Model-Based Testing of Industrial Transformational Systems

Petur Olsen[1,⋆], Johan Foederer[2], and Jan Tretmans[3,4,⋆⋆]

[1] Department of Computer Science
Centre for Embedded Software Systems
Aalborg University
Aalborg, Denmark
petur@cs.aau.dk
[2] Test Automation
Océ-Technologies B.V.
Venlo, The Netherlands
johan.foederer@oce.com
[3] Model-Based System Development
Radboud University
Nijmegen, The Netherlands
tretmans@cs.ru.nl
[4] Embedded Systems Institute
Eindhoven, The Netherlands

Abstract. We present an approach for modeling and testing transformational systems in an industrial context. The systems are modeled as a set of boolean formulas. Each formula is called a clause and is an expression for an expected output value. To manage complexities of the models, we employ a modeling trick for handling dependencies, by using some output values from the system under test to verify other output values. To avoid circular dependencies, the clauses are arranged in a hierarchy, where each clause depends on the outputs of its children. This modeling trick enables us to model and test complex systems, using relatively simple models. Pairwise testing is used for test case generation. This manages the number of test cases for complex systems. The approach is developed based on a case study for testing printer controllers in professional printers at Océ. The model-based testing approach results in increased maintainability and gives better understanding of test cases and their produced output. Using pairwise testing resulted in measurable coverage, with a test set smaller than the manually created test set. To illustrate the applicability of the approach, we show how the approach can be used to model and test parts of a controller for ventilation in livestock stables.

⋆ Work performed while visiting Radboud University and Océ-Technologies B.V.

⋆⋆ This work has been supported by the EU FP7 under grant number ICT-214755: Quasimodo.

1 Introduction

Océ is a leading company in designing and producing professional printers. As the complexity of these printers grows, both due to features added and due to the requirement to support several input formats and backwards compatibility, the task of effectively testing the printer controller becomes very difficult. In this paper we present a model-based approach to improve the testing of the controller software of Océ printers.

We consider the part of the controller which processes input job descriptions and sends commands to the hardware. The system considered is in its abstract form a simple function. It takes a set of parameter values as input and computes a set of output parameter values. Input parameters are specific settings to a print job (number or pages, duplex/simplex, etc.), and the output is the description, in terms of output parameters, of the actually printed job. The dependencies between inputs and outputs are not trivial, and as the number of input parameters is over 100 and the number of output parameters is 45, the size of the system makes testing a difficult task.

The controller is modeled using a set of constraint clauses on the input parameter values, in the form of boolean formulas. Each clause relates a set of input values to an expected output value. The approach that we take in this paper is similar to that of QuickCheck [4] and Gast [9], both of which are automatic testing tools for functional programming languages, that generate random test cases. Both tools are less suited for use at Océ, since, being based on functional programming languages, they are cumbersome to integrate into existing test frameworks, whereas randomness makes structured generation of test cases and coverage determination more challenging. This led us to implement an internal prototype in Python to handle the testing.

Others have tried similar approaches to testing real world applications, such as Lozano et al. [11], who model a financial trading system with constraint systems. This leads us to believe that the approach is applicable to other types of systems as well. To further evaluate this approach we analyze how it can be used to model parts of the controller software for a ventilation system for livestock stables.

While the final goal is to detect faults in the SUT, this has not been the main focus in this project. Rather the focus has been to take steps toward creating a maintainable, large-scale model-based testing environment. It was not our aim to compare numbers of bugs found in model-based and manual testing.

This paper presents the problem of testing printer controller software. We present an approach to modeling the controller as a set of boolean formulas, including a modeling trick to enable us to make relatively simple models for the complex system. We present the testing process and how the desired coverage can be achieved. Additionally we present some discussions on using model-based testing in an industrial setting, and which benefits this approach has given Océ. Finally, to illustrate the applicability of our approach, we show how it can be adopted to model and test part of the controller software for a ventilation system for livestock stables.

