Electronic Communications of the EASST
Volume 70 (2014)

Proceedings of the
14th International Workshop on
Automated Verification of Critical Systems (AVoCS 2014)

A Constraint-Solving Approach for Achieving Minimal-Reset Transition Coverage of Smartcard Behaviour

Renaud De Landtsheer, Christophe Ponsard, Nicolas Devos

15 pages

Guest Editors: Marieke Huisman, Jaco van de Pol
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Constraint-Solving Approach for Achieving Minimal-Reset Transition Coverage of Smartcard Behaviour

Renaud De Landtsheer, Christophe Ponsard, Nicolas Devos

{rdl,cp,nd}@cetic.be, CETIC Research Center, Belgium

Abstract: Smartcards are security critical devices requiring a high assurance verification approach. Although formal techniques can be used at design or even at development stages, such systems have to undergo a traditional hardware-in-the-loop testing phase. This phase is subject to two key requirements: achieving exhaustive transition coverage of the behavior of the system under test, and minimizing the testing time. In this context, testing time is highly bound to a specific hardware reset operation. Model-based testing is the adequate approach given the availability of a precise model of the system behavior and its ability to produce high quality coverage while optimizing some cost criterion. This paper presents an original algorithm addressing this problem by reformulating it as an integer programming problem to make a graph Eulerian. The associated cost criterion captures both the number of resets and the total length of the test suite, as an auxiliary objective. The algorithm ensures transition coverage. An implementation of the algorithm was developed, benchmarked, and integrated into an industrial smartcard testing framework. A validation case study from this domain is also presented. The approach can of course be applied to any other domains with similar reset-related testing constraints.

Keywords: Model-based testing, Constraint Solving for Verification, Hardware in the loop

1 Introduction

Testing remains the major industrial approach for software validation and verification. However, with the development of model-based engineering and the use of Domain Specific Languages, testing is not any more disconnected from modeling activities. In this context, Model-Based Testing (MBT) has appeared as way to automate test generation from models [UL06, DSVT07]. Based on a model of the system under test, it provides the advantage of automated generation, ensuring complex coverage criteria, evolution of the test suites as the specification evolves. However, there are also a number of shortcomings: the effort required to build a model is also quite high and requires a deep change in the design process. Moreover, additional effort is also required to manage a translation from an abstract test suite to concrete test cases. There are also still language and tool limitations to cope with, especially when managing large models and generating tests from them [UL06].

In this paper, we take a more focused approach which does not try to capture the behavior of the whole system but only specific behavioral aspects expressed in the widely industrially adopted formalism of state machines [Har87, Fow03].

Several coverage criteria for state machines have been proposed and are now considered as standard: transition coverage, state coverage, pair of transition coverage, pair of state coverage, path coverage, etc. They provide relevant trade-off points between the level of assurance provided by a successful test campaign, and the cost and time devoted to the creation and execution of a test suite. The overall length of the test suite and the individual cost of some steps of testing are the two factors influencing the time necessary for executing a test suite.

In this paper, we focus on transition coverage which is quite popular. Our contribution is to take into account a frequent problem of the cost of managing transition under tests by driving the system into a well-defined state, for example by applying some ”reset” procedure. The cost of this system reset can be quite prohibitive and should be minimized. We present an algorithmic approach to automatically generate test suites that minimize the number of *resets* that must be performed to execute a test suite. We consider that a reset must be performed between each test case. Our algorithm also minimizes the overall length of the test cases as an auxiliary cost criterion. So far, our algorithm exclusively supports the transition coverage criterion.

This problem was encountered in the context of smartcard testing. In this context, resets are expensive because they are performed either by flashing the program memory, or by switching to a new and fresh device in case the hardware flashing is disabled. Hardware flashing might be disabled for security reasons, as it might give access to the innards of the card. Those operations are an order of magnitude more time consuming than performing the software actions listed by the state machine implemented in the smartcard. This problem can be encountered in other domains, for example in business workflow management systems where going to a known state can require complex database operations.

In this paper we will consider a running example from the smartcard domain: a fragment of the public Eurocard-Visa-Mastercard (EMV) specification [EMV08]. It is described in a relatively small state machine presented in **Figure 1** driving the card lifecycle between initialization to more operational states. This system runs on smart card commonly encountered for debit and credit cards.

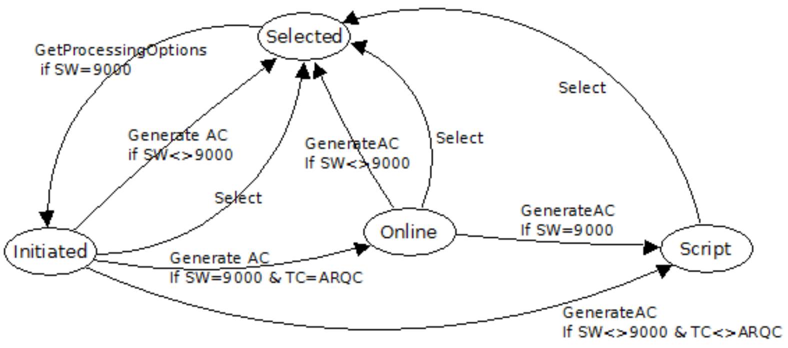


Figure 1: The EMV state machine that governs smart cards

Our proposed approach can be summarized by the following reasoning:

1. We consider that **a test suite is a path in the state machine**. Each test case is a fragment of this path, and test cases are separated by hardware resets. The hardware reset is modeled

as an extra transition in the state machine, each state but the initial one has a reset transition that leads to the initial state. Hardware resets are thus added to the purely software state machine in the beginning of our algorithm.

