Completion of Test Models Based on Code Analysis

Michael Dierkes, Alain Faivre, Hélène Le Guen, Nicky Williams

Completion of Test Models Based on Code Analysis

Michael Dierkes[1], Alain Faivre[2], Hélène Le Guen[3], Nicky Williams[2]

Keywords: Model Based Testing, MC/DC, structural coverage.

I. Introduction

In industries developing highly critical embedded software, the use of formal analysis techniques as well as model based testing techniques is increasing thanks to the availability of new efficient tools and new versions of the relevant standards, like the DO-178C and its supplements in the case of the aerospace industry. Naturally the question arises in which way static analysis and model based testing can be combined in order to overcome the limitations that each approach is subjected to when used individually. The European project MBAT gives answers to this question, and in this work we show how a combined approach is applied.

Model Based Testing (MBT) is an approach to software testing in which handwritten tests are replaced by tests automatically generated from a test model. This has several advantages: a huge number of tests can be generated from a test model in a small amount of time, and in the case of requirement changes, the update of a test model is much less time consuming than the update of a database of individual test cases. By rising the abstraction level on which test engineers specify their tests, high productivity gains are possible, which enable companies to obtain very high quality products at reduced effort.

Test suites for critical software which must be certified at highest level are in general required to satisfy criteria concerning the code coverage, like for example the MC/DC coverage criterion. However, such code based quality criteria are difficult to deal with on the test model level, because implementation details like condition expressions may not be described in sufficient detail. On the other hand, by using formal code analysis tools, it is possible to generate test suites with MC/DC coverage automatically from the implementation code, but even if such test suites fulfill the MC/DC criterion formally, they often don't make sense to human engineers, and this methodology does not comply with some standards. Therefore, the combination of model based testing and formal code analysis is a promising approach, since the two techniques can complement one another.

In our approach, we propose to extend MBT using results obtained by formal code analysis in order to obtain tests which still make sense to human engineers but which also achieve the goal of MC/DC coverage for the implementation. In our study, we compare two different model based testing tools.

This article is structured as follows: in section II, we give a global presentation of our method. In section III we give an example of its application and in section IV we mention open problems. We conclude in section V.

[1] Rockwell Collins, 6, avenue Didier Daurat, 31701 Blagnac, France.
[2] CEA, LIST, Point Courrier n° 174, 91191 Gif sur Yvette, France.
[3] ALL4TEC, 6 rue Léonard de Vinci, BP 0119, 53001 Laval Cedex France.

II. Presentation of the method

We propose a semi-automated approach: first, a test suite is generated from a test model, and an analysis is performed of how this test suite covers the source code of the implementation. If decisions in the source code are found which are not completely covered by the test suite, the analysis suggests how certain test scenarios from the suite can be modified in order to cover these decisions. In our approach, the results of the analysis help the user to complete the test model in such a way that complementary test scenarios will be generated automatically from the test model. The modification of the test model is not done automatically, but by the user, in order to ensure the meaningfulness of the tests generated using the resulting model.

One alternative to our approach would be just to try generating new tests from the existing model in the hope that by chance, the coverage would improve. Another alternative would be to directly add the new test scenarios proposed by the analysis of the implementation to the existing suite in order to ensure complete MC/DC coverage.

In the current presentation of our approach, we only treat the addition of details to the model in order to complete coverage. However, any issues revealed by the coverage analysis must be carefully studied by the user in order to ensure that an uncovered item in the source code also is not the result of a fault or ambiguity in the code rather than an incomplete model. We only treat test inputs here but normally an oracle would be used to ensure that gaps in MC/DC coverage are not due to a fault implementation.