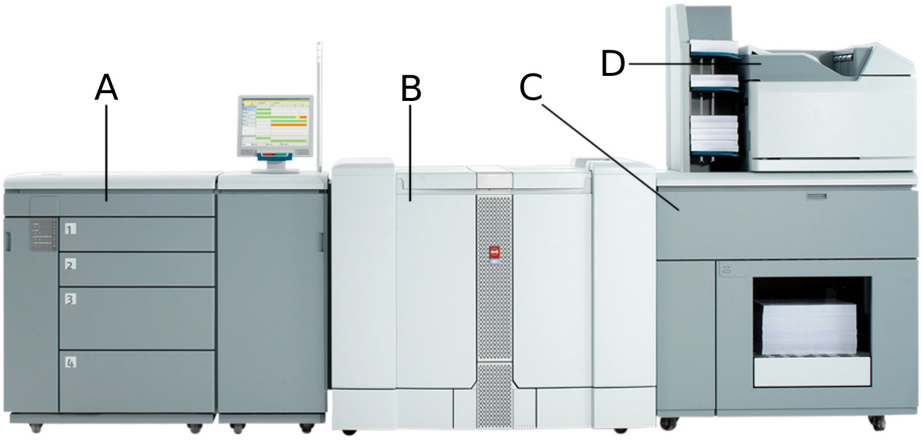

Fig. 1. VarioPrint 6250. A) Input module with four trays. B) Printer module. C) High capacity stacker output location. D) Finisher output location.

2 Problem Description

The problem is to test the controller of Océ printers. Océ produces professional printers, an example of which is shown in Figure 1.

- A is the input module with four input trays.
- B is the actual printer module.
- C is an output location called the High Capacity Stacker (HCS).
- D is an output location which supports stapling, called the Finisher.

This example is a small configuration of a printer. Several input modules can be attached and different output locations with different finishing options are supported.

The controller in these printers basically has two tasks: (*i*) handling the printing queues, and (*ii*) processing an input job description and sending the corresponding commands to the printing hardware. The part of the controller handling the printing queues can be seen as a reactive system, which continually monitors for job descriptions, sends them through the job processor to the printing hardware, and allows the user to perform actions on a user interface, for instance to cancel a job. This part can be modeled using some form of state machine. The part handling job processing, however, does not operate reactively. It accepts one job description at a time, and produces output for that job. Such a system can be seen in its abstract form as a simple, stateless function, accepting a set of input parameter values and returning a set of output parameter values. This is the part of the controller which this project focuses on, and which will be tested.

A *job description* consists of two parts: a document in a Printer Description Language (PDL) format and an optional *ticket* describing how to print the document. Several PDL and ticket formats are supported, each supporting different

features and using different formats for expressing features. Some features are supported in both the PDL and the ticket, requiring the job processor to handle contradictions. Example input parameters include output location, stapling, and punching.

As output the job processor presents a set of parameter values for each sheet to be printed. These values are sent to the printer hardware which prints the job as specified. Example output parameters also include output location, stapling and punching, however the relationship between the inputs and outputs is not as simple as it might seem.

First, there are the contradictions. Stapling, for instance, can be specified both in the PDL and in the ticket, in which case the ticket will overrule the PDL. While stapling is enabled, an output location can be selected which does not support stapling, in which case the output location is overruled to one which does support stapling. However, there might not be any output locations attached to the printer which support stapling, in which case the stapling is disabled and the output location is as specified. Just for these two simple parameters we already have a lot of cases.

In addition to contradictions, there are different formats for specifying values. Specifying stapling in a ticket, for instance, has ten possible values: None, Top, TopLeft, Left, ..., and Saddle. The stapling output from the job processor only has five: None, Portrait, Landscape, Booklet, and Saddle (the orientation of the paper and the output location determine where the specific staple is located). Similarly other PDLs and ticket formats might have different formats for specifying stapling. Translating between these formats is not trivial.