2. **A transition coverage criterion is very close to the definition of *Eulerian path*.** An Eulerian path in a graph is a path that takes each transition exactly once. Finding an Eulerian path in a graph has complexity $O(\#\text{Edges})$. The search for the test suite is therefore nothing else but searching for an Eulerian path, assuming such a path exists. The length is of course optimal because it is equal equal to the number of edges.

3. **There can only be an Eulerian if the graph is Eulerian itself and this can be checked by simple conditions, mainly involving the degree of the nodes**. There are two possibilities to cope with this, either to develop a specific algorithm derived from the one that finds an Eulerian path or to transform the graph into an Eulerian one. We have opted for the second approach because it seemed computationally more efficient, much more elegant, and less expensive to implement.

4. Transforming an ordinary graph into an Eulerian graph can be achieved by adding some edges to enforce the conditions that make a graph Eulerian. This operation allows for degrees of liberty. However, we can only add reset edges (from any state to the initial state) or duplicate existing transitions. We cannot add any other new transitions.

5. All added transitions do not have the same impact, especially **adding reset transitions is very costly and one can prefer to add many ordinary transitions instead, this can be formulated as an optimization problem.** The optimization problem is expressed as finding the set of transitions making a graph Eulerian while minimizing the cost criteria which is expressed as the weighted sum of the added transitions, with reset transitions having a much larger costs than normal ones. **The optimal test suite is then simply the Eulerian path cut at the reset transitions**.

6. On the technical side, **this problem can be easily expressed as an Integer Programming (IP) problem, which can be solved efficiently using Integer Programming techniques (MIP solver)**.

The remainder of this paper is structured as follows. Section 1 formally defines the problem that we are solving in this paper and the notation used throughout the paper. Section 2 introduces the specific technological background of our approach, notably Eulerian graphs and integer programming. Section 3 presents our algorithm in full details. Section 4 illustrates the result of running our algorithm on our EMV case study as well as on a full set of benchmarks showing how it scales. Section 5 presents some related work. Finally, section 6 summarizes the paper and gives open issues and possible extensions of this approach.

2 Problem Statement

The problem solved in this paper is formally defined as follows:
Given a state machine,

- described by its states S and its transitions T, $T \subseteq S^2$
- with initial state *init* , init $\in S$
- ensuring reachability (i.e. each state is reachable from *init*)

Find a set of paths, each path starting at *init*, such that:

- each transition of the state machine is taken at least once by one of the paths
- the number of paths is minimal
- the overall length of these paths is minimal

The two objective functions are strictly prioritized: the number of resets is minimized first, and then, keeping the same number of resets, the overall length of the test paths is minimized.

3 Technical Background

This section presents our approach. It first recalls necessary background on graph theory, Eulerian paths, and integer programming. It then presents some inspiring problems from graph theory and explains the intuition behind our algorithm. Throughout the paper, the concepts of *state machine* and *graph* are undistinguished, and denoted by any of these two terms.

3.1 Graph Theory and Eulerian Path

An *Eulerian path* in a graph is a path that takes each transition exactly once. An Eulerian cycle is an Eulerian path such that the end node of the path is the same as the start node of the path. An *Eulerian graph* is a graph with an Eulerian path. There is a classical theorem that links the Eulerian property to simple properties of a graph. It can be found in lecture books [Tru93]. We only recall the property itself and not its proof. A directed graph is Eulerian if:

- it is strongly connected: for each pair of node, there is a path leading from one to the another
- for each node, its in-degree is equal to its out-degree

As exception to the rule above: a pair of nodes called *init* and *end* might exist:

- the out-degree of *init* is equal to the in-degree of *init* plus one
- the in-degree of *end* is equal to the out-degree of *end* plus one
- there might not be a path from *end* to *init*

Hierolzer's algorithm finds an Eulerian path in $O(\#Edges)$ in an Eulerian graph [Hie73]. The principle of the algorithm is to build a path starting from the *init* node such that the path takes edges at most once. The path does not necessarily cover all edges of the graph. The algorithm then iteratively completes the path with edges that were not taken. It proceeds by walking on the path until reaching a node that has edges leaving the node and that are not in the path yet. The algorithm explores a path starting from this new edge, and only takes edges that were not taken so far. This additional path is necessarily a cycle that comes back to the node where it started. This secondary path is then added to the initial path. The walk then proceeds on the completed path by first following the added section, and iteratively enriches the path with other missing edges. When the walk is finished, the path has been enriched with all the edges of the graph, it is therefore Eulerian.

