

Automatic test generation on a (U)SIM smartcard

Céline Bigot¹, Alain Faivre², Christophe Gaston² and Julien Simon¹

¹ Oberthur Card Systems, R&D – EMC, 71-73 rue des Hautes-Patures,
92726 Nanterre Cedex, France,
{c.bigot, j.simon}@oberthurcs.com

² CEA/LIST, Saclay, F-91191 Gif sur Yvette Cedex, France
{alain.faivre, christophe.gaston}@cea.fr

Abstract. Most of the time, test cases of smartcard software are produced by specialized engineers in a proprietary language. Test represents generally half of smartcard development effort. With more and more household use of semi-formal and formal modeling languages, like UML, and the emergence of automatic test generators in the industry, we study a way to adapt these techniques for smartcard. In this paper, we present an automatic test generator, called AGATHA, and its architecture, which can treat UML specifications. Then, we propose a way to model smartcard functionalities in UML. We use the test generator on our UML models and produce automatically our first test cases for GSM functionalities.

1 Introduction

It's not necessary to remind that in any industry, more later a bug is discovered in a development process more it costs to correct it. Today, in the smartcard industry, half of the effort of the development activity is devoted to test. The test activity regroups:

- unit testing, carried out during the programming activity by programmers, which ensures that each elementary element has correct behaviour and eliminates basic programming errors,
- α -testing carried out after the programming activity by α -testers, which ensures that smartcard has correct behaviour in relation to functionalities and specificities described in its specification,
- β -testing carried out after α -tests by β -testers, which ensures that smartcard in a mobile phone, in a payment machine or in some other devices also respects its specification.

In the context of this long and complex process, handmade by programmers and testers, we would like to study the possible automatic generation of a part of these tests. Our first idea, described in this article, consists to take into account the α -test activity. By automatic test generation, we expect to increase the coverage and the quality of test in order to ensure a complete validation of the specification.

Moreover, with the increase of system complexity, it's difficult between two versions of a project to know which tests evolve, which are obsolete, etc. It's also difficult for a non-tester to understand produced tests. Thus, our idea is to combine auto-

matic test generation with a simple formalism to represent specifications, tests and evolutions of these.

The need of methods and tools for validation is not recent and a lot of researches try to fill the gaps. For test generation, we can take as examples [14, 20, 4, 31]. Semi-formal and formal methods, such as UML [35], B [3] or 34 [34], allow an abstract design for a behavioural specification of the system under test. Thanks to simple, expressive and abstract notations, textual or graphical, we can easily use these types of formalisms to design smartcard. Moreover, these formalisms allow the use of existing validation tools.

The last few years, several studies were conducted on design and validation of smartcard software. For example, [12] represents results on the SEPS standard, [5] shows validation results on the GSM 11.11 standard [1], [30] used automated test generation on the WAP Identity Module, [6] describes techniques which can be apply at different levels of smartcard software, [7] represents an automatic test generation with the LEIRIOS tool [26] from B specifications, [29] presents a method to automatically generate test for Java card applets and [9] offers a semi formal model of Java Card applications in UML.

In our context, we would like to use a more simple and graphical formalism, which can be use by any engineer. With the emergence of UML in industry and the multiple types of diagrams offered, this formalism represents a good alternative. In the panel of automatic test generators (see [32] for examples of automatic test generators), we were interested by the symbolic approach of the AGATHA¹ tool [18, 25, 33, 8, 28, 15], developed at the CEA²-List.

Therefore, the paper is organized as follows. Firstly, we present the AGATHA tool and the automatic test generation. Secondly, after presenting how we can model a part of smartcard in UML, we describe the use of AGATHA on our semi-formal models and present our first results on a PIN command. We finally conclude and explain our future actions.

2 AGATHA, an automatic test generator

There exists several ways to validate system specifications. A first one consists in theorem proving and model checking [11]. These kinds of techniques have proved successfully their use for the validation of critical parts systems. But two major drawbacks to these techniques remain: the combinatorial explosion due to variable domains, for the model checking, and a need for high-level skills from the developer, who must be aware of formal method fundamentals, for theorem proving.

Automatic test generation is another way to tackle the problem of system validation. Conformance testing is the most well known part of this domain. Our first purpose is to validate the specification itself, and by the way generate tests in order to simulate them in the specification.

¹ AGATHA for Atelier de Génération Automatique de Test Holistiques à partir d'Automates, automatic holistic tests generation framework for automata

² Commissariat à l'Energie Atomique – atomic energy research center of France

Most validation tools use enumerative techniques and are therefore limited by the combinatorial explosion problem when trying to exhaustively identify the numerical behaviours of a system. Several validation tools focus verification on particular aspects: test purpose [16], temporal properties [37], etc ...

