URL: http://www.elsevier.nl/locate/entcs/volume82.html 10 pages

Compositional Generation of MC/DC Integration Test Suites

Alexander Pretschner [1]

Institut für Informatik, Technische Universität München, Germany

Abstract

We present a method for automatically generating tests for reactive systems specified by concurrently executing extended finite state machines. The generated test suites satisfy the modified condition/decision coverage criterion at unit and integration levels. The generation of MC/DC suites for eager first-order functional programs is subsumed. An industrial chip card case study illustrates the approach.

1 Introduction

The main difficulty in testing is to choose "good" test cases. This is because the quality of test cases is bound to a particular application. A criterion for what constitutes a "good" test case serves as test case specification—i.e., the "property" to be tested—, as a stopping criterion for the testing process, and as a metrics for assessing the quality of a test suite [8]. In the domain of testing, coverage criteria enjoy some popularity. While nearly everybody agrees that they should only be used as a complement to functional, i.e., domain- and problem-specific testing, they exhibit the utterly useful characteristics of being domain- and application-independent. Whether or not coverage criteria are suitable to assess the quality of a test suite is not the subject of this paper.

We present a method for computing test sequences that satisfy a control flow oriented structural coverage criterion called the modified decision/condition coverage (MC/DC). MC/DC is recommended as a complement to functional tests by the DO-178B standard used in the aircraft industry.

Coverage criteria are usually defined on the grounds of units. Examples for units include functions in C, or methods in Java. Since functions usually contain some implicit assumptions on their inputs and the current data state, unit-based test suites may well contain test cases that can never be executed by the integrated system. Our approach not only generates test suites that

[1] Supported by the DFG (Be 1055/7-3); Fax +49 8928917307; Email pretschn@in.tum.de

satisfy the criterion on a per-unit basis, but also for arbitrary compositions of units, including the entire system. This means that the generated test suites are executable by the system, and they satisfy MC/DC for each unit. We generate integration test suites on the grounds of previously generated unit tests.

The language under consideration is that of the CASE tool AutoFocus [1]. Reactive systems are specified by hierarchic concurrently executing extended finite state machines (EFSMs), i.e., finite state machines with a local data space. We only consider deterministic systems. Guards and assignments of transitions are specified in a simple functional language. Test cases are generated by means of symbolic execution. The idea is to first generate a test suite for each transition, i.e., each pair of guard and assignment. This yields source and destination states for this transition. We monitor which function definitions have not been entirely covered and try to find additional test cases for those transitions that access these function definitions.

Using directed search [3], we then try to find a trace of the component (the EFSM) the transition belongs to. If such a trace cannot be found, because the computation is too complex or the state is unreachable, then a different test suite for the transition is generated and the process is iterated. Once test cases for transitions have been turned into test cases for EFSMs, we try to turn these into test cases of composed systems. This is, again, achieved by using directed search.

Clearly, compositional test case generation at the level of coverage criteria is just one application of the overall scheme. There are no objections to using it for alternative test case specifications. Furthermore, if incremental development is understood as adding functionality in form of new components, then test cases for an earlier increment can be used as a basis for test cases for later increments. Model-based testing in the context of incremental system development is discussed in [3].

Test cases are used for both validating the model and verifying the respective implementation. In the first case, outputs must be checked manually due to a lack of a formalized specification—the model is the specification (the existence of a further formalized specification obviously only shifts the problem but does not solve it). In the second case, the model's output may serve as reference output for the implementation. Clearly, this requires bridging the gap between the different levels of abstraction. This is a difficult problem. In our case study, however, it turns out to be solvable.

We illustrate the approach by a chip card application, the WAP identity module (WIM [6]), taken from a recent study [2]. In cellular phones, it is used for card holder verification, for cryptographic operations such as computing digital signatures, and for the security related parts of the handshake between mobile equipment and some server. Concurrent EFSMs were used for functional decomposition of the system. The general ideas, however, are expected to carry over to actually distributed systems.