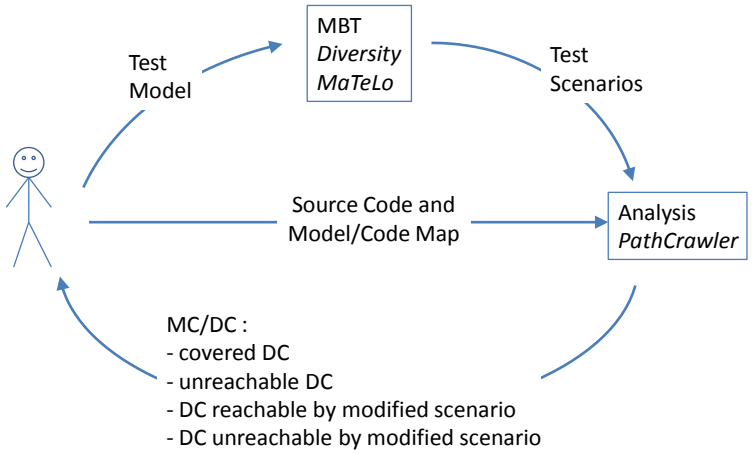

Figure 1: The MBAT method for MC/DC completion

In order to illustrate our approach, we show its application to a simple example.

III. Illustration of the method on an example

A. The example

The following requirements specify the microwave oven control logic.

- The setup mode is the initial mode.
- In the setup mode, if the user enters a time of *n* steps and pushes the start button, then the oven will cook for *n* steps unless the door is opened or the clear button is pressed.
- If the door is opened or the clear button is pressed while the oven is cooking, then the cooking is suspended.
- If the cooking is suspended, the door is closed and the start button is pressed, then cooking is resumed for the number of steps which remained before the suspension.

- If the cooking is suspended and the clear button is pressed, the oven goes into the setup mode.
- If the cooking is finished, the oven goes into the setup mode.
- The oven must never be cooking if the door is open.

B. Construction of the test model

We compare the use of two Model Based Testing tools to implement our method:

- MaTeLo (1) which focuses on usage models built from functional requirements.
- DIVERSITY (3), based on functional or design models at different abstract levels.

On the one hand, MaTeLo can generate test cases corresponding to typical uses of the SUT. On the other hand, DIVERSITY can generate test cases and an oracle from either an abstract model or a more concrete model including the details that are necessary to describe additional behaviors that must be tested in order to fulfill the MC/DC criterion.

1. Construction of a test model for MaTeLo

MaTeLo (Markov Test Logic) has been industrialized, developed and maintained by All4tec since 2003. Test engineers do not need to consider the implementation details, but just focus on the added value with the help of a test modeler. The only input needed to build this test model is the system requirements.

MaTeLo basically describes the usage model of the SUT (System Under Test) implemented for "Black Box Testing" in all xIL steps (MIL, SIL, PIL, HIL). A test model is based on an Extended Finite State Machine and Markov chains where stimulations and expected results are basically associated with transitions. Some facilities are provided in order to define the expected results, to link the test model with system requirements and also to automate test execution.

The MaTeLo model on Microwave oven is depicted in the Figure 2.