On top of all these are the settings of the job processor. For instance the limit for the number of pages which the printer can staple can vary. Several other settings are available in the job processor.

It is clear that the job processor needs to handle all peculiarities in the input, as well as any settings of the printer. The job processor needs to support all configurations of printers, and needs to be able to print any job on any configuration, albeit possibly with some functionalities disabled. The configuration and settings of the printer can be seen as inputs to the job processor. Adding these to the PDL and ticket, and looking at all available parameters, the number of parameters for the job processor comes to well over 100. These facts make the job processor a very complex system, and testing such a system is not trivial.

2.1 Testing at Océ

The current testing process at Océ involves running a job description on a simulation of the hardware. When running the job processor on the simulated hardware, the output is presented in the form of a so-called APV file. This APV file contains all parameters for each printed sheet. Currently there are 45 parameters in the APV file.

The resulting APV file is analyzed manually. If deemed correct it is saved as a reference for future test runs. In subsequent automatic test runs the output can be checked against the reference and the result of the test can be determined.

There are several issues with this testing process. The first is maintainability when updating the job processor to support more parameters. This requires all test cases to be updated to support this parameter. Secondly, changing some requirements, which lead to failing test cases, requires the new APV file to be manually analyzed again. This manual analysis is very time consuming and error prone. It occurs that errors survive through the development process because of faulty analysis of the APV file.

The execution time of the test cases is also becoming an issue. Nightly runs, executing the complete set of test cases, have to finish in the morning, to present the results to the engineers. At the current number of test cases some of these runs do not complete in time. Due to expansions and new developments the number of test cases is expected to double, in the near future. This poses big requirements to the computer farm running the nightly tests, and requires expensive expansions. Therefore it is desirable to reduce the number of test cases, but the quality of the complete test set must not suffer.

Currently a *test case* is a Python script which sets up the printer, generates one or more job descriptions, and sends them to the controller in a specified order. These test cases are designed by test engineers who know the system intimately. The test cases are designed to find likely errors, and are very specifically designed, such that a failing test case gives some hint to where the error occurs. For instance, a test case might focus on stapling by generating several job descriptions with different stapling positions. If this test case fails the error is most likely in the stapling module. This gives very specific test cases and to get good coverage it requires a lot of test cases. This leads to the desire to have structurally generated test cases which have some measure of coverage, while minimizing the number of test cases.

In the current framework there is no uniform way of defining test cases. This stems from the different formats for PDLs and tickets, and the fact that current test cases are directly coded at a low level in Python. Since these are often generated in batches in for-loops, it can be difficult for other testers and developers to understand exactly what a test case does. This leads to problems with understanding test cases, and once a test case fails, it can also be troublesome to understand precisely what the parameters of the failing job were.

This presents four areas where Océ wants to improve their testing process:

- maintainability,
- execution time,
- coverage, and
- understanding of test cases.

We will improve on these aspects, by implementing a model-based approach to testing.

3 Modeling the Controller

The job processor is in its abstract form a simple, stateless function. It takes a number of input parameter values and computes a number of output parameter

values. We modeled the job processor as a collection of Boolean formulas, where each formula specifies the value for one output parameter through an implication with on the left-hand side a conjunction of input parameter constraints, and on the right-hand side an output parameter constraint:

$$i_1 = v_1 \wedge i_2 = v_2 \wedge \cdots \wedge i_n = v_n \Rightarrow u_p = v_p \tag{1}$$

This formula expresses that the expected output of parameter u_p is value v_p if input parameters i_j have values v_j for $1 \leq j \leq n$, respectively. Each of these formulas is called a *clause* in the model.

For integer parameters we also allow comparisons like $i_j \leq v_j$, e.g., to refer to equivalence classes of input parameters. As an example, a simplified clause of the staple position could look like this:

$$\begin{aligned}(Staple = TopLeft \wedge SheetCount \leq 100) \Rightarrow \\ StaplePos = Portrait\end{aligned} \tag{2}$$

This specifies that the output parameter for Staple Position *StaplePos* has value *Portrait* if the input parameter *Staple* is *TopLeft* and there are less than 101 sheets.