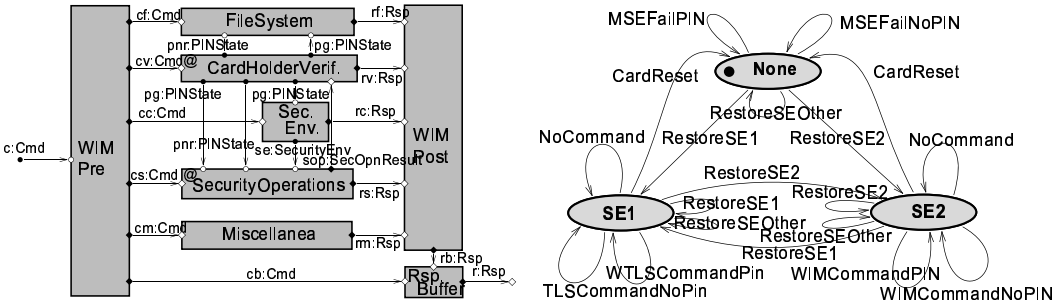

Fig. 1. WIM system structure; Security Environments EFSM

2 AutoFocus model of the WIM

In AutoFocus [1], systems are specified by hierarchic networks of concurrently executing clock-synchronously communicating EFSMs. Communication takes place over typed channels represented by arrows. Typed local variables can be assigned to components. Fig. 1, left, shows a system structure diagram of the WIM. Commands enter the system at the left hand side and are dispatched to the functional block that is responsable of taking care of them. Functional blocks are defined for the file system, for card holder verification with PINs and PUKs operations, for security environments—data structures for security-related and transport-level commands—, for security operations—enciphering, deciphering, computing and verifying digital signatures, computing cryptographic checksums—, and for miscellaneous other commands such as generating random numbers. Responses of the functional blocks are merged, and stored, if necessary.

Each bottom level component is associated with an EFSM. Fig. 1, right, shows an example for the state machine of component `SecurityEnvironment`. It handles the two security environments (storing keys and other cryptographic parameters), one for transport level security and one for signature related operations. Control states are depicted as ovals, transitions as arrows.

```
c?Cmd : not((is_PSOComputeDigSig(Cmd) || is_CardReset(Cmd))) :
pinNRState = pinNRTransition(Cmd,pinNRState) :
r!pinNROutput(Cmd,pinNRState); s!pinNRTransition(Cmd,pinNRState)
```

Fig. 2. Example Transition

Each transition is defined as a tuple consisting of pattern matching condition for the input channels, a guard ranging over input and component local variables, an assignment to local variables once the transition has fired, and output statements. Fig. 2 shows an example for a transition of a subcomponent of `CardholderVerification`, namely that taking care of the PIN that is used for digital signatures etc. The guard constrains nothing but the input; one local variable (`pinNRState` is updated by a function `pinNRTransition`). `pinNRState` is of a complex type `tCHVState` the handling of which turns out to be coped with more conveniently by referring to an explicit function rather

```
pinNRTransition: tCommand -> tCHVState -> tCHVState;
fun pinNRTransition(verifyCheck(pinNRRef), S) = S
  | pinNRTransition(verify(pinNRRef,P), cs(S,Pi,0,Pu,Puc)) =
      cs(unver,Pi,0,Pu,Puc)
  | pinNRTransition(verify(pinNRRef,P), cs(S,Pi,N,Pu,Puc)) =
      (if (P == Pi) then cs(verif,Pi,maxPinCtr,Pu,Puc)
                    else cs(unver,Pi,N-1,Pu,Puc) fi)
  | ...
```

Fig. 3. Functional definitions (variables start with capitals)

than accessing its components at the level of transitions.

Functions (as well as guards and assignments) are defined in a simple first-order functional language. A part of the definition of function `pinNRTransition` is given in Fig. 3. Roughly, it specifies that (1) the PIN's state is not changed if the command just checks the verification status, that its status is (2) set to "unverified" if the respective retry counter has reached 0, and (3) that (a) its state is set to "verified" and the retry counter is reset if the correct PIN was entered or (b) is set to "unverified" by decrementing the retry counter if a wrong PIN was entered.