3.2 Integer Programming

Integer programming is about solving problems of finding x such that:

- $x \cdot c$ is minimal
- $A \cdot x \leq b$
- all values of x are integers

Where x is a vector of unknowns, b and c are vectors of constants, A is a matrix of constants, $x \cdot c$ is the Cartesian product of x and c, $A \cdot x$ is the matrix product of A and x, the inequality $\leq$ holds for each row of the two vectors.

Efficient solvers are available for such problems [GLP, SCI, LPS, IBM]. Notice that the above IP problem only includes an inequality but can easily be extended to handle equality as well: equality $p = q$ can be encoded by the conjunction of two constraints $p \leq q$ and $-p \leq -q$.

3.3 Inspiring Problems

Related problems are presented in [EJ73, Thi03]. Among them, the approach is notably inspired from the *Chinese postman problem*. The Chinese postman must take each street of the city at least once to deliver the mail, and wants to minimize his overall walking distance. This problem is efficiently solved by an IP-based approach, to duplicate the necessary edges in order to make the graph Eulerian. Out of the Eulerian graph, an Eulerian path can be computed efficiently.

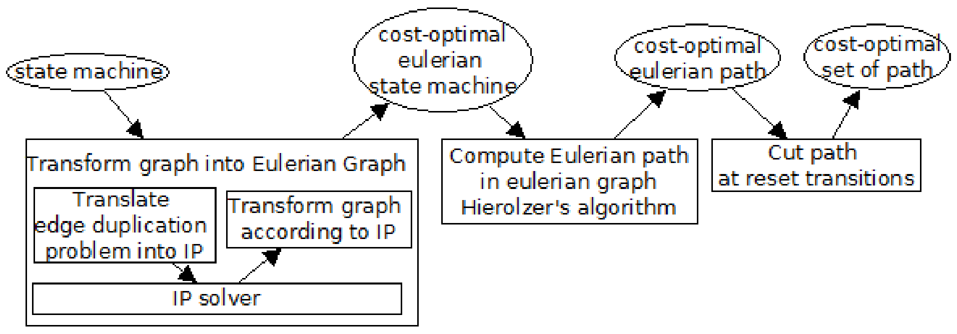
Two different classes of edges appear in a related problem called the *rural postman problem*. This problem is similar to the Chinese postman problem except that some streets are actually rural pathways where no mail must be delivered. They can be used as shortcuts and might not be taken in the final path.

In our case, the graph is directed, and there are two classes of edges: reset transitions and other transitions. Reset transitions must be minimized first, and might not be taken in the final path. Other edges must also be minimized, as a secondary objective.

4 Algorithm

The global view of the algorithm is depicted in **Figure 2**.

- The first step transforms the initial state machine into an Eulerian state machine by inserting reset transitions representing hardware resets, and duplicating some transitions of



state machine
cost-optimal eulerian state machine
cost-optimal eulerian path
cost-optimal set of path
Transform graph into Eulerian Graph
Translate edge duplication problem into IP
Transform graph according to IP
IP solver
Compute Eulerian path in eulerian graph Hierolzer's algorithm
Cut path at reset transitions

Figure 2: Architecture of our test generation approach

the state machine. The state machine resulting from the transform is designated as the *Eulerian machine*, with transitions T_e and states S_e.

- The second step finds an Eulerian path in the Eulerian state machine by applying Hierolzer's algorithm. It delivers a path that takes each transition at least once, and includes both transitions from the initial state machine and reset transitions. The algorithm is called so that the path starts at the *init* node.

- The third step cuts the Eulerian path at reset transitions. Each fragment is a path that starts at *init* and ends anywhere in the state machine.

We will focus here only on the first step as the two others are trivial. This first step enriches the initial state machine with *reset* transitions. The set of reset transitions added to the initial state machine is represented by R.

In order to express the duplication of the edges present in the initial state machine, each edge e of the initial state machine gets an associated unknown x_e that represents the number of time it will be present in the Eulerian state machine. Similarly, reset transitions also get an associated unknown x_i that will be resolved.

The main constraint that we want to enforce on the x variable is the one of degree equality: for each node, its in-degree should be equal to its out-degree. This constraint, taken from the definition of the Eulerian graph in **Section** 3.1 has an acceptable exception regarding the degree of nodes: a pair of nodes called the *init* and *end* node, respectively must enforce a slightly different equality: the init node has one more out-edge and the *end* node has one more in-edge. The *init* node of the exception is fixed to be the *init* node of the state machine, while the *end* node of the transition is not identified yet at this stage. To elegantly cope with this exception, we introduce a set of additional transitions to the initial state machine, called *fictive* transitions. An example of such a fictive transition is presented in **Figure** 3.

These fictive transitions will enable us to post the same equality constraint on each node, independently of the possible exception on this constraint, because we will take the fictive transitions into account in the equality constraints. In the Eulerian state machine, if the exception turns out to happen, one fictive transition will link the end node to the init node, and ensure that the degree constraint will be enforced. If the exception does not happen, no fictive transition will be present

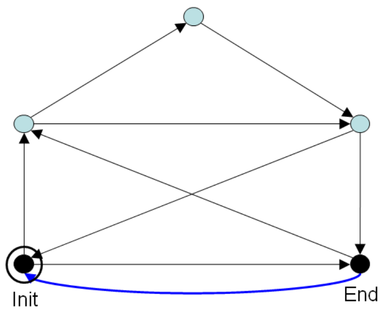


Figure 3: Adding a fictive transition.

in the Eulerian state machine. Each node but the init node has a fictive transition that links it to the *init* node. We represent the set of fictive transition by the set F. Each fictive transition f gets also an associated unknown variable x_f.