For integer output parameters we allow the expected output to be calculated by a function on the input parameters. For instance if the *Plexity* is set to *Duplex* (printing on both sides of the paper), *SheetCount* becomes half of the number of printed pages: $SheetCount = \lceil Pages/2 \rceil$.

The actual job processor model has many more parameters and also more possible values for the parameters, resulting in many more clauses. A complete job processor model consisting of such a collection of clauses must first be verified for completeness and consistency, i.e., checked whether the collection indeed specifies a function from input parameters to output parameters, but such a verification is orthogonal to testing.

Satisfaction of the model has been used as oracle for our testing process. This means that the model is instantiated with actual output parameter values of the job processor implementation, together with the corresponding input parameter values. (Section 4 will deal with choosing input values). If all clauses hold the test passes; if a clause does not hold then the test fails and the output parameter specified in the false clause is wrong.

3.1 Dependencies

As seen above the model for staple position depends on two input values, and the complete model is even bigger. If we have a look at a simplified clause of the output location *OutputLoc*:

$$\begin{aligned}(TicketOutputLoc = HCS \wedge Staple = TopLeft \wedge \\ SheetCount \leq 100) \Rightarrow \\ OutputLoc = Finisher\end{aligned} \tag{3}$$

then we can see that it depends on the input parameters *Staple* and *SheetCount*. This is because the output location should only be overridden if a staple was requested, and the printer is actually able to staple. This causes a chain of dependencies, where the output location clause must contain all – transitive – dependencies in its clause. These chains clutter the clauses and make modeling cumbersome, since there are a lot of these type of dependencies. To simplify the clauses we can observe that a part of (3) can be substituted with (2). Substituting $(Staple = TopLeft \wedge SheetCount \leq 100)$ for $StaplePos = Portrait$ we get the simpler clause:

$$(TicketOutputLoc = HCS \wedge StaplePos = Portrait) \Rightarrow$$
$$OutputLoc = Finisher$$

We can see that the output location actually depends on the output parameter *StaplePos*. Formally, we allow $(i_j = v_j)$ from Equation 1 to refer to input- and output parameters.

One potential problem arises with this approach. If there are circular dependencies, we can not trust the results. To avoid circular dependencies we arrange all clauses in a hierarchy, where the leaves have no dependencies and parents depend on the parameters of their children. As long as this hierarchy is kept, it is safe to use some output parameters to verify other output parameter values. This approach simplifies the clauses significantly, and enables us to model these complex systems.

4 Testing

Testing a job processor implementation involves three steps:

- Selecting input values,
- executing the SUT with the input values, and
- verifying output from the SUT using the model.

Executing the SUT is done using the existing framework for automatic testing at Océ. Verifying the outputs is done using the model, as explained in the previous section. To select input values we look at the complete set of input parameters supported by the model, and the domains of these parameters. Instantiating each parameter constitutes a single test case. This can be done randomly, to generate a set of test cases, or it can be done structurally, based on some coverage criterion.

It has been shown in several projects [5, 6, 8, 2, 12] that most software errors occur at the interaction of a few factors, i.e. most errors are triggered by particular values for only a few input parameters, whereas the error is independent from the values of the other input parameter. Some projects report up to 70% of bugs found with two or fewer factors and 90% with three or fewer [5, 10], others show up to 97% of bugs found with only two factors [12]. This, combined with the fact that the number of test cases needs to be minimized, leads to combinatorial testing techniques, such as pairwise (or more generally n-wise) testing. The

number of test cases in n-wise testing for fixed n grows logarithmically compared to exponential growth for testing all possible combinations.