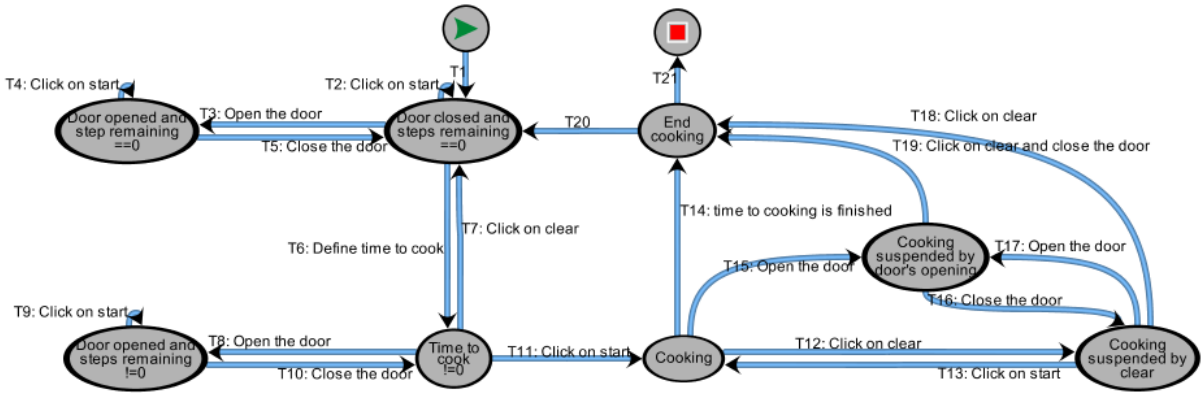

Figure 2: MaTeLoV1 model

2. Construction of a test model for DIVERSITY

The DIVERSITY toolset, developed by CEA LIST, is a validation and verification platform based on model analysis. Models may be described with the help of Stateflow-type languages describing potentially concurrent and communicating automata (Statemate, IF, UML statecharts ...). They can also be characterized using dataflow languages such as the one used in Matlab/Simulink.

Its tools first provide a symbolic simulation of the model, in order to give an early feedback on the model behaviours. The symbolic execution tree can highlight application-independent unexpected

behaviours such as livelocks, deadlocks, bad synchronisations between components, over-designing (parts of model never activated) or sub-designing (reachable states not handled).

At the end of the design process, once the real system is built, one needs to ensure that it conforms to its requirements or to its model. The concrete numerical test cases may be executed on the real system in order to verify the compliance of the implementation with the model.

A requirement model was written for the microwave example using the internal stateflow language of DIVERSITY. Figure 3 illustrates this model, which we will call DiversityV1.

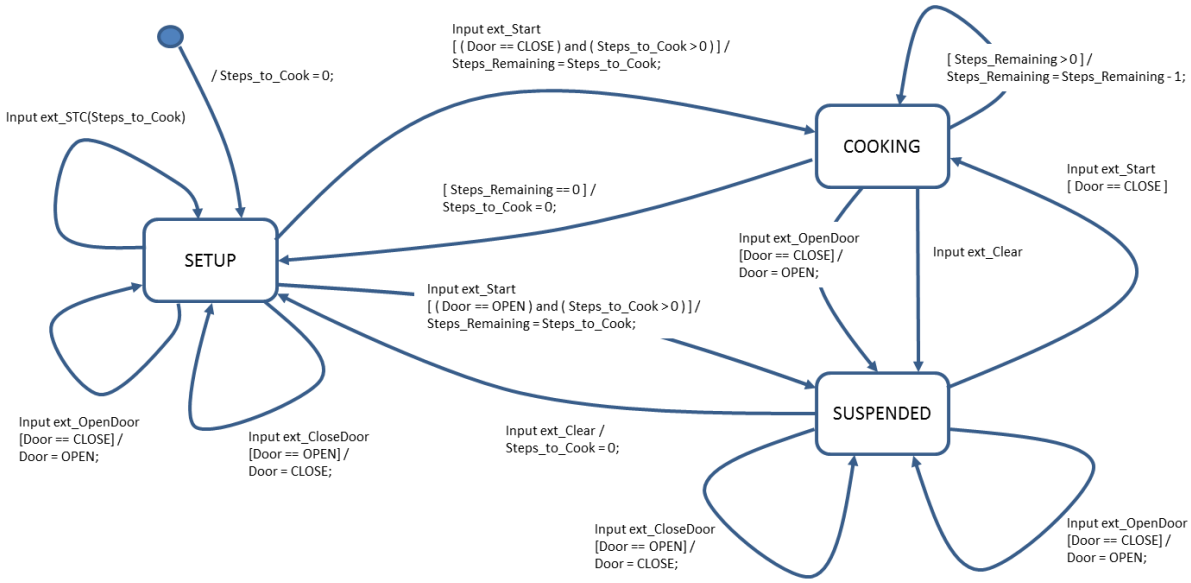

Figure 3: The diversityV1 model

C. Automatic model-based test generation

1. Automatic test generation using MaTeLo

MaTeLo can automatically generate the test-cases required to ensure good product quality. The user can choose between the following algorithms: test model arc coverage or Monte Carlo simulation according to a user-oriented profile.

In this situation, because MC/DC coverage is the objective, we have chosen to generate test cases in order to cover every transition of the MaTeLo model.