Test case generation is then done by translating the system into a Constraint Logic Programming (CLP) language and executing this program. The execution is symbolic since unlike in approaches like explicit-state model checking, we may compute with sets of values that are dynamically created during execution. For instance, if a guard is specified by $i \neq v$ for input channel i and some value v, then we do not enumerate all possible instantiations for v but rather compute with two constraints, $i = v$ and $i \neq v$. At the end of the generation procedure, test sequences must hence be instantiated; this can be done at random or, if types are ordered, w.r.t. limit analysis.

3 Generation of MC/DC test suites

This section starts by defining MC/DC at the level of propositional formulae. It is shown how MC/DC test cases for functional expressions, function definitions, transitions, atomic EFSMs, and composed EFSMs are generated.

Propositional formulae The idea behind MC/DC is as follows. For each condition occurring in a program, a test suite is required that ensures that for each atom in the condition there exist two test cases that yield different results when independently toggling the atom under consideration. Such test suites may not always exist; just consider tautologies or conditions with one literal being directly dependent on another. If an MC/DC test suite exists, then there are $n + 1$ test cases for each condition that consists of n literals. We illustrate this by the formula $F = \neg a \vee (\neg b \wedge c)$. The set of possible evaluations is given by $\{1 : 000 - 1, 2 : 001 - 1, 3 : 010 - 1, 4 : 011 - 1, 5 : 100 - 0, 6 : 101 - 1, 7 : 110 - 0, 8 : 111 - 0\}$ where the running number labels a test case,

the triple denotes the values of a, b, and c, respectively, and the value after the hyphen is the value of F when evaluated.

For this formula, there are two test suites that satisfy MC/DC, namely the one consisting of test cases (variable assignments) $S_1 = \{1, 5, 6, 8\}$, and S_2, consisting of $\{4, 5, 6, 8\}$. For S_1, we have the following: Cases 1 and 5 toggle the value of a by changing the formula's evaluation; $(5, 6)$ toggle c, and $(6, 8)$ toggle b. For S_2, $(4, 8)$ toggle a instead of $(1, 5)$ in S_1. An algorithm for computing such test suites is straightforward and thus omitted for the sake of brevity. In the following, $\pi_{MC/DC}$ denotes the transformation that takes a propositional formula and picks one set of valuations that satisfy MC/DC.

Functional expressions In order to compute a test suite for a functional expression—the RHS of a function definition, a guard, or an assignment—we fix one conditional $ite(C, T, E)$. The basic idea is that we wrap this conditional into another one that enforces a valuation of the atoms in C such that the requirements for one of the test cases in an MC/DC test suite are fulfilled.

In the following, let $\bot$ denote the Boolean value *false* and $\top$ denote *true*. In a first step, the atoms $A_1, \ldots, A_n$ of C are abstracted into symbolic names, $S_1, \ldots, S_n$, resulting in an abstract condition C'. For instance,

$$e = f(X, ite(\neg g(X) \vee Y < 7, 123, 456)) \tag{1}$$

becomes $f(X, ite(C', 123, 456))$ with $C' = \neg S_1 \vee S_2$ and the substitution $\sigma = \{S_1 \mapsto g(X), S_2 \mapsto Y < 7\}$. We then compute an MC/DC suite for C', $\pi_{MC/DC}(C')$ as described above. In this case, there is just one, namely $\pi_{MC/DC}(e) = \{S_1 = \bot \wedge S_2 = \bot, S_1 = \top \wedge S_2 = \bot, S_1 = \bot \wedge S_2 = \top\}$. For test case generation, we then use the set $M = \{f(X, ite(\sigma(m), ite(C, 123, 456), fail)) : m \in \pi_{MC/DC}(C')\}$ where $\sigma(x)$ denotes the application of substitution σ to x and *fail* denotes a special value indicating that the computation did not succeed. In the following, we will refer to the computation of M for an expression e by $\mu'(e)$. μ' computes all suites if more than one exists (i.e., a set of sets). $\mu(e)$ picks one random element of $\mu'(e)$ (i.e., a set—a set of test cases).

Doing the above independently for each condition in a functional expression, we get such a set of expressions for each condition. Since we are equipped with a test case generator that by symbolically executing the expression is able to find all possible valuations for all variables occurring in an expression like the above one, we can use it to generate a test suite that contains values for X and Y such that the resulting test suite satisfies MC/DC.