We now review each condition that must be enforced by these unknowns from **Section** 3.1 and from the problem statement in **Section** 2 and show how they are translated into constraints of an IP problem:

- **Constraint C1 - For each node, its in-degree is equal to its out-degree**. This translates neatly into an IP problem. Considering a node n of the graph, we can find all transitions reaching (resp. leaving) the node, including the fictive and reset ones, they are denoted as I_n (resp. O_n), we just need to post that $\sum_{e\in I_n} x_e = \sum_{e\in O_n} x_e$. This constraint is posted for each node n.

- **Constraint C2 - There is at most one fictive transition in the state machine**. It is is posted on the set of fictive transitions F: $\sum_{f\in F} x_f \leq 1$.

- **Constraint C3 - Each transition from the original machine is taken at least once**. It is to be posted for each transition of the original state machine, thus not for reset or fictive transition. For each transition t in the initial state machine, we have to post the following constraint: $x_e \geq 1$.

- **Constraint C4 - No transition can be taken a negative number of times**. This is straightforwardly translated into a MIP: for each transition e in the state machine, we have to post that $x_e \geq 0$. This is partially redundant with the previous constraint, so that it need only be posted on reset and fictive transitions.

- **Constraint C5 - The graph must be strongly connected**. This constraint is more complex. From **Section** 2 we know that the initial state machine is quasi-strongly connected, as there is a path from init to each node. We only need to ensure that there is a path from each node to init. This is implemented by adding the necessary reset transitions. Consider a state s of the state machine. It must be connected to *init*. We thus consider Z_s the set of states reachable from s. If init $\in Z_s$, the constraint is entailed for s. If init $\notin Z_s$, at least one reset transition must be added from one of the set if Z_s to establish the connection to *init*. Considering R_{Z_s}, the set of reset transitions from R that start in a state of Z_s, the corresponding constraint is as follows: $\sum_{r \in R_{Z_s}} x_r \geq 1$. Computing all the Z_s can be performed at once through the Floyd-Warshall algorithm [Flo62]. [1]

The **cost criterion** is composite and is expressed in terms of the IP problem as follows:

- The first cost criterion is that the number of reset transitions in the Eulerian path should be minimal. The associated objective function to minimize is: $\sum_{e \in R} x_e$.
- The second cost criterion is that the overall length of the test case should be minimal. The associated objective function to minimize is: $\sum_{e \in T} x_e$.

Combining the two objective functions into a single one is performed by summing them with a high weighting on the first criterion. This weighting is computed to be strictly bigger than the maximal value of the second criterion. We selected the value #Edge^2, where Edge is the set of transitions in the initial graph. This is higher than the number of edges in the Eulerian path because the worst case would be that taking one more transition of the state machine requires crossing the whole state machine again. Other values can be used, but the outcome of the optimization will be the same, for the x variables.

The result of the IP problem is the number of times each transitions must be represented in the transformed state machine to make it Eulerian with minimal cost.

The Eulerian state machine is elaborated from the initial one, the reset transitions and the computed x values: the states of the Eulerian state machine are the ones of the initial one, and each transition t of the initial state machine, or from the set of reset transition is present exactly x_t times in the Eulerian one. The potentially identified fictive transition is not added to the Eulerian state machine because it was only a modeling artifact to ease the encoding of the problem into an IP problem. It can be omitted in the Eulerian state machine without losing the Eulerian property of this machine thanks to the exception of the definition.