The solution proposed by AGATHA is an exhaustive symbolic path coverage, which apart test generation permits to detect specification deadlocks, unreachable transitions, losses of messages, etc.

Moreover, the AGATHA toolset is designed to be as transparent as possible in order to reduce the effort of detection and comprehension of errors. In that context, it is not necessary to be an expert in formal methods, as for model checkers or theorem provers, to interpret AGATHA's results and to correct specifications and implementations.

The following subsections present the AGATHA architecture and an overview of the different academic techniques used in order to reach minimal exhaustive path coverage.

2.1 An overview

The AGATHA approach, developed at the CEA-LIST, intends to help conception and validation of formal specifications modelled with communicant automata systems. Thus, with symbolic execution techniques, AGATHA computes the exhaustive symbolic behaviour graph of the specification. Then, from this graph, it generates test cases used to debug the specification or to validate the implementation, all along an incremental conception process.

Figure 1 presents the general architecture of AGATHA.

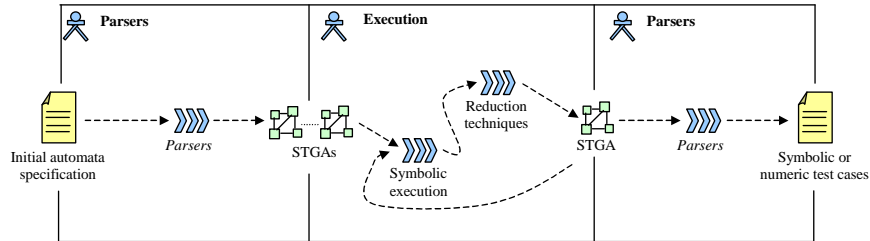


Fig. 1. General architecture of AGATHA

The tool treats automata specifications and translates it in its internal language, called STGA (Symbolic Transition Graph with Assignment) [27]. This translation allows the symbolic execution of the specification as defined in [19]. Thus, it allows obtaining an exhaustive behaviour graph of the specification. Thanks to reduction techniques defined in [25] and in [33], with the help of the rewriting tool Brute [21], the graph is reduced in a particular STGA. On this particular STGA, AGATHA uses a constraint solver Omega [23] providing for each path of the graph corresponding to a symbolic behaviour one or more numerical test cases.

2.2 Main principle: symbolic execution

At the beginning, symbolic execution has been proposed in [24] and in [10] to construct structural tests for sequential programs. The main idea of symbolic execution consists to use symbols as entry data of the program, denoting any entry data, instead of numerical values and to interpret the entry language in a way which allows to manipulate symbolic expression instead of numerical ones. In the context of AGATHA, the CEA-LIST adapts symbolic execution to generate tests from specifications based on automata.

The input language of AGATHA is based on the STGA formalism [27], which is a symbolic transition graph. As every graph formalisms, a STGA is composed of states and transitions. This type of graph allows to represent in an abstract manner all behaviours of a specification. Transitions represent events that allow the evolution of the system: events can be received or emitted, firing a transition can be conditioned by a logical expression and system variables can evolve.

STGA example. Figure 2 presents a STGA sample of an elevator system. It contains four states and six transitions. The initial state is Q_0 .

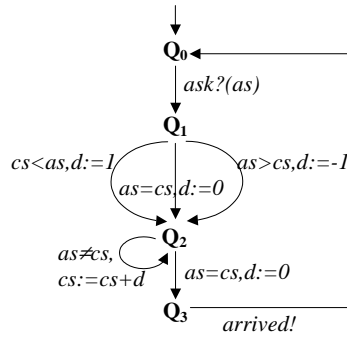


Fig. 2. STGA sample of an elevator system

To fire the output transition of Q_0 , the elevator system waits the reception of the message *ask*, denoted $ask?(as)$, which represents a call to the stage *as* by the user in the cabin. After the trigger of the transition, the system is in the state Q_1 . Q_1 has three output transitions towards Q_2 . The left one is conditioned by the logical expression: $cs < as$ meaning the asked stage *as* is over ($<$ sign) the current stage represented by the *cs* variable. The right one is conditioned by: $cs > as$ meaning the asked stage is under the current stage. The middle one is conditioned by: $cs = as$ meaning the asked stage is equal to the current stage. In the first case the elevator needs to make move up the cabin, materialized by the operation $d := 1$ where *d* represents the direction, while in the second case the elevator needs to make move down the cabin, materialized by $d := -1$, and in the third one to not move the cabin, materialized by $d := 0$. Q_2 has two outgoing transitions: one with Q_2 for target and one with Q_3 . The transition with Q_2 for target is conditioned by the logical expression: $as \neq cs$ and increases the current stage *cs* with the direction *d*, materialized by the operation $cs := cs + d$. This transition

means that as long as the current stage is different of the asked stage, the cabin has to continue to go up or down. The second transition is conditioned by: $as=cs$ and initializes the direction d to 0. This transition means that if the current stage is equal to the asked stage, the cabin is stopped. The Q_3 outgoing transition allows to come back in Q_0 and announce to the user in the cabin that the elevator has reach the asked stage, which is represented by the emission of the message *arrived*, denoted *arrived!*.