Function definitions In the functional language of AutoFocus, functions are defined as follows. Let $\vec{a_i}$ denote the tuple $a_{i1}, \ldots, a_{in}$. f of arity $n \geq 1$ is defined by m equations $f(\vec{a_1}) = rhs_1, f(\vec{a_2}) = rhs_2, \ldots, f(\vec{a_m}) = rhs_m$. Assume that d_i for $1 \leq i \leq m$ denotes the $i - th$ definition. The intuitive meaning of pattern matching with the function's actual parameters is that the first pattern that matches is used, and the other ones are ignored. This

motivates rewriting the definitions into one single function $f(\vec{A}) = \varphi(d_1)$ via a function φ for $1 \leq i \leq m$:

$$\varphi(d_i) = ite\Big(unif(\vec{a_i}, \vec{A}), (mgu(\vec{a_i}, \vec{A}))(rhs_i), \varphi(d_{i+1})\Big) \tag{2}$$

with $\vec{A} = A_1, \ldots, A_n$ being a tuple of fresh variables and $\varphi(d_{m+1}) = fail$. At runtime, the A_i are bound to the actual parameters of f. $unif(t_1, t_2)$ decides whether or not t_1 and t_2 are unifiable; $\lambda s.mgu(t_1, t_2)(s)$ computes the most general unifier of the two arguments and applies it to term s.[2] Clearly, this choice is somewhat arbitrary. One may well decide to use the following definition instead,

$$\varphi(d_i) = ite\Big(\bigwedge_{j=1}^{n} unif(a_{ij}, A_j), (mgu(\vec{a_i}, \vec{A}))(rhs_i), \varphi(d_{i+1})\Big), \tag{3}$$

which is equivalent but obviously does impact the number of necessary MC/DC test cases. The problem is a result of the definition of *unif* which can be seen as an atomic proposition, a conjunction of propositions over all arguments of the function, or as a conjunction of propositions over all positions of all arguments. This example is typical for coverage criteria defined on the grounds of units: If *unif* is seen as a black box function as in Eq. 2, the resulting test cases are less numerous than if the definition of Eq. 3 or the position-wise definition are taken into account.

Applying the above transformation μ to $\varphi(d_1)$, we get a set of functions, $\mu(\varphi(d_1))$. By means of symbolic execution, they yield an MC/DC test suite for the definitions of a function. Functions may well contain implicit assumptions on their actual parameters; these may not be reflected by the test cases.

Transitions Consider a component with $p \geq 1$ input channels, $q \geq 1$ variables, and $r \geq 1$ output channels ($\{p, q, r\} \cap \{0\} \neq \emptyset$ is handled similarly). Each transition of the associated EFSM is of the form $\vec{I} = \vec{\iota} : g : \vec{V'} = \vec{v} : \vec{O} = \vec{o}$ with the following intuitive meaning. Provided the current input values, $\vec{I} = I_1, \ldots, I_p$ match given values $\vec{\iota} = \iota_1, \ldots, \iota_p$, it is checked whether or not guard g, ranging over $\vec{I}$ and the current values of the local variables $\vec{V} = V_1, \ldots, V_q$, holds. If this is the case, then local variables, $\vec{V'}$, as well as output channels, $\vec{O} = O_1, \ldots, O_r$ are assigned new values, $\vec{v}$ and $\vec{o}$. If the pattern matching condition or the guard do not hold, then the transition is not enabled. Similar to the above, we rewrite each transition

$$t(\vec{I}, \vec{V}, \vec{V'}, \vec{O}) = ite\Big(unif(\vec{\iota}, \vec{I}), (mgu(\vec{\iota}, \vec{I}))(ite(g, a, st(\vec{I}, \vec{V}))), st(\vec{I})\Big), \tag{4}$$

where assignment $a = (\vec{V'} = \vec{v} : \vec{O} = \vec{o})$ assigns new values to the output channels and updates the local variables. Guard g and assignment a may contain arbitrary functional expressions; since a assigns more than one value, we assume some function that defines sequential composition to be given.

[2] Replacing matching by unification is part of the translation for symbolic execution.