5 Implementation and Validation

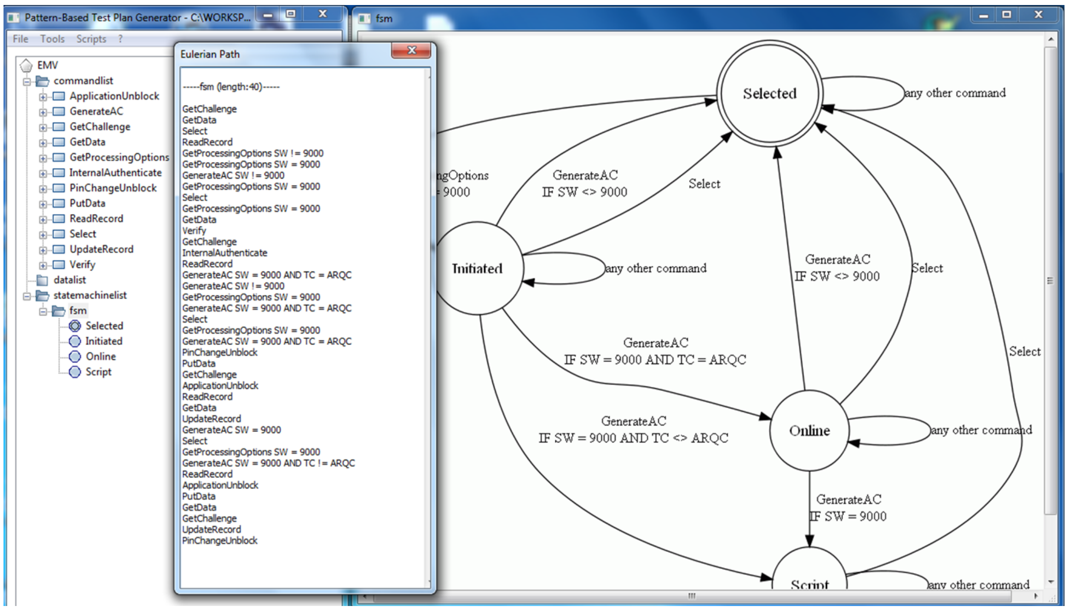
5.1 Implementation

An implementation of this approach has been developed in Python and relies on several open source libraries such as NetworkX and Coopr [Net, San]. NetworkX is a Python software package for the creation, manipulation, and study of complex networks. It offers a lot of useful features such as graph generators, network structure and analysis measures, and graph drawing. Coopr is a collection of Python software packages that supports a diverse set of optimization capabilities for formulating and analyzing optimization models. In particular, we used the Pyomo package which supports mathematical modeling of integer programs in Python. Pyomo enables

[1] In case we were only minimizing the number of resets with no concern about the total number of transitions of the test suite, we could consider a simpler approach for this constraint: adding at least one reset transition for each strongly connected component that has no edge getting out of it, and that does not contain the initial state. This approach would restrict the positions where reset transitions can be added, thus potentially increasing the total number of transitions of the test suite.

to define symbolic problems, create concrete problem instances, and solve these instances with standard solvers. We used the GLPK solver for the MIP resolution [GLP].

The prototype is fully integrated in a test management toolset in a smart card context [DPD+12]. **Figure** 4 shows the tool with the FSM editor open as well as a generated test scripts. Test scripts can easily be generated from those FSM as each transition is directly referencing a command with parameters. Special scripts are also available (e.g. for the reset transition).



```
-----fsm (length:40)-----

GetChallenge
GetData
Select
ReadRecord
GetProcessingOptions SW != 9000
GetProcessingOptions SW = 9000
GenerateAC SW != 9000
GetProcessingOptions SW = 9000
Select
GetProcessingOptions SW = 9000
GetData
Verify
GetChallenge
InternalAuthenticate
ReadRecord
GenerateAC SW = 9000 AND TC = ARQC
GenerateAC SW != 9000
GetProcessingOptions SW = 9000
GenerateAC SW = 9000 AND TC = ARQC
Select
GetProcessingOptions SW = 9000
GenerateAC SW = 9000 AND TC = ARQC
PinChangeUnblock
PutData
GetChallenge
ApplicationUnblock
ReadRecord
GetData
UpdateRecord
GenerateAC SW = 9000
Select
GetProcessingOptions SW = 9000
GenerateAC SW = 9000 AND TC != ARQC
ReadRecord
ApplicationUnblock
PutData
GetData
GetChallenge
UpdateRecord
PinChangeUnblock
```

Figure 4: Prototype integration inside the STMicroelectronics test tooling

5.2 Illustration on the EMV Case Study

For the EMV model presented in **Figure** 1, our tool finds a single test case that alone ensures the transition coverage criterion. No reset transition is needed here because the state machine is strongly connected from the start. The run time was around a tenth of second on a 2GHz machine. The single test case has 16 transitions. It is described in **Figure** 5.

5.3 Benchmarking

We have performed some benchmarking of our algorithm, both to check its efficiency and scalability, and to try to measure how efficient it could be with respect to the cost criterion. Benchmarking the cost criterion is actually only aimed at checking that minimizing it gives some good result and we know that our algorithm is optimal with respect to it.

Our benchmarking is based on randomly generated state machines with specific characteristics. They are generated randomly so as to reach a given number of nodes and transitions. We have swept the two dimensional space of number of nodes and transitions so as to give a wide view of the efficiency of our algorithm. The number of nodes range from 4 to 84 while the number of transitions ranges from 4 to the maximal connectivity the size of these state machines.

```
GetProcessingOptions SW=9000 (Selected -> Initiated)
GenerateAC SW<>9000 (Initiated -> Selected)
GetProcessingOptions SW=9000 (Selected -> Initiated)
Select  (Initiated -> Selected)
GetProcessingOptions SW=9000 (Selected -> Initiated)
GenerateAC SW=9000 AND TC=ARQC (Initiated -> Online)
GenerateAC SW<>9000 (Online -> Selected)
GetProcessingOptions SW=9000 (Selected -> Initiated)
GenerateAC SW=9000 & TC=ARQC (Initiated -> Online)
Select  (Online -> Selected)
GetProcessingOptions SW=9000 (Selected -> Initiated)
GenerateAC SW=9000 & TC=ARQC (Initiated -> Online)
GenerateAC SW=9000 (Online -> Script)
Select  (Script -> Selected)
GetProcessingOptions SW=9000 (Selected -> Initiated)
GenerateAC SW=9000 & TC<>ARQC (Initiated -> Script)
```

Figure 5: Single test trace elicited for the EMV state machine

There are 2268 test cases. The machine used for the benchmarking is a dual-core Intel 2,53 GHz with 3,5Gb of RAM running Microsoft Windows.