The coverage of output parameters is not guaranteed by using the combinatorial testing technique. Pairwise testing does, however, tend to cover most of the unintended uses of the system, and many of the paths which lead to special cases, whereas manually generated test cases tend to focus on normal operation of the system. Once a test suite has been created, the output coverage can be analyzed, and the test suite can be updated to add any required coverage.

Test cases are generated based on a set of input parameters and their domains. A test case is an assignment of each input parameter to a single value from its domain. A *test specification* is a set of relations between input parameter name and the discrete domain of that parameter:

$$\begin{gathered}(P^1, \{v_1^1, v_2^1, \ldots, v_{n_1}^1\}) \\ (P^2, \{v_1^2, v_2^2, \ldots, v_{n_2}^2\}) \\ \vdots \\ (P^m, \{v_1^m, v_2^m, \ldots, v_{n_m}^m\}).\end{gathered}$$

Given such a test specification, algorithms can generate a set of test cases which cover all pairs of values. That is, for every v_k^i and v_l^j where $i \neq j$, $P^i = v_k^i$ and $P^j = v_l^j$ for at least one test case.

For instance if we have three input parameters: $TicketOutputLoc$, $Staple$, and $SheetCount$, the test specification could be:

$$\begin{gathered}(TicketOutputLoc, \{Finisher, HCS\}) \\ (Staple, \{None, Top, Left\}) \\ (SheetCount, \{\leq 100, 101\})\end{gathered}$$

Each pairwise combination of values, e.g. $TicketOutputLoc = Finisher$ and $Staple = Left$, must be present in at least one test case. In this case six test cases are needed. Generating complete coverage would require 12 test cases.

4.1 Diagnosis

The automatic test case generation based on a job specification can be used as a tool in diagnosis. Reducing the domain of one or more parameter in the job specification, can provide information about the fault. As an example consider the job specification above, and consider a fault has been found with one of the generated test cases. Reducing the domain of $Staple$ to $\{None\}$, and re-running test case generation and test case execution, can help locate the fault in the SUT. If, for instance, none of these new test cases finds the fault, it tells us that the fault only occurs when a staple is requested. This tells us that the error is either in the staple module, or occurs at the interaction between the staple module and some other part of the controller. Using this approach the test engineers can easily generate new sets of test cases to help locate bugs in the SUT.

5 Implementing the Test Tool

Other projects to improve the testing process, have previously been carried out at Océ. Experiences from these projects have shown that integrating tooling with existing frameworks can be difficult and cumbersome, and introducing new tools and frameworks requires some learning effort from the engineers. This led us to implement a prototype tool for this project by hand. The existing framework for automated testing has a lot of tooling and libraries for testing the job processor, therefore it was decided to integrate the prototype into this framework. The existing framework is written in Python, so Python is the language of choice.

The clauses are implemented as if-then-else statements. To give a logical grouping of clauses, all clauses pertaining to the same output parameter are grouped into the same Python class. This way a class is said to *check* an output parameter by implementing all clauses for that parameter. To ease implementation several output parameters can be checked by the same class.

The entire set of actual input values and output values is passed to each class. This enables the class to access any values needed in the clauses to verify their respective output value. The classes access the values they need and then go through a series of if-then-else statements that implement the clauses in the formal approach. Once an expected value is found, it is checked against the actual output value, and an appropriate response is added to the return set.

The return set contains *OK* or *Error* responses from each class, and is returned back to the test system. Here it can be analyzed and all errors found by the model, can be returned to the engineer.

Once an output value has been verified, it is removed from the set of output values, which is passed to subsequent classes. This feature has two effects. First, it enables us to check whether all output values have been verified, by examining if the set is empty when all classes have been invoked. Second, it enforces the hierarchy required to detect circular dependencies as explained in Section 3.1. This is enforced since a circular dependency would result in a missing value in the latter class in the circle. This also means that classes with dependencies need to be invoked first, and classes with no dependencies are invoked last.