◇

recueillir

In the AGATHA context, [25] redefines symbolic execution for STGA using the approach defined in [19]. In our case, symbolic execution simulates the behaviour of a STGA specification in assigning symbolic values to variables instead of numerical ones. Then, the specification is executed according to the semantics of each instruction and communication.

The general principle of symbolic execution consists in computing symbolic states of a system, each of them being denoted by a couple (guard, symbolic memory), where:

- guard is the condition needed to reach this symbolic state,
- symbolic memory is a function which associates to each variable of the system an expression based on symbolic input values.

The value associated to a variable in symbolic state corresponding to an execution path, from the initial state of the system, is computed by interpreting one by one instructions met along this execution path. The associated guard is composed of the conjunction of all the execution conditions (denoted by constraints on symbolic input values) met along the execution path considered. This guard is called path condition or PC. To simplify this type of expression, AGATHA uses the simplifier Brute [21] extracted from CafeOBJ tool of JAIST (Japan Advanced Institute Technology). This is a rewriting tool, which transforms terms in normal form with the help of a set of rewriting rules and evaluation strategies defined by the AGATHA user.

The result of a symbolic execution is a symbolic execution tree such as each path represents the symbolic evolution of all variables according to initial symbolic values. Each path is a particular behaviour of the STGA specification.

Symbolic execution example. In our example of elevator presented in Figure 2, an extract of the symbolic execution tree computed by symbolic execution is presented in Figure 3.

At the initialization, the STGA specification obtained by symbolic execution is in a state corresponding to the initial state Q_0 of the elevator system. The elevator specification manipulates the variables: as , cs and d on which there is no initial constraint. A symbolic constant is assigned to each variable: a_0 , b_0 and c_0 (resp.) on which there is no initial constraint denoted by the condition true. The elevator system can evolve if it receives the ask message with a value. This value is stored in the variable as and supposed to have the symbolic value a_1 . a_1 is assigned to the as variable. As the firing of the transition is not conditioned the condition to reach this second symbolic state is always true (true and true = true).

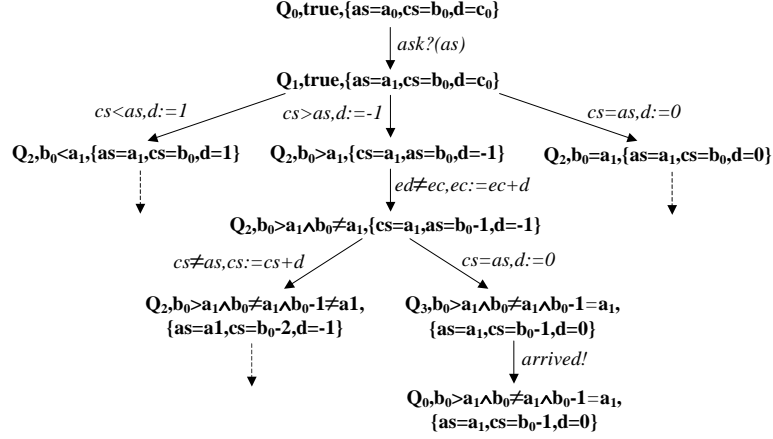


Fig. 3. Extract of the symbolic execution of the elevator system

To leave the Q_1 state, there are three transitions. So to leave the symbolic state Q_1 , there are also three transitions. As the first transition is conditioned by the expression: $cs < as$ and $cs = b_0$ and $as = a_1$ in this state, the condition to reach the symbolic state Q_2 corresponds to $b_0 < a_1$. On this transition, 1 is also assigned to d which is reflected in the symbolic state. The second transition is conditioned by the expression: $cs > as$, so the condition to reach the symbolic state Q_2 corresponds to $b_0 > a_1$ and -1 is assigned to d . The third transition is conditioned by the expression: $cs = as$, so the condition to reach the symbolic state Q_2 corresponds to $b_0 = a_1$ and 0 is assigned to d . To leave the symbolic state Q_2 where $d = -1$, two transitions have to be considered. The first one is conditioned by: $as = cs$ and the second one by: $as \neq cs$. However, as $b_0 > a_1$, inevitably $as \neq cs$ and only the second transition can be fired. The output transition of the symbolic state Q_2 where $d = -1$ leads to another symbolic state Q_2 reached if the condition: $b_0 > a_1$ and $as \neq cs$ is verified and such as $as = a_1$, $cs = b_0 - 1$ and $d = -1$. In this state, we can also fire the two same transitions. As the symbolic value of cs evolves the two transitions can be fired. The first one leads to the symbolic state Q_3 and the second one to another symbolic state Q_2 . The firing of the first transition implies that the condition to verify to reach the symbolic state Q_3 is: $b_0 > a_1$ and $as \neq cs$ and $a_1 = b_0 - 1$ and $as = a_1$, $cs = b_0 - 1$ and $d = 0$. To leave this state the only transition is conditioned by the emission of the message *arrived* and allows the system to come back in the state Q_1 .