We have measured a set of parameters, both related to the generated test state machine and to the behavior of our algorithms, namely:

- The number of nodes that have no access to the initial state (those will trigger the need for inserting reset transitions)
- The diameter of the graph; to compare it to the length of the generated test suite
- The run time and memory consumption of our algorithm
- The number of test cases found in the generated test suites which is equal to the number of resets plus one
- The overall length of the generated test suite: this is the summed length of the test cases

Figure 6 reports the run time and peak memory consumption with respect both to the number of transitions and the number of nodes. Surprisingly, the run time seems to be nearly insensitive to the number of nodes. It seems to be more or less linear with respect to the number of transitions of the state machine. There is an exception in the run time, which is a graph with 45 edges and around 100 transitions which can either be a very badly shaped test case, or an external factor (e.g.: antivirus) which could not be reproduced. The peak memory consumption seems to be also linear with respect to the number of transitions, and seems to be much less dependent on the number of nodes.

The main objective function is the number of reset transitions. The left graph of **Figure 7** presents the number of reset transitions generated by our algorithm with respect to the number of nodes that have no path to the initial state. These are the nodes that potentially require the insertion of a reset transition in order to make the graph Eulerian. We have drawn the 1:1 line on the graph to enlighten two phenomena. First, the number of resets is often lower than the number of nodes that have no path to the initial state. This is because the algorithm manages to visit them

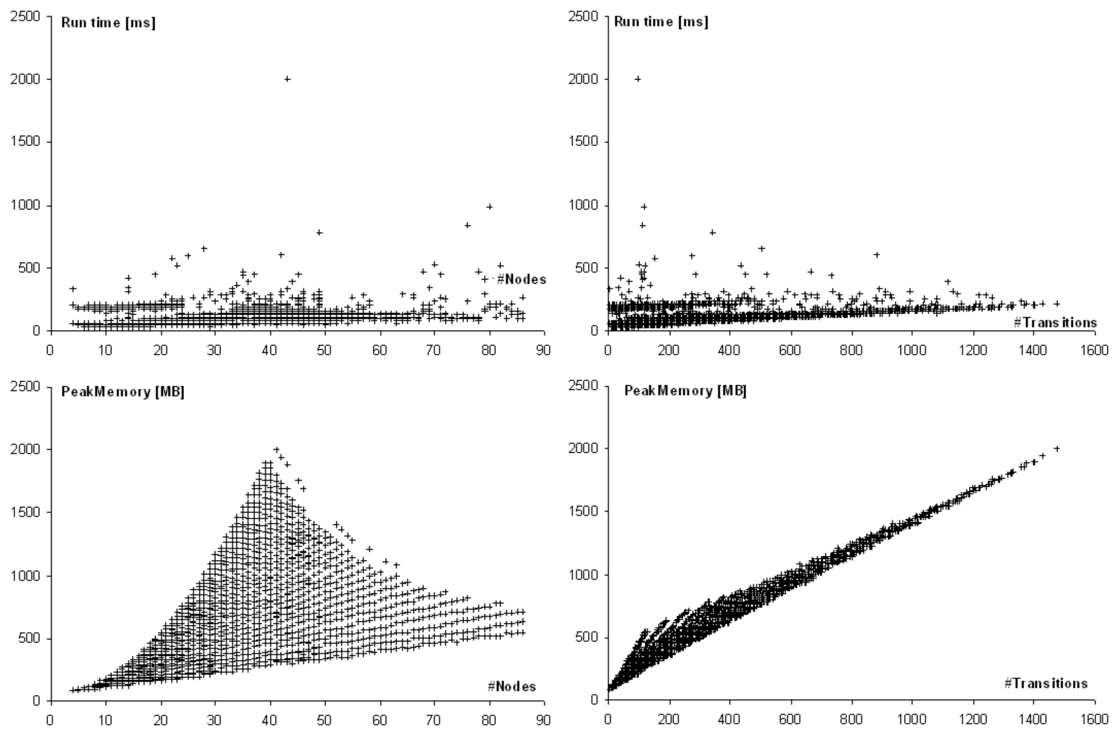


Figure 6: Run time and peak memory consumption of our algorithm on our benchmark suite

in a row, as they form a chain towards some "dead end" node. Second, there can be more resets inserted than the number of such nodes because such dead end node can have several incoming arcs; hence it requires as many resets to be inserted to ensure that the graph is Eulerian.

