38.79% of the lines executed; there is no difference in coverage between the tc_1 and tc_2, since no options are included in the cat run. From the tc_1 to the tc_3, line coverage increases by 4.92 points, by simply substituting option -E for option -b, reaching 43.71% of lines covered. Nevertheless, between the tc_3 and tc_4 case, despite the inclusion of two more options, the percentage of line coverage only increases 2.91 points over the previous one. Finally, it is interesting to note that between the tc_4 and tc_5 cases, the coverage increases up to 4.63 points, having the same number of options as the previous case, simply substituting option -n by option -v, reaching a cumulative percentage, total for all the test cases of 51.25.

IV. Related work

In [3], Sun et al. propose a constraint-based model-driven testing approach for web services. They extend the Web Services Description Language (WSDL) to include constraints related to web services behavior and make an empirical study with 3 real-life applications. Although it is a complete study, it is specific to web services, unlike our proposal, which aims to be more generic.

Vishal et al. [16] applied MBT using Spec Explorer to the Image Detection Subsystem responsible for generation, detection and translation of X-rays to images. They developed their own tool that interfaces Spec Explorer with a constraint solver to generate specific test data for the model. They used a small constraint solver called ZogSolve, limited and without recent updates compared to MiniZinc which contains a graphical interface and supports numerous constraint solvers and is updated frequently. Furthermore, Spec Explorer is proprietary software, unlike GraphWalker which is open source.

RESTest [2] is a framework for automated black-box testing of RESTful APIs (representational state transfer application programming interface). Among its main features, RESTest supports the specification and automated analysis of dependencies between parameters, allowing the use of constraint solvers for the automated generation of valid test cases. Similar to our proposal, it is based on tests for black box environments, where the code is not accessible. In this case, open source software tools are used, such as IDLReasoner. Given an the OpenAPI Specification (OAS) and a set of IDL dependencies (e.g., using IDL4OAS), the tool translates them into a CSP expressed in MiniZinc. Afterwards, they analyze the result of CSP, to find out whether an API call satisfies all dependencies between parameters. Therefore, CSP is used for the same purpose as in [16] to manage the dependencies between parameters. However, unlike the work presented here, MBT is used as a technique to obtain the system model, but visual representations such as those of GW are not used.
Overall, unlike other approaches, our proposal has the advantage of using two open source and frequently updated tools. In addition, the proposal aims to be more general, both in applications and in the domains in which it is applied.

Although a simple case study of the proposed technique has been presented for didactic purposes, we believe that it is possible to apply this technique to other more complex systems, as long as they can be represented in GraphWalker as a finite state transition machine and the input conditions can be represented as a set of constraints for a constraint solver system. For constraint implementation and execution, we have chosen MiniZinc, because of the advantages discussed in Section III, but it is possible to use other tools containing constraint solvers. Moreover, there may be other model representation software tools similar to GraphWalker that can be used to represent the model. It should be ensured that they are also capable of running such a model and that they are configurable in terms of path coverage.

V. Conclusion

It has been proposed to use the combined MBT and CP techniques to generate a test cases suite that meets the path coverage criteria. The process has been developed through a case study of a widely known utility, `cat`, using GraphWalker to represent the state model and execute it visually. GraphWalker has produced a set of paths providing sets of input constraints to be solved through CP using the MiniZinc tool to generate the test cases covering the paths. Finally, the results have been checked by running `cat` with the corresponding options and the generated inputs, while measuring

 ISBN: 978-1-61208-997-3

test coverage with `gcov`. Therefore, we conclude that the combination of both techniques and the presented tools is a promising approach to facilitate software testing, making the process more readable by representing and running the model in a visual way. As future work, we intend to test this process with other larger programs and in other contexts, and introduce more formalisation and automation on the constraints.

ACKNOWLEDGMENTS

This work was partially funded by the Spanish Ministry of Science and Innovation and the European Regional Development Fund (ERDF) (projects RTI2018-093608-BC33 and RED2018-102472-T).

REFERENCES

[1] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-Based Testing Approaches: A Systematic Review". In: Proceedings of ASE 2007. pp. 31–36. ACM, NY, USA, 2007. https://doi.org/10.1145/1353673.1353681

[2] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs". In: Service-Oriented Computing. pp. 459–475. Springer International Publishing, Cham, 2020

[3] C. Sun, M. Li, J. Jia, and J. Han, "Constraint-based model-driven testing of web services for behavior conformance". In: International Conference on Service-Oriented Computing. pp. 543–559. Springer, 2018

[4] D. MacKenzie et al., "GNU Coreutils". Free Software Foundation, Inc, 1994-2022

[5] D. MacKenzie et al., "GNU Coreutils 9.1". Free Software Foundation, Inc, 1994-2022, https://www.gnu.org/software/coreutils/manual/html_node/cat-invocation.html, [Retrieved: July 2022]

[6] Data61 research network, CSIRO, "MiniZinc". https://www.minizinc.org/index.html, 2015, [Retrieved: July 2022]

[7] F. Barber and M. Salido, "Introducción a la programación de restricciones". [In English: "Introduction to constraint programming"]. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial vol. 7, pp. 13–30, 01 2003

[8] F. Rossi, P. van Beek, and T. Walsh, "Chapter 4 Constraint programming". In: Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 181–211. Elsevier, 2008. https://doi.org/10.1016/S1574-6526(07)03004-0

[9] H. Kjellerstrand, "Comparison of CP systems — and counting". In: SweConsNet Workshop 2012, 2012, [Retrieved: July 2022]

[10] M. Bernardino, E. M. Rodrigues, A. F. Zorzo, and L. Marchezan, "Systematic mapping study on MBT: tools and models". IET Software vol. 11, pp. 141–155, 2017

[11] N. Olsson and K. Karl, "GraphWalker, an open-source model-based testing tool". https://graphwalker.github.io/, [Retrieved: July 2022]

[12] P. Ammann and J. Offutt, "Introduction to software testing". Cambridge University Press, New York, 2nd edn., 2016

[13] R. Amadini et al., "Minizinc with strings". In: Logic-Based Program Synthesis and Transformation, pp. 59–75. Lecture Notes in Computer Science, Springer, 2017. https://doi.org/10.1007/9783319631394_4

[14] R. Amadini, G. Gange, P.J. Stuckey, and G. Tack, "GStrings Gecode with (dashed) string variables". https://github.com/ramadini/gecode, 2019, [Retrieved: July 2022]

[15] S. Rosaria and H. Robinson, "Applying models in your testing process". Information and Software Technology vol. 42, pp. 815–824, 2000. https://doi.org/https://doi.org/10.1016/S0950-5849(00)00125-7

[16] V. Vishal, M. Kovacioglu, R. Kherazi, and M. R. Mousavi, "Integrating Model-Based and Constraint-Based Testing Using SpecExplorer". In: IEEE 23rd International Symposium on Software Reliability Engineering Workshops. pp. 219–224, 2012. https://doi.org/10.1109/ISSREW.2012.88

 ISBN: 978-1-61208-997-3