The other steps of the calculus are based on the same principles.

◇

2.3 Further techniques

As the symbolic execution tree represents all behaviours of a specification, its construction is subordinate to reduction procedures in order to eliminate as many redundant paths as possible. There exists different tactics such as:

- use a classical graph coverage, as for example a transition coverage (the symbolic execution stops when all the transitions are fired once if possible), a state coverage, a path coverage, etc,
- cut “empty” path conditions when detected both from a boolean criteria or polyhedral criteria. AGATHA uses a constraint solver, Omega, based on Presburger theory [23] to achieve that,
- avoid computation of a path deductible from another modulo an interleaving detection less sophisticated than in [36]: an internal transition without any temporal constraint with other transitions,
- compute comparison procedures for each symbolic node and refer to an already existing symbolic node.

These procedures are necessary to avoid the state explosion problem.

AGATHA uses several heuristics to compute comparison procedures for each symbolic node:

- an equality procedure: two symbolic nodes are equivalent if the corresponding control nodes are the same and the symbolic guards are syntactically equal,
- an inclusion procedure: two symbolic nodes are equivalent if the corresponding control nodes are the same and if the polyhedron induced by variables domains defined by the guard of one is included in the other polyhedron,
- an equivalence procedure: two symbolic nodes are equivalent if the corresponding control nodes are the same and if the polyhedron induced by variables domains defined by the guard of one is equal to the other polyhedron.

As symbolic expressions of variables may also quickly grow, a last simplification procedure must be applied “on the fly” in order to shorten expression and to detect useless paths [17]. We use a simplifier based on rewriting techniques [13, 22]. The rewriting engine is Brute. The rewriting rules used by Brute are actually composed of more than three hundred rules. These rules allow both to maintain symbolic expressions within a reasonable size range, and to obtain normal forms for the expressions, easing the comparison between expressions needed by algorithms such as comparison procedures.

Others tactics and reduction techniques have been introduced in [33] and in [32]. Generally, a mix of the different tactics is used to obtain the minimal result requisite to guarantee the entire coverage of the specification.

2.4 Test extraction

Symbolic test cases are extracted from the symbolic execution tree. As each path of the tree represents a symbolic behaviour of the specification, one test case is extracted from each leaf of this symbolic tree. From each symbolic test case, one or more numerical test cases may be produced with the help of constraint resolution techniques used on the path condition associated to the tree leaves. The constraint solver is used to extract the symbolic value of each variables with the associated path condition and

to generate numerical values which respect the path condition. The choice of the constraint solver connected to AGATHA depends on the applicative context. For example, we can use the Omega tool.

Test extraction example. With the symbolic execution of the elevator system, see Figure 3, we identify the path: $Q_0Q_1Q_2Q_3Q_0$. This path represents the symbolic test case such as the asked stage is under the current stage of one stage. To generate a numeric test case corresponding to this symbolic test case, we have to find numerical values for: a_0 , a_1 , b_0 and c_0 which verify the path condition: $b_0 > a_1 \wedge b_0 \neq a_1 \wedge b_0 - 1 = a_1$.

For example, we can choose: $a_0 = 2$, $a_1 = 3$, $b_0 = 2$ and $c_0 = 0$.

Any other series of numerical values verifying the path condition is valid. Not that the size of our elevator, which is not defined, doesn't step in the symbolic calculus. Thus, our specification allows representing an elevator with two, three or more stages.

◇

These test cases can be simulated either on the industrial tool that allows generating the specifications, or on the implementation. It often requires an adjustment to the adequate format.

3. Application to smartcard

The aim of this article is to study the usability of the AGATHA tool in smartcard domain. To begin our experience, we limit our domain to embedded mobile communication smartcard that means SIM card. For SIM cards, there are different standards, which describe a lot of card specificities. Function specifications are described in the 3GPP 11.11 standard [1] and tests on these functions are described in the 3GPP 11.17 standard [2]. For Oberthur Card System, a test case is a sequence of instructions in a proprietary language.