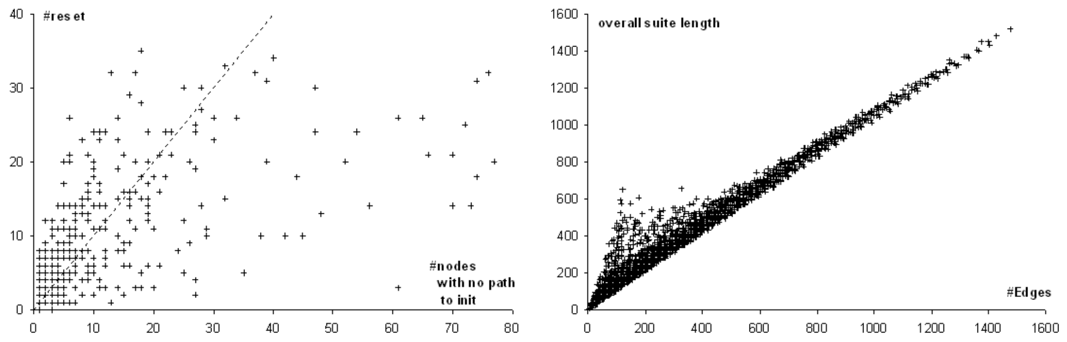


Figure 7: Number of generated resets wrt. number of nodes without path to init (left) and suite length wrt. number of edges (right)

The right graph of **Figure 7** presents the length of the generated test suite with respect to the number of edges of the graph. We can see that the dependence is mostly linear, except for small numbers of edges, where it might be higher.

Our graphs are generated randomly, so that, in the statistical sense, our benchmarks are only valid assuming a random set of graphs. Extreme cases can likely be found, such that the algorithm behaves very inefficiently. Nevertheless, these benchmarks give a good overview of the average behavior of the algorithm.

6 Related Work

Finite state machines are a formalism that is implicitly or explicitly present in a number of model-based testing approaches. We focus here on approaches with explicit FSM descriptions.

- The commercial QTronic tool suite [Ver] provides a graphical FSM editor. The FSM description can also be coupled with a richer JAVA-inspired language. Considering only the FSM part, it supports not only state and transition coverage but also more complex coverage criteria like transition pairs (all pairs of input/output transition to a state). However, it does not support the management of the cost of special transitions like resets. The underlying technology is based on algorithms, including symbolic state space analysis, constraint solving, and combinatorial optimization. A more complete evaluation of this tool can be found in [And10].

- SmartTesting relies on UML class and state diagrams to model the system under test [Sma, BGLP08]. More complex behaviors can be specified using UML OCL (Object Constraint Language). The test generation relies on a set-constraints based solver. It does not cope with reset costs.

- The UML-B plug-in for RODIN enables the specification of FSMs in graphical form with Event-B as underlying formalism [SB08, Abr10]. Model-based test generation is currently being developed based on the ProB model-checker which is implemented in SICStus Prolog and uses co-routining and finite domain constraint solving [LB08]. Transition costs are also currently not taken into account.

Other research has addressed the problem of generating reset-minimal test suites in the larger scope of generating a checking sequence and taking into account the notion of distinguishing sequences. A first work proposed some heuristics [UWZ97]. Hierons improved them based on optimizations of the state recognition sequences and their use to construct the test segments. This resulted in shorter checking sequences produced from minimal, completely specified, and deterministic finite state machines [HU02]. An improved algorithm was more recently proposed: it produces minimal length sequences in its class and does not require the FSM to be strongly connected [HU10]. The approach is totally different and does not rely on the notion of Eulerian path. Besides, we did not consider distinguishing sequences. The notion of distinguishing sequences is not relevant in the smartcard domain where a separate oracle is available (emulator implementation). However, in a more general context, guaranteeing distinguishing sequences would be relevant.

MIP-based tools have also been used in the context of test case generation. [NSZ07] presents how scheduling problems can be solved by means of a MIP and how the generated schedules can be used as test cases for real-time systems. They furthermore present how the randomization introduced by some MIP solvers on the generated schedules can lead to test cases exhibiting similar randomness.

An MIP-based tool able to statically check meta-properties of the state machine such as disjointness and exhaustiveness of transitions is described in [OQL08]. This work relies on a state-machine model manipulating numerical data. Their solver is also able to identify sets of values to inject in such state machine to lead it towards some pre-defined states. This can be used to generate test data that should bring the state machine into some predefined state. Iteratively, one should generate a single test case for each state through this procedure. Our approach does not encompass the decoration of state machine by condition and numerical state variables, but it provides test suite generation approach enforcing a coverage criterion for the whole test suite.

Consequently, we believe that the work described in this paper is an original algorithm for generating a checking sequence minimizing the number of reset transitions used.

7 Conclusions and Perspectives

This paper has presented an original algorithm that generates a test suite for a state machine using a constraint solving approach. The test suite ensures that each transition of the state machine is taken at least once, and guarantees to include as few test cases as possible, to keep the number of resets as low as possible. It furthermore ensures that the overall length of the test suite is as small as possible. The approach is original because it is based not on an exploratory approach such as depth first search, but on a graph transforms approach. It first transforms the state machine into an Eulerian one where such a test suite can be found straightforwardly. The transformation is computed efficiently with the help of a MIP solver.

The insights of this approach are interesting for the following reasons: despite the apparent complexity of the problem, the proposed solution is elegant and can easily be implemented. Its general performance is excellent and scales well.