5.1 Test case generation

N-wise testing has currently been implemented using the tool `jenny`[5]. `jenny` is an executable which accepts as input, the domain size of each input parameter and generates a set witnessing n-wise coverage of all input parameter values. Currently `jenny` is always executed for pairwise coverage. This could be extended, if the coverage requirements increase. A wrapper has been written around `jenny`. A test specification is passed to the wrapper, which generates a `jenny` query for the domain sizes of the parameters. `jenny` returns a set of test cases which are translated back into the specific values in the test specification. As an example

[5] http://burtleburtle.net/bob/math/jenny.html

consider the test specification:

$$(TicketOutputLoc, \{Finisher, HCS\})$$
$$(Staple, \{None, Top, Left\})$$
$$(SheetCount, \{\leq 100, 101\})$$

The wrapper generates a query for `jenny` with three parameters with domain size two, three, and two respectively. `jenny` generates six test cases:

```
1a 2a 3b
1b 2c 3a
1b 2b 3b
1a 2b 3a
1b 2a 3a
1a 2c 3b
```

Each line represents a test case. The number represents the input parameter. The letter represents the value of the parameter. The test case `1b 2c 3a` is translated into $(TicketOutputLoc = HCS, Staple = Left, SheetCount = \leq 100)$.

This way test cases are generated based on the test specification as created by the test engineer. In the case of diagnosis the engineer can reduce the domain sizes in the test specification, and rerun the wrapper to get a new set of test cases.

5.2 Run Time

The time required to execute a single test case is highly dependent on the number of pages printed in that test case; depending on the hardware running the simulator, printing a single page can take up to half a second. This fact means that test cases with a lower number of pages are preferred. In the model for testing the stapling module the equivalence classes for the number of sheets are:

- 1 (too few sheets),
- 2 (too few sheets for duplex),
- $3 - 100$,
- $>= 101$ (too many sheets).

Including all these in the pairwise test case generation would include many jobs with 100 pages or more. However, it can be observed that if the staple limit works for a single test case, it most likely works for all test cases. This observation leads us to implement the possibility to add manually generated jobs to the test set. By adding two jobs with 100 and 101 sheets printed, we reduce the equivalence classes for staple limit to three and by choosing a low value for the equivalence class $3 - 100$ the number of sheets printed can be kept low in all other jobs. This dramatically reduces the time required to execute the test suite, while still keeping acceptable coverage.

5.3 Invalid Test Cases

Since the PDL and ticket formats support different features, it is possible to generate invalid test cases. For instance, one ticket format supports accounting options, so we need to have input parameters for accounting. However, if we generate a test case with accounting activated, while using a ticket format which does not support accounting, the test tool is unable to generate the job for the controller, since activating accounting in this ticket format is not possible. This is an invalid test case, since it can not be executed on the SUT.

These invalid combinations need to be removed from the pairwise coverage. This is because the pairs covered by an invalid test case are not executed on the system. Excluding simple combinations of parameter values is supported in the current version. More complex exclusions, such as employed by e.g. AETG [5], ATGT [3], or Godzilla [7] might be required in the future.

6 Status and Discussion

It was chosen to focus on modeling the stapling capabilities of the printers, as an initial step. The currently implemented models support four parameters: PDL format, page count, output location, and staple location. The four parameters have domain sizes two, three, three, and eight respectively. This set is pairwise covered in 27 test cases. To cover the upper page limits for stapling four test cases are manually added, bringing the total number of test cases generated from this project to 31.

It is difficult to say precisely how this coverage compares to the current set of test cases, as a lot of test cases touch stapling, while testing other areas of the controller as well. Of the manually generated test cases a total of 170 test cases use stapling. Looking at all the parameters and parameter values used in these 170 test cases, we observe 11 parameters with domains ranging from two to nine. Getting pairwise coverage of all these input parameters can be achieved with only 86 test cases. These parameters are not currently supported by the model, so the test cases can not be executed at this time. This shows us that once the models are extended, the number of test cases can be reduced. Determining if the fewer number of test cases will locate as many or more errors will require further work.