Step	Transition	Input	Output
0	T1	Data initialisation (door close, ...)	Not cooking
1	T6	Time_to_cook (25 steps)	Not cooking
4	T11	Click on start	Cooking
9	T12	Click on clear	Not cooking
11	T13	Click on start	Cooking
14	T15	Open the door	Not cooking
18	T19	Click on clear	Not cooking
21	T19	Close the door	Not cooking
	T20		
26	T6	Time_to_cook (7 steps)	Not cooking
28	T8	Open the door	Not cooking
32	T9	Click on start	Not cooking
36	T10	Close the door	Not cooking
41	T7	Click on clear	Not cooking

42	T6	Time_to_cook (13 steps)	Not cooking
47	T11	Click on start	Cooking
50	T12	Click on clear	Not cooking
51	T17	Open the door	Not cooking
55	T16	Close the door	Not cooking
58	T18	Click on clear	Not cooking
	T20		
60	T3	Open the door	Not cooking
65	T4	Click on start	Not cooking
67	T5	Close the door	Not cooking
69	T2	Push start	Not cooking
73	T6	Time to cook (16 steps)	Not cooking
75	T11	Click on start	Cooking
91	T14		Not cooking
	T21		

Figure 4: MaTeLoV1 test scenario

In the previous scenario, only the steps with an input or an output are considered. The column transition shows the path on the model. Only the property of cooking is set in output, but in a real test case (use to find bugs) other system output should considered.

2. Automatic test generation using DIVERSITY

The test-case generator of DIVERSITY generates test-cases from models according to several coverage criteria, which may be structural criteria such as transition coverage or more sophisticated ones. In this study case, we use the « inclusion coverage criteria » which, if it succeeds, characterizes all symbolic behaviors of the system. Then DIVERSITY associates a numerical test case to each symbolic behavior. For our example, DIVERSITY generates 12 test cases. Scenarios number 4 and 10 are shown in the following Figure 5.

D. Analysis

In this work, the code analysis is performed by the PathCrawler tool. PathCrawler was actually designed for the automatic generation of test-case inputs to ensure structural coverage of the source code, such as MC/DC coverage. It is based on an analysis of the source code of the implementation, using a method which is often called “concolic” or “Dynamic Symbolic Execution” in the literature. CEA has been developing and improving the PathCrawler tool since 2003 and a version, with further documentation, is freely available as a web-service, for evaluation or use as a teaching support, at (2).

As we explain in the following steps, we use PathCrawler to successively analyse:

- the scenarios generated by MBT,
- possible tests representing one transition from any state and
- small modifications of the scenarios generated by MBT.

Indeed, the test cases generated by MBT are called “scenarios” because each test input is associated with a symbolic time. Our method is based on the paradigm of synchronous reactive systems which supposes that the top-level of the implementation code first calls some initialization function and then enters an infinite loop. In each iteration of the loop, input signals are polled and then one or more

```
------- SCENARIO NUMBER 4 ----------------------------------

  Initialization values:
    Steps_to_Cook = 0
    Door = CLOSE

  INPUT -----> ext_STC( 3 )
  OUTPUT ----> ext_State( SETUP )
  INPUT -----> ext_Start()
  OUTPUT ----> ext_State( COOKING )
  INPUT -----> ext_OpenDoor()
  OUTPUT ----> ext_State( SUSPENDED )
  INPUT -----> ext_CloseDoor()
  OUTPUT ----> ext_State( SUSPENDED )
  INPUT -----> ext_OpenDoor()
  OUTPUT ----> ext_State( SUSPENDED )

...

------- SCENARIO NUMBER 10 ---------------------------------

  Initialization values:
    Steps_to_Cook = 0
    Door = OPEN

  INPUT -----> ext_STC( 3 )
  OUTPUT ----> ext_State( SETUP )
  INPUT -----> ext_Start()
  OUTPUT ----> ext_State( SUSPENDED )
```

Figure 5: Two examples of the DiversityV1 scenarios

functions, including the tested function, are called. The symbolic time in the test scenarios is related to the number of iterations of this loop. Each call of the tested function effects a transition from one state of the model to another (or to the same state as before in the case of transitions represented as loops in the test models).