In this section, we propose a UML representation for test cases or SIM smartcard behaviours. Then, we present results obtain by the application of the automatic test generation on these UML models.

3.1 A UML representation for smartcard test

As smartcard test are sequences of instructions and as we would represent smartcard tests and smartcard behaviours, we propose to use UML state diagrams. This type of diagram is very intuitive: it's a sort of automata language, with states and transitions. The firing of a transition can be conditioned by a message reception, a message emission, a logical expression, ... For our work, we only use a sub-part and not entire powerful of state diagram notations.

Suppose that we would like to represent a test case from the 3GPP 11.17 standard. As for a function, the test is a sequence of instructions, the corresponding UML state diagram is also a sequence of state and transition. In some case, we can identify sub-parts in a test case and represent these in the UML state diagram.

The *CHANGE CHV*³ function example: UML representation for test cases. Above all, we recall the *CHANGE CHV* specification extracted of the 3GPP 11.11 standard:

The *CHANGE CHV* function assigns a new value to the relevant CHV subject to the following conditions being fulfilled: CHV is not disabled; CHV is not blocked.

The old and new CHV shall be presented.

1) If the old CHV presented is correct, the number of remaining CHV attempts for that CHV shall be reset to its initial value 3 and the new value for the CHV becomes valid.

2) If the old CHV presented is false, the number of remaining CHV attempts for that CHV shall be decremented and the value of the CHV is unchanged. After 3 consecutive false CHV presentations, not necessarily in the same card session, the respective CHV shall be blocked and the access condition can never be fulfilled until the *UNBLOCK CHV* function has been performed successfully on the respective CHV.

Input: indication CHV1, old CHV1, new CHV1.

Output: none.

[1] p.34

The test case of the *CHANGE CHV* function, extracted from the 3GPP 11.17 standard (see [2] pp.65-67), is composed of:

- 1) an incorrect *CHANGE CHV*, steady of a status verification: how much attempts remained, a correct *CHANGE CHV* and a status verification,
- 2) two incorrect *CHANGE CHV*, steady of a reset, an incorrect *CHANGE CHV*, a reset, an incorrect *CHANGE CHV* and a correct *UNBLOCK CHV*,
- 3) a correct *DISABLE CHV*, an incorrect *CHANGE CHV* and a correct *ENABLE CHV*.

A correct function is characterized by a returned status 90 00 and an incorrect status is characterized by a returned status 98 04 or 98 40, this one meaning that the smartcard is blocked. A status verification is done by comparison of expected data and effective data.

The corresponding state diagram is presented Figure 4. The initial state is •. We suppose that the output transition of the initial state to A state contains data for initializing smartcard such as the personalization. A holds two output transitions.

The right one to C is conditioned by the reception of the message *disable_chv(chv1_ref, used_chv1)* which represents the *DISABLE CHV* function. This transition is also conditioned by the logical expression *used_chv1 = chv1*, which represents the fact that the chv used by *DISABLE CHV*, is equal to the chv of the card. The expected status for this reception is 90 00. Next transitions represent the sub-case 3 of the *CHANGE CHV* test.

The left one to B is conditioned by the reception of the message *change_chv(chv1_ref, used_chv1, new_chv1)*, which represents the *CHANGE CHV* function. This transition is also conditioned by the logical expression *not(used_chv1*

³ chv for card holder verification information; access condition used by the SIM for the verification of the identity of the user.

= *chv1*) which represents the fact that the *chv* used by CHANGE CHV is not equal to the *chv* of the card. The expected status for this reception is 98 04. C holds two output transitions. The right path represents the sub-case 1 of the CHANGE CHV test and the left one the sub-case 2.

A status verification is represented by a transition conditioned by the reception of message *status(n)* where *n* represents the size of the data to verify, in byte, 22 for our example. The expected status for this reception is 90 00. The data to compare are given in the <<compare>> stereotype. For example, on the transition from B to E, the <<compare>> stereotype contains: *xxxx xxxx xxxx xx xxxxxx xx xx xx xx xx xx xx xx 83 xxxxxx* which means the 19th byte is 83 and other bytes are any value, denoted *x*.