The choice of implementing a prototype in Python proved very useful. This also supports previous experience in similar projects. Integrating with the existing Océ framework was very easy. In the current version of the tool, implementing the models as Python classes is cumbersome and requires some copy-paste. With further development and refactoring, we expect to build a viable environment for developing models.

Using a model-based approach gives the engineers far better understanding of the test cases and their outcome. The new framework gives better overview of which parameter values are selected in each test case, and the models can be used to give a hint as to where an error is located in the code. The controllable

test case generation can also be used for analysis, to find the precise interactions between parameters which cause the error.

The issues with maintenance are also improved, as making changes to the requirements only requires updating the model, then all test cases should work. Updating and implementing the models is not trivial, and errors in the model could cause false positives and false negatives. However, since the same models are used by all test cases, it is more likely that errors in the model will be found.

Currently only pairwise coverage is supported. As the usage of this approach grows in Océ, it will be seen how effective this coverage is in locating faults. It might be the case that the coverage needs to be supplemented by manually generated test cases, or replaced by a different coverage measure. Currently no analysis of output coverage is done. This requires engineers to manually examine the test cases, and possibly supplement with additional cases.

Even though the focus in this project was not to detect faults in the SUT, one unknown fault has been located. The unknown fault has not been located by the old set of manual test cases. This also indicates that the coverage criteria might be good.

Based on the advantages of the model-based approach, Océ has decided to continue development of this prototype, and extend models to continue improving the testing process.

7 Modeling a Livestock Stable Controller

Since our approach shows promising results for modeling and testing printer controllers, and the literature shows similar approaches used in other types of systems, we wanted to examine if our approach could be applied in other companies. We have initiated contact with a company designing and producing ventilation systems for livestock stables, to examine how the approach applies there. The controller for these systems is split into several components, each of which monitors some input sensors and controls some output actuators. The inputs are continuous measurements of e.g. temperature. The outputs are either *on/off* values or a percentage value describing how much power should be given to, for instance, a ventilator. Calculation of the two types of output are done in a standard way.

For *on/off*-type of controllers there are two important parameters: t, and T_δ. The value of t describes when the output should be activated. To avoid oscillation between *on* and *off*, T_δ describes how far below t the input has to fall before deactivating the output. It can be observed that between $t - T_\delta$ and t this controller shows nondeterministic behavior. This nondeterminism can be seen as an internal state in the controller, storing its previous output value. For inputs between t and $t - T_\delta$ the controller outputs the same value as previously. For inputs below $t - T_\delta$ the output is always *off*, and above t the output is always *on*. Figure 2 illustrates the possible values.

For percentage-type of controllers the output value is a linear function of the input. The function is described by two parameters: p and P_δ. The value of p

describes when the output must start increasing. P_δ describes how far above p the output must reach 100%. Below p the output is always 0%, above $p + P_\delta$ the output is always 100%. In between the output grows linearly from 0% to 100%. Figure 3 illustrates the function.

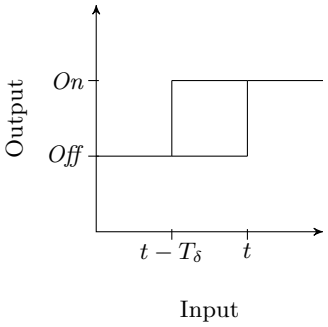

Fig. 2. Graph for *on/off* values

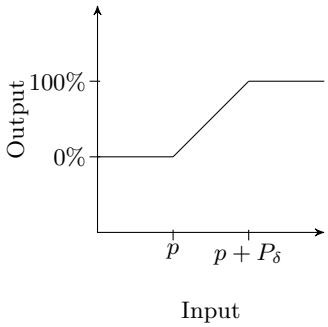

Fig. 3. Graph for percentage values

Components can have several inputs based on the same patterns, to form more complex components. For instance, a component can have a temperature reading and a humidity reading as input, and can activate a ventilation fan as output. The ventilation fan should be activated when the temperature reaches above some value while the humidity is below some value. Both inputs will have T_δ values for when the fan should be deactivated again.