1. Concretisation of the model-based test scenarios

In order to run them on the implementation, the model-based scenarios must first be concretised by identifying and matching the external input events in the model and the inputs sampled by the code. The test models are based on an abstraction of the inputs to the SUT as events and in order to run the model-based tests, each input event in the model must be mapped to an input of the tested code.

This is standard practice in MBT. However, in our approach the differences in the semantics of events at the model level and of input variables of the tested code must also be made explicit. Events in models are transient, lasting only one “tick” of symbolic time whereas changes to input variables may be remanent (e.g. in our example, once the door is opened, it stays open) or also be reset after one “tick” of symbolic time (e.g. pressing start in our example).

2. Analysis of the coverage of the model-based tests

This step supposes that the implementation respects the reactive systems paradigm mentioned above and takes the form of a top-level function which calls the tested function in a loop, polling the inputs at the start of each iteration. The user may need to encapsulate the tested function by hand in a simplified top-level function with this form.

PathCrawler is first configured to run on this top-level function in order to measure the MC/DC coverage of the source code of the tested function obtained by the MBT scenarios. For each decision, the coverage of the different conditions by the test scenarios is reviewed.

```
MCDC COVERAGE REPORT

FILE: microwave.c
FUNCTION: test_me

Fully covered decisions:

…

Unreached decisions:
None

Partially covered decisions:

Decision 'microwave.c':55:
Fully covered conditions:
Condition 54:
Coverage f(AND(f(54))): Test case existing(SCENARIO NUMBER 4)
Coverage t(AND(t(54),t(54b))): Test case existing(SCENARIO NUMBER 5)
Partially covered conditions:
Condition 54b:
Coverage t(AND(t(54),t(54b))): Test case existing(SCENARIO NUMBER 5)
Coverage f(AND(t(54),f(54b))): MISSING
Uncovered conditions:
None

Decision 'microwave.c':48:
Fully covered conditions:
None
Partially covered conditions:
Condition 48:
Coverage f(48): Test case existing(SCENARIO NUMBER 4)
Coverage t(48): MISSING
Uncovered conditions:
None

Decision 'microwave.c':42:
Fully covered conditions:
None
Partially covered conditions:
Condition 42:
Coverage t(42): Test case existing(SCENARIO NUMBER 5)
Coverage f(42): MISSING
Uncovered conditions:
None
```

Figure 6: Coverage results for DiversityV1

Figure 6 shows an extract from the coverage report for the DiversityV1 scenarios. The decisions and their conditions are referenced by their line number in the source code, with indices b,c,… in case of several conditions on the same line. The FALSE branch of the 1st condition on line 54 is denoted `f(54)` and the TRUE branch of the 2nd condition on the same line is denoted `t(54b)`. The TRUE decision of the logical AND of the two conditions on line 54 is denoted `t(AND(t(54),t(54b))`. Figure 6 shows that all decisions are at least partially covered by the DiversityV1 scenarios but they do not cover the following decision/condition combinations (DC):

DC1. The combination of the TRUE branch of the 1st condition and the FALSE branch of the 2nd condition of the AND decision on line 54
DC2. The TRUE branch of the simple decision on line 48
DC3. The FALSE branch of the simple decision on line 42

The coverage report for the MaTeLoV1 scenario is similar except that only DC1 and DC3 remain uncovered.

Although the coverage report is expressed in terms of the source code of the implementation, the user does not use this information at this point in our approach. Instead, the user notes that the coverage is incomplete and so the analysis must be continued, as we now describe.

3. Analysis of the feasibility of the coverage gaps

In this step, possible problems at the source-code level are identified by checking whether the coverage of certain items is completely infeasible.

PathCrawler is run just on the tested function of the implementation and configured so that the tool constructs new test inputs to complete the MC/DC coverage obtained by the MBT scenarios. These tests represent one transition of the system from any theoretically possible state. They are not meant to be added to the MBT test suite because they may not be "realistic" and they do not contain any information on expected output values: all that interests the user at this point is that such tests can be found. The point is that if it can be demonstrated that improving the coverage of a certain decision is always infeasible (even when not restricted to "realistic" tests) then the source of the problem cannot be the test model but must reside in the implementation code.