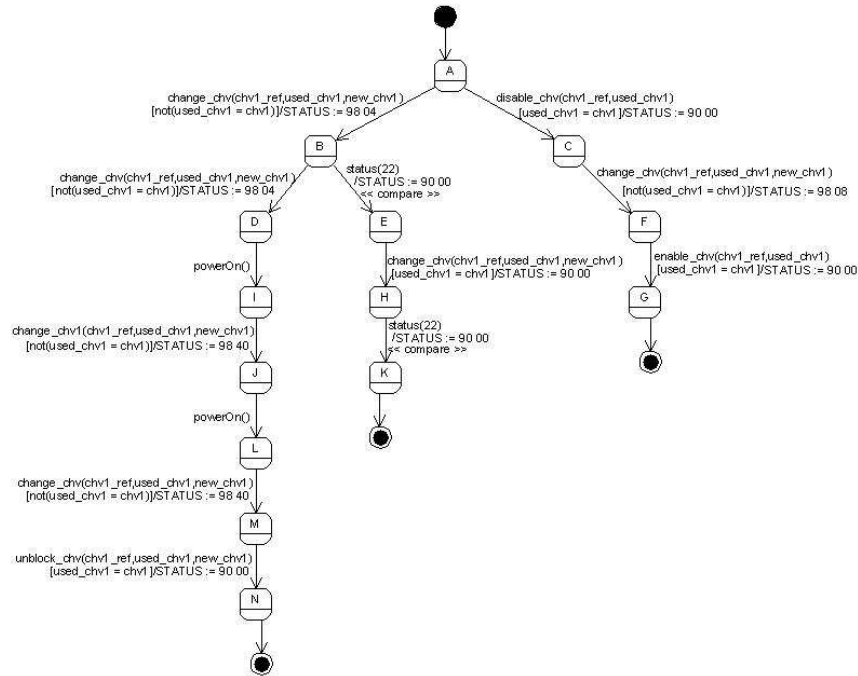


Fig. 4. The 3GPP 11.17 test of the CHANGE CHV function

◇

3.2 A UML representation for smartcard specification

With a UML state diagram, we can also represent a function specification. This diagram contains all the behaviours of a function in the same way that for a test. This abstract vision allows representing a function specification exhaustively. For example, on the CHANGE CHV function, this diagram has to represent a case with a direct correct CHANGE CHV, which is not considering in Figure 4.

The *CHANGE CHV* function example: UML representation for specification. To represent the *CHANGE CHV* specification exhaustively, we consider the 3GPP 11.11 standard given before. The corresponding state diagram is presented Figure 5.

As we abstract the behaviour of the function we introduce a counter *try*, which represents the number of attempts to change a *chv*. The initial state is •. The output transition of the initial state is enriched with the initialization of the *try* variable to 3 as said in the specification.

The sub-case 1 of the specification is represented with bold lines: a *change_chv(chv1_ref, used_chv1, new_chv1)* message is received. The *used_chv1* is equal to *chv1* so the expected status is 90 00, the *chv1* is changed to *new_chv1* and we can verify the status by the transition from A to A with the *status* message. As this diagram is an abstraction of the specification, the <<compare>> stereotype contains: *xxxx xxxx xxxx xx xxxxxx xx xx xx xx xx xx xx (80+try) xxxxxx*.

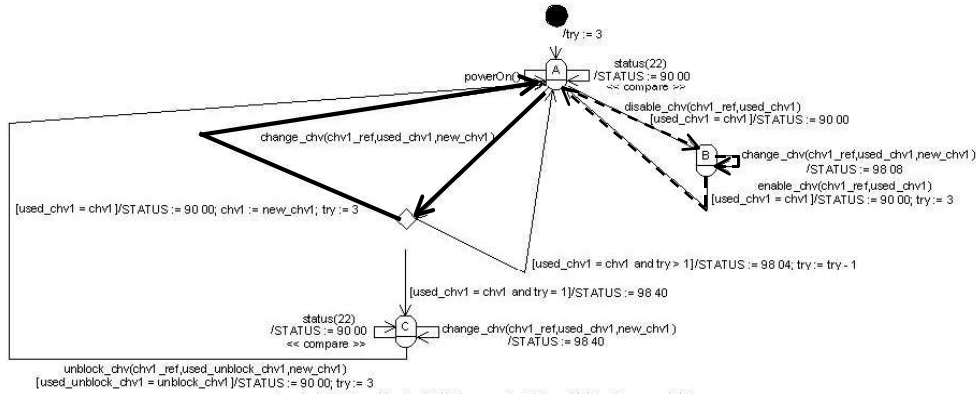


Fig. 5. Example of an abstract state-transition diagram for the *CHANGE CHV* functionalities

The sub-case 2 of the specification is represented with normal lines: a *change_chv* is received. The *used_chv1* is not equal to *chv1* so the expected status is 94 04 if *try* is different of 1 and is 98 40 if *try* is equal to one. If status is 94 04, the transition goes in A and we can verify the status or reset the session card. If the status is 98 40, the card is blocked and the transition goes in C where we can verify the status and receive *change_chv* messages. As the card is blocked, nothing appends. Except if an *unblock_chv(chv1_ref, used_unblock_chv1, new_chv1)* message is received with *used_unblock_chv1* equal to *unblock_chv1*.