Function *st* stores the current values of its arguments and then fails. The rationale for this is that if pattern matching condition or guard cannot be satisfied, this situation must be taken care of by another transition that emanates from the same control state. For the corresponding values of $\vec{I}$ and $\vec{V}$, the transition cannot fire. Since the system is supposed to be input enabled, we try these values for all other transitions emanating from the same control state; one will be enabled for the corresponding values. Note that it is obviously necessary to adjust the translation μ for functional expressions by replacing *fail* by $st(\vec{I}, \vec{V})$ when $\mu(g)$ and $\mu(a)$ are computed.

We use the above definitions to compute test suites that satisfy MC/DC for each single transition. Source and destination control states for each transition are known. The resulting test suite constrains the source data state, values at input channels, and yields constraints for updated local variables and output channels. Clearly, the set of source states defined by the test suite may be locally or globally unreachable, i.e., it may be unreachable in the component under consideration or in the overall system.

Furthermore, in general it is the case that not all function definitions are covered w.r.t. the required criterion. That is to say, if MC/DC for all transitions is achieved, this does not necessarily mean that all function definitions are covered w.r.t. MC/DC. We cope with this in a manner similar to what we did with function *st* for transitions above: It is monitored which function definitions have not been executed. For these function definitions, alternative transitions are then tried in order to enforce their execution.

Single EFSMs Applying the above procedure to all transitions of an EFSM yields a set of source and destination (control and data) states. We need to find traces of the corresponding component that eventually lead to them. In order to do so, one can, up to a given length, enumerate all possible execution traces of the component and check whether or not the given states are contained. In principle, this is what happens when entire components are symbolically executed; efficiency is increased by (a) preventing sets of states from being visited more than once (in fact, specializations of previously visited sets of states are excluded) and (b) direct the search. This can be done by computing distances between the actual state and the state that is sought for, and heuristically choosing the next transition w.r.t. the minimal distance (often, it is beneficiary to rely on backwards search). This approach is described in detail in [3,4]. If a state turns out not to be reachable or the computation appears to be too costly, then one might try another set of MC/DC test cases for transitions, or check why this situation occurred.

Communicating EFSMs The traces of single components may well turn out not to be executable when the system is integrated. Again, the reason is that components usually contain implicit assumptions on their inputs.

In the simple clock-synchronous setting, one starts by trying whether or

not the computed sequences for one component are projections of the behavior of the composed system. In case they are, it is rather simple to compute sequences of the overall system by means of symbolic execution—the component's test sequence determines much of the system's behavior which renders the search space rather small.

In case they are not, we check whether or not each single state of the component's test sequence is reachable. That is to say, if the sequence is of the form $[\sigma_1, \sigma_2, \ldots, \sigma_n]$, we start by trying to find a sequence of the overall system such that the respective projection of its last state, Σ_1, equals σ_1 (in fact, it is advisable to also allow generalizations and specializations of σ_1). This process is iterated. Starting by Σ_1, we try to find a sequence the last state of which, Σ_2, is a specialization or generalization of σ_2. This must be done in a way that the constraints on all elements of the first sequence remain satisfiable. The process repeats from Σ_2, until Σ_n is found that specializes or generalizes σ_n. Clearly, elements of the sequences that lead to Σ_i may well lead to states that make it impossible to reach some σ_i; in this case, backtracking is needed. Searching is, as in the case of single components, rendered more efficient by state storage and directed search.

4 Evaluation

The above procedure was applied to the model of the WIM. The system uses a total of 210 function definitions for which 312 MC/DC test cases were generated. Tab. 1 shows the results for component-wise and system-wide test case generation. The second column shows the number of subcomponents for the hierarchical components `CardHolderVerification` and `SecurityOperations`. The third column shows the number of transitions (arrows in the EFSM) for each component. For the hierarchical ones, the sum is built over their subcomponents. The fourth column displays the number of MC/DC test cases that were generated for the components. The respective test cases are symbolic traces and not mere values. Finally, the fifth column shows the number of 1-step symbolic executions. That is to say, if the source state is not determined, the component can perform as many different symbolic steps as given by the respective number. For the hierarchical components, we give the sum of the number of executions of their subcomponents (Σ) as well as the number of steps if the component's subcomponents are connected one to the other (Π).