If possible, the code should be corrected before continuing. If it is not possible to remove a redundant condition, then the results of the analysis at this point can be used to explain and justify an inevitable gap in MC/DC coverage.

In our example, the result of this step for both the MaTeLoV1 and DiversityV1 scenarios is a new MC/DC coverage report in which DC3 remains uncovered. At this point, the user does take into account the source code lines referenced in the report. Inspection of the source code reveals that the corresponding branch is indeed unreachable, because it is identical to a previous condition (see Figure 7).

```
if (state->steps_remaining <= 0) {
...
} else {
if (...) {
...
} else {
if (state->steps_remaining > 0) { /* line 42 */
```

Figure 7: Unreachability of the FALSE branch of the condition at line 42

4. Modification of scenario inputs to fill coverage gaps

In this next step, PathCrawler searches for new test scenario inputs which increase coverage and are obtained by making slight changes to the test scenario inputs generated by MBT. Once again, the point is not to add these modified scenarios to the MBT test suite but to give the user an example of how the MC/DC coverage could be increased.

PathCrawler is again run just on the tested function of the implementation but this time its search for test inputs to complete coverage is restricted to those tests which start either from a system state already reached by an MBT scenario or from a state which would have been reached by changing the arguments of certain inputs in an MBT scenario. The tests obtained in this way, or the demonstration that no such tests can be constructed, can indicate what is missing in the test model.

In our example, in the case of the MaTeLoV1 scenario, PathCrawler constructs a scenario to cover DC1 by pressing start after the 14th step (and then ending the scenario). Note that PathCrawler does not (cannot) predict the expected output corresponding to the changed input.

In the case of the DiversityV1 scenarios,

- PathCrawler constructs a scenario to cover DC1 by changing the inputs of SCENARIO NUMBER 4 (shown in Figure 5) so that START is pressed at the 4th step (and then ending the scenario).
- PathCrawler constructs a scenario to cover DC2 by changing the inputs of SCENARIO NUMBER 10 (shown in Figure 5) so that the value of the Steps To Cook input in the 1st step is 0.

E. Completion of the test model

Using the analysis results from step D.4 above, the user decides how to refine the test model. Automatic MBT is run on the new model and the resulting scenarios are analyzed to ensure that MC/DC coverage is now complete (except for infeasible condition combinations).

Inspection of the MaTeLoV1 model and the MBT scenario shows at the 14th step cooking is suspended by the door being opened. DC1 therefore consists in pressing start at this point. A new transition is added to the model to cover and test this feature, resulting in the MaTeLoV2 model shown in Figure 8.

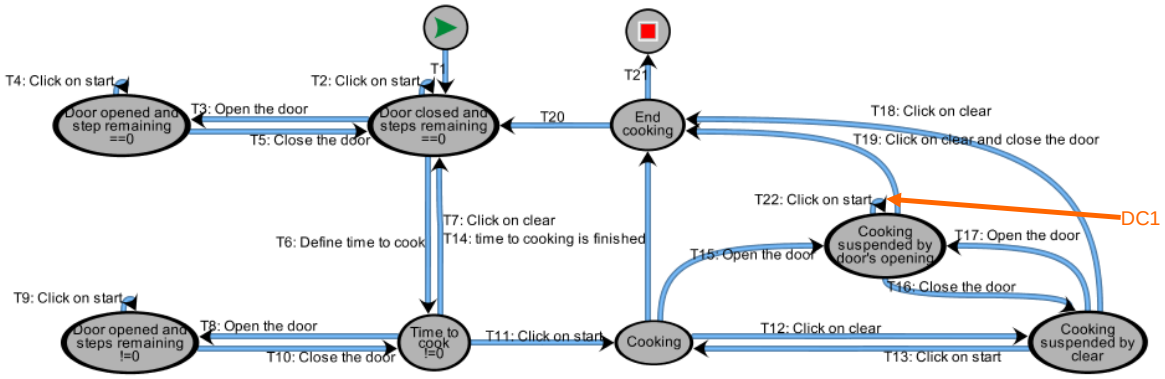

Figure 8: MaTeLoV2 model

Inspection of the DiversityV1 model and SCENARIO NUMBER 4 shows that pressing START when the oven is in the SUSPENDED state and the door is open has no effect in this model, but that this looping transition is not explicit.