We enrich the specification with a behaviour describe in the 3GPP 11.17 test but absent of 3GPP 11.11 specification: the use of a correct *DISABLE CHV* before a *CHANGE CHV*. This behaviour is designed with dashed lines.

◇

3.3 Automatic test generation for UML smartcard model

In part 2.2, we present how the AGATHA tool can generate automatically test cases. We apply this tool on our UML diagrams.

Firstly, as our representation of smartcard test is very sequential, we use a coverage of transitions to compute a set of symbolic test cases. In that case for our example, AGATHA computes three paths. AGATHA extracts three numerical test cases. For a card which validates the 3GPP 11.17 tests, it also validates these automatically generated tests.

Secondly, for our representation of smartcard specification, we use a more complicated criterion, the inclusion one to cover all the symbolic behaviours. In that case for our example, AGATHA computes more than two hundred paths, each of them corresponding to a symbolic behaviour. AGATHA extracts as many tests cases. On a card which validates the 3GPP 11.17 tests, it has also to validate these automatically generated tests. It could be impressive to pass two hundred tests for a simple function but we test all possible behaviours of the CHANGE CHV function. Current works on AGATHA will certainly permit to reduce this number of tests with some optimization associated to the inclusion criteria. But in our case this reduction will not be very important due to the fact that the number of distinct symbolic behaviours associated to our example remains very close to the present one calculated by AGATHA: this is the price of exhaustivity.

4 Conclusions

In this article, we have summarized a solution to automatically generate tests for smartcard functions. Assuming the validity of our approach, we have presented an automatic test generator, AGATHA based on symbolic execution techniques. We have also presented a way to design smartcard function with UML state diagram. We have used AGATHA on our UML diagrams and exposed obtained results. This first experience shows that it is possible to generate tests for smartcard functions in an automatic way. Surely, and this is our first objective, our approach has to be used in a real context and in a complete development cycle of a smartcard to completely improve its efficiency. We could reasonably hope an increase of the coverage and quality of test for each function taken separately.

The use of UML state diagrams to design smartcard behaviours allows us to consider more global behaviours that mix different smartcard functions. Then we could stress smartcard and control results. We also could consider atypical (or negative) tests which allow to verify smartcard reactions outside of the admissible input domain define by the specification. In this context we could ensure a complete validation of smartcards.

Last point, as AGATHA is not only a test generator, we envisage validating smartcard properties corresponding to a security policy as defined for example in common criteria. In this context, we could ensure security properties of smartcards.

References

- [1] European Telecommunications Standards Institute, F-06921 Sophia Antipolis (France), *GSM 11.11 v7.2.0 Technical Specification*, 1999
- [2] European Telecommunications Standards Institute, F-06921 Sophia Antipolis (France), *GSM 11.17 v8.1.0 Technical Specification*, 1999
- [3] J.R. Abrial, *The B-Book, Assigning programs to meanings*, Cambridge University Press, 1996
- [4] S. Behnia, H. Waeselynick, *Test criteria definition for B models*, in procs. of the World Congress on Formal Methods (FM'99), vol.1708 of LNCS, pp.509-529, Toulouse (France), 1999
- [5] E. Bernard, B. Legeard, X. Luck, F. Peureux, *Generation of test sequences from formal specifications: GSM 11.11 standard case-study*, The Journal of Software Practice and Experience, vol.34.10 pp.915-948, 2004
- [6] F. Bouquet, F. Peureux, *Generation of functional test sequences from B formal specifications – Presentation and industrial case-study*, in procs. of the 16th International Conference on Automated Software Engineering (ASE'01), pp.377-381, San Diego (USA), November 2003
- [7] F. Bouquet, B. Legeard, F. Peureux, E. Torrebore, *Mastering Test Generation from Smart-card Software Formal Models*, in Procs of the International Workshop on Construction and Analysis of Safe, secure and Interoperable Smart devices (CASSIS'04), vol.3362 of LNCS, pp.70-85, Marseille (France), March 2004
- [8] C. Bigot, A. Faivre, J.P. Gallois, A. Lapitre, D. Lugato, J.Y. Pierron, N. Rapin, *Automatic test generation with AGATHA*, TACAS, 7-11 April 2003
- [9] O. Carre, H. Martin, J.J. Vandewalle, *A semi formal model of Java Card 2.1 in UML*, in 1st Gemplus Developer Conference, Paris, France, June 21-22, 1999
- [10] L.A. Clarke, *A System to Generate Test Data and Symbolically Execute Programs*, IEEE Transactions on Software Engineering, vol.SE-4 n.3, PP.178-187, September 1976
- [11] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Cheking*, The MIT press, 1999
- [12] D. Clarke, T. Jérón, V. Rusu, E. Zinovieva, *Automated test and oracle generation for smart-card applications*, in procs. of the International Conference on Research in Smart-cards (e-Smart'01), vol.2140 of LNCS, pp.58-70, Cannes (France), September 2001
- [13] N. Dershowitz, J.P. Jouannaud, *Rewrite systems*, Handbook of Theoretical Computer Science, vol. B pp.243-320, ed. J. van Leeuwen, North Holland, 1990
- [14] J. Dick, A. Faivre, *Automating the generation and sequencing of test cases from model-based specifications*, in procs. of the International Conference on Formal Methods Europe (FME'93), vol.670 of LNCS, pp.268-284, April 1993
- [15] A. Faivre, C. Gaston, *Test generation methodology based on symbolic execution for the Common Criteria higher levels*, in MoDeVa workshop, Montego Bay (Jamaica), October 2005
- [16] J.C. Fernandez, C. Jard, T. Jérón, C. Viho, *Using on the fly verification techniques for the generation of test suites*, Proceedings of CAV'96, LNCS 1102, Springer, New Brunswick, n.46, pp.145-150, 1997
- [17] J.P. Gallois, A. Lanusse, *Le test structurel pour la vérification de spécifications de systèmes industriels*, Génie logiciel n.46 pp.145-150, 1997
- [18] J.-P. Gallois, A. Lapitre, P. Lé, *Analyse de spécifications industrielles et génération automatique de tests*, ICSSEA'99, CNAM-Paris, France, 8-10 décembre, 1999
- [19] M. Hennessy, H. Lin, *Symbolic bisimulations*, Theoretical Computer Science, Vol.138 pp.353-389, Elsevier, 1995
- [20] R. Hierons, *Testing from Z specification*, The journal of Software Testing, Verification and Reliability, vol.7 pp.19-33, 1997