An apparent limitation of our approach is the use of flat FSM while real world examples are expressed as parallel and hierarchical decomposition. For example, smartcards are expressed as the combination of multiple agents with specific behavior. They can be either generic (e.g. the presented EMV agent for the lifecycle, a PIN management agent, ...) or application specific. However, several algorithms exist to translate such problems to a flat FSM [DPC$^+$14]. The resulting FSM is more complex but our algorithm scales well.

Another area of extension would be to consider the notion of distinguishing sequences in order to detect invalid test executions without any external oracle. However, in our industrial case an oracle was available as the test sequence is being played both on a simulator and an implementation, including in hardware-in-the-loop mode which takes long reset time and which motivated this paper.

Acknowledgements: The research leading to the results presented in this paper has received funding from Wallonia through the project INOGRAMS (convention nr 7171). We also thank STMicroelectronics Belgium for providing us the case studies and allowing us to publish the result of this research.

Bibliography

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[And10] M. Andermo. Msc Thesis: Evaluation of Qtronic for Model-based Testing. Technical report, KTH, 2010.

[BGLP08] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux. A test generation solution to automate software testing. In *Proceedings of the 3rd international workshop on Automation of software test*. AST '08, pp. 45–48. ACM, New York, NY, USA, 2008.

[DPC$^+$14] X. Devroey, G. Perrouin, M. Cordy, A. Legay, P.-Y. Schobbens, P. Heymans. State Machine Flattening: Mapping Study and Assessment. *CoRR* abs/1403.5398, 2014.

[DPD$^+$12] N. Devos, C. Ponsard, J.-C. Deprez, R. Bauvin, B. Moriau, G. Anckaerts. Efficient Reuse of Domain-specific Test Knowledge: An Industrial Case in the Smart Card Domain. In *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12, pp. 1123–1132. IEEE Press, Piscataway, NJ, USA, 2012.

[DSVT07] A. C. Dias Neto, R. Subramanyan, M. Vieira, G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proc. of the 1st ACM international workshop on empirical assessment of software engineering languages and technologies*. WEASELTech '07, pp. 31–36. ACM, New York, NY, USA, 2007.

[EJ73] J. Edmonds, E. Johnson. Matching Euler tours and the Chinese postman problem. *Mathematical programming* 5(1):88–124, 1973.

[EMV08] EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems Book 3 Application Specification Version 4.2. Technical report, EMV, 2008. http://www.emvco.com/specifications.aspx?id=155.

[Flo62] R. W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM* 5(6):345, June 1962.

[Fow03] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 3 edition, 2003.

[GLP] GLPK. GNU Linear Programming Kit. http://www.gnu.org/s/glpk.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8(3):231–274, 1987.

[Hie73] C. Hierholzer. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen* 6(1):30–32, 1873.

[HU02] R. M. Hierons, H. Ural. Reduced Length Checking Sequences. *IEEE Trans. Comput.* 51:1111–1117, September 2002.

[HU10] R. M. Hierons, H. Ural. Generating a checking sequence with a minimum number of reset transitions. *Autom. Softw. Eng.* 17(3):217–250, 2010.

[IBM] IBM. ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer.

[LB08] M. Leuschel, M. Butler. ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer* 10(2):185–203, 2008.

[LPS] LPSolve. Reference Guide. http://lpsolve.sourceforge.net.

[Net] NetworkX. High productivity software for complex networks. http://networkx.lanl.gov/.

[NSZ07] A. Nahir, Y. Shiloach, A. Ziv. Using linear programming techniques for scheduling-based random test-case generation. In *Proc. of the 2nd Int. Haifa verification conference on Hardware and software, verification and testing*. HVC'06, pp. 16–33. Springer-Verlag, Berlin, Heidelberg, 2007.

[OQL08] M. Ouimet, M. Quenot, K. Lundqvist. Mixed Integer Programming for Automated Testing and Automated Verification of System Specifications. Technical report, Massachusetts Institute of Technology, 2008.

[San] Sandia National Labs. Coopr: A COmmon Optimization Python Repository. https://software.sandia.gov/trac/coopr/wiki/Documentation/CooprOverview.

[SB08] C. Snook, M. Butler. UML-B: A Plug-in for the Event-B Tool Set. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z*. ABZ '08, pp. 344–344. Springer-Verlag, Berlin, Heidelberg, 2008.

[SCI] SCIP. Solving Constraint Integer Programs. http://scip.zib.de.

[Sma] Smartesting. CertifyIt. http://http://www.smartesting.com/.

[Thi03] H. Thimbleby. The directed Chinese Postman Problem. Volume 33(11), pp. 1081–1096. John Wiley & Sons, Ltd., 2003.

[Tru93] R. J. Trudeau. *Introduction to Graph Theory*. Dover Books on Mathematics, 1993.

[UL06] M. Utting, B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier, 2006. 550 pages.

[UWZ97] H. Ural, X. Wu, F. Zhang. On Minimizing the Lengths of Checking Sequences. *IEEE Trans. Comput.* 46:93–99, January 1997.

[Ver] Verisoft. Qtronic. http://www.verifysoft.com/en_conformiq_automatic_test_generation.html.