Currently test cases for the system are generated manually. A test engineer examines the parameters of the component and generates a set of inputs generating an acceptable coverage, and generates corresponding expected outputs. Subsequently the test cases are executed automatically and the expected outputs are compared to the actual outputs.

This type of system can be modeled within our framework, with a single modification. We need to handle the nondeterministic behavior. This can be done by representing the model as a hybrid automaton[1] by handling the state as an input. We make a fresh input parameter to represent the state of the model. The domain of this parameter is the state space of the model. This way the clauses in the model can depend on the state the system is in and act accordingly. However, in this simple setting this seems like too complex a solution. We only need a single state variable; a Boolean. We only need to know the value of this variable one time step backward. This can be easily be handled in the current setting of transformational systems. The problem with adding this functionality is that pairwise test case generation will not work since test cases become traces, where each step depends on the previous one. Some work needs to be done to find a good way to generate test cases for this type of system. The test case generation needs to generate valid test cases and provide coverage of the data in input parameters as well as coverage of the state space of the model.

With this modification a large set of manually generated test cases, can be automatically generated from simple models. Future test cases can easily be created by instantiating the model with the required values.

This shows that the approach is indeed applicable for different types of systems, even though it was developed for testing printer controllers. The diversity of the SUTs shows that there are potentially several industrial areas where similar approaches could be applied to improve the testing process.

8 Conclusion

This paper has presented the initial steps towards a model-based testing framework for testing printer controllers at Océ. The approach has proved promising in improving the testing process and the quality of test cases. Advantages of the approach include: improved maintenance, reduced number of test cases, measurable coverage, and better understanding of the test results. While the approach seems promising at improving the testing process, further work is needed to state this for certain. While test cases can now be generated based on coverage requirements, it is unclear if the generated test set will locate as many errors as the old test cases. Also development of the model requires some significant effort, but it is expected to prove valuable in the long run. Based on the outcome of this project, Océ has decided to continue with the model-based approach. Finally, we illustrated that the approach is useful for modeling and testing controller software for ventilation systems in livestock stables. This diverse usefulness of our approach indicates that several other industrial areas could benefit from similar approaches.

The connection to the current way of working at Océ, and usability of the methods by Océ in their current environment, were among the main requirements and starting points of this project. This did not always lead to the most sophisticated, or theoretically new solutions, and sometimes even to ad-hoc solutions, e.g. to establish the connection to the existing Python tooling. Future work will include the use of more sophisticated approaches, such as using SAT-solvers or SMT tooling to solve, check, and manipulate Boolean formulas.

References

[1] Rajeev Alur, Costas Courcoubetis, Thomas Henzinger, and Pei Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57318-6_30.

[2] Kevin Burr and William Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513. West, 1998.

[3] Andrea Calvagna and Angelo Gargantini. A logic-based approach to combinatorial testing with constraints. In Bernhard Beckert and Reiner Hhnle, editors, *Tests and*

Proofs, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-79124-9_6.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[5] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23:437–444, July 1997.

[6] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *Software, IEEE*, 13(5):83 –88, September 1996.

[7] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 17(9):900–910, 1991.

[8] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing: experience report. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 205–215, New York, NY, USA, 1997. ACM.

[9] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: generic automated software testing. In *Proceedings of the 14th international conference on Implementation of functional languages*, IFL'02, pages 84–100, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] Richard Kuhn and Michael Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceeding of the 27th NASA/IEEE Software Engineering Workshop*. IEEE, 2002.

[11] Roberto Castañeda Lozano, Christian Schulte, and Lars Wahlberg. Testing continuous double auctions with a constraint-based oracle. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 613–627, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] Dolores R. Wallace and D. Richard Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. In *ACS/ IEEE International Conference on Computer Systems and Applications*, pages 301–311, 2001.