The test cases for components covered almost all function definitions, except for those cases that were known not to be applicable (the respective function definitions had been inserted for the sake of completeness). MC/DC coverage for the integrated model, including all transitions and function definitions, could be achieved by 407 test cases (this number does not take into account impossible cases that cannot be executed by the system). That this number is comparable to the number of test cases for the function definitions

Component	Subc.	Transitions	MC/DC	executions
WimPre	–	9	11	33
FileSystem	–	12	12	16
CardHolderVerification	6	Σ68	Π28	Σ187, Π2530
SecurityEnvironments	0	19	35	69
SecurityOperations	9	Σ51	Π41	Σ159, Π388
Misc	0	3	3	3
WimPost	0	5	32	6
ResponseBuffer	0	7	13	11
System	21	Σ174	407	$\Pi 94,244$

Table 1
MC/DC test cases

comes as no surprise since almost all interesting functionality is encoded in these functions. There is, however, redundancy in the test cases for we did not check whether or not one test case subsumes another. Removing redundant test cases is the subject of future work.

As a comparison, we generated about 60,000 test sequences [2] when not using MC/DC as test case specification. Out of these, roughly 1,500 were chosen to test the actual card. These test cases contain repetitions of commands (re-entering wrong PINs) that are not directly reflected by a structural criterion like MC/DC.

5 Related Work

Various approaches to test case generation on the grounds of explicit state transition diagrams are discussed and compared in [3]. Our approach differs in not explicitly building the state space but rather symbolically executing the system. This alleviates the task of coping with structural criteria since we do not have to explicitly built the labeled transition system and translate the coverage criterion accordingly.

[5] use a model checker for computing MC/DC test suites at the level of units. [7] use genetic algorithms for the generation of structural unit tests. Reactive systems—more concretely, different states of a unit—are not considered. In terms of our approach to directed search, the approaches are somewhat similar. In both cases, integration tests are not taken into account.

6 Conclusion

We have presented a method for generating MC/DC integration test suites from models specified with communicating EFSMs. The distinct feature of the approach is that we not only generate test suites for units (functions, transitions) but also for the entire system. The test suite for the entire system does, however, satisfy MC/DC at the level of units. Clearly, the compositional

approach to test case generation generalizes to test case specifications different from coverage criteria. The practical applicability of our approach was demonstrated along the lines of an industrial case study. Clearly, the more complex systems become, the more power is necessary to direct the search in order to generated test cases for composed systems. The use of constraints enables one to "help" the system to explicitly disregard certain parts of the state space. We consider this the key to graceful degradation of our approach.

Test cases were generated from a model. While we have shown for the smart card example that it is possible to bridge the gap between the different abstraction levels of model and actual system [2], the general problem persists.

Coverage criteria are syntactic in their nature and must be complemented by functional, domain-specific tests. In fact, we consider technologies like the presented one to be only a smart part of the solution to the problem of testing. The main difficulty is that in general engineers do not know what to test (which is, among others, one reason to use coverage criteria). Narrow domain-specific error classifications—and in turn, test case specifications—might be one approach to solving this problem.

Acknowledgment Jan Philipps built the model and engaged in many stimulating discussions.

References

[1] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In *Proc. FME'97*, LNCS 1313, pages 122 – 141, 1997.

[2] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. Submitted, 2003.

[3] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Incremental System Development. *The Journal of Systems and Software*, 2003. To appear.

[4] A. Pretschner and J. Philipps. Constraints for test case generation. Submitted to J. Theory and Practice of Logic Programming, 2002.

[5] S. Rayadurgan and M. Heimdahl. Coverage Based Test-Case Generation using Model Checkers. In *Proc. 8th Intl. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–93, 2001.

[6] WAP Forum. Wireless Identity Module. Part: Security. Wireless Application Protocol WAP-260-WIM-20010712-a, 2001.

[7] J. Wegener, K. Buhr, and H. Pohlheim. Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing. In *Proc. Genetic and Evolutionary Computation Conference*, 2002.

[8] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.