- [21] M. Ishisone, T. Sawada, *Brute: brute force rewriting engine*, GAIST, <http://www.theta.theta.ro/cafeobj>, January 2001
- [22] J.W.Klop, *Term of rewriting systems*, Handboof of Logic in Computer Science, vol. 1, ed. D. Gabbay S. Abramski and T. Maibaum, Oxford University Press, 1991
- [23] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, *The Omega Library version 1.1.0*, University of Maryland, <http://www.cs.umd.edu/projects/omega>, November 1996
- [24] J.C .King, *Symbolic Execution and Program Testing*, communications de l'ACM, vol.19 n.7, pp.385-394, July 1976
- [25] A. Lapitre, *Procédure de réduction pour les systèmes à base d'automates communicants : formalisation et mise en oeuvre*, Phd Thesis, University of Paris XI, in collaboration with the CEA, December 2002
- [26] <http://www.leirios.com/index.php>
- [27] H. Lin, *Symbolic Transition Graph with Assignment*, CONCUR'96, Springer-Verlag, LNCS, Pise (Italie), August 1996
- [28] D. Lugato, C. Bigot, Y. Valot, *Validation and automatic test generation on UML models : the AGATHA approach*, STTT (Software Tools for Technology Transfer), vol.5 n.2 pp.124-139, March 2004, Springer
- [29] H. Martin, L. du Bousquet, *Automatic test generation for Java-Card applets*, in Java card Workshop, Cannes (France), September 2000
- [30] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglestorfer, S. Kriebel, K.Scholl, *Model-based test case generation for smartcards*, in procs. of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03), vol.80 of ENTCS, Trondheim (Norway), June 2003
- [31] J.-Y. Pierron, J.-P. Gallois, E. Fievet, A. Lapitre, D. Lugato, *Validation de systèmes industriels par le test symbolique sur spécification STATEMATE*, ICSSEA'00, CNAM-Paris, France, December 5-8, 2000
- [32] J.Y. Pierron, *Définition de critères de sélection de tests fonctionnels pour la validation des systèmes électroniques embarqués*, Phd Thesis, University of Evry, France, in collaboration with the CEA and PSA, April 2003
- [33] N. Rapin, *Validation de spécification à base d'automates par des techniques de dépliages et d'exécution symbolique*, Phd Thesis, University of Evry (France), in collaboration with the CEA and Ligeron S.A., July 2004
- [34] International Union of Telecommunications, *Langage de programmation – Langage de Description et de Spécification du CCITT – Norme 34*, Recommandation UIT T Z.100, March 1993
- [35] OMG, *Unified Modelling Language 2.0*, OMG, Rapport formel/2003-04-01, January 2003
- [36] P. Wolper, P.Godefroid, *Partial-Order Methods for Temporal Verification*, procs. of CONCNU'93, pp.233-246, Hildesheim (Belgium), August 1993
- [37] S. Yovine, *Kronos: A verification tool for real time systems*, Springer International Journal of Software Tools for Technology Transfer, vol. 1 n.1/2, October 1997