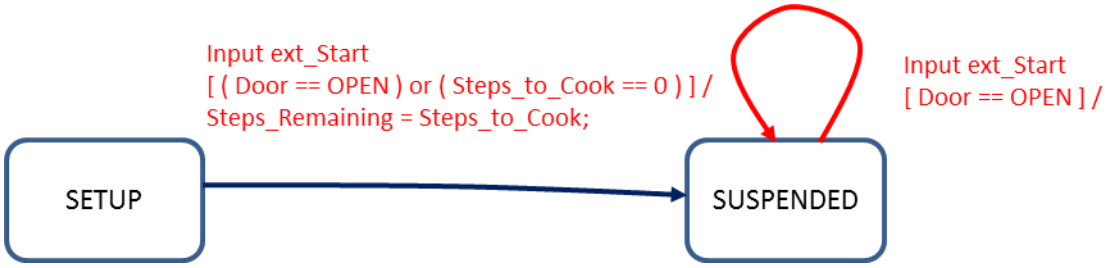

Figure 9: Transitions added to the DIVERSITY model

A similar solution which seems obvious for DC2 is to add a loop associated with Start to the state SETUP. But in this case, when executing the test cases generated by DIVERSITY, we find that the outputs of the source code do not match those predicted by the model. This is because the behaviour in this case was not made explicit in the specification and so if we change the DiversityV1 model in this way, then it creates a divergence between the implementation and the model. In fact, another possible modification is to change the condition associated with the transition from mode SETUP to

mode SUSPENDED with the door open and Steps_to_Cook set to 0 and this additional detail corresponds to the choice made in the implementation

Figure 9 shows changes made to the DiversityV1 model (one transition modified and one transition added) necessary to obtain the MC/DC coverage with DIVERSITY.

IV. Conclusion

We have presented a method for the automated generation of test suites with complete MC/DC coverage. The test generation is based on test models which are completed with the help of code analysis. The advantage of our approach is that it can produce a suite of realistic test cases with complete coverage under a code-based criterion.

Applying our approach to the microwave oven example has produced promising results. Through successive iterations, we obtained full MC/DC coverage of the C code corresponding to the implementation using MBT tools. This experience highlighted two reasons for incomplete MC/DC coverage of the implementation when using MBT, and how they can be revealed by code analysis.

Our future work will focus on how to apply our approach to industrial-scale applications. In our example, the proposed changes to the inputs of existing scenarios indicate to the user where the coverage is deficient in terms of the test model, thereby avoiding the need to study the implementation code and compare it to the model. We need to experiment with other examples to find whether this can usually be achieved with small changes to existing inputs. By constraining the search for new inputs to increase coverage to changes to existing inputs, we avoid combinatorial explosion in the search space but the treatment of very long or very numerous test scenarios may also pose challenges for the analysis tool. Finally, the initial premise that it is possible to complete a test model so as to achieve code coverage implies that the abstraction gap between the test model and the source code is not too great and that there are relatively few gaps in the coverage of the initial MBT scenarios.

V. Acknowledgements

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no 269335 (ARTEMIS project MBAT) (see Article II.9. of the JU Grant Agreement) and from the French government (5).

VI. References

(1) http://www.all4tec.net/wiki

(2) http://www.pathcrawler-online.com

(3) Symbolic execution techniques for test purpose definition, C. Gaston, P. Le Gall, N. Rapin, and A. Touil. *Testing of Communicating Systems*: 18th IFIP TC 6/WG 6.1 International Conference, TestCom 2006. Lecture Notes in Computer Science (New York, NY, USA), Springer, May 16-18 2006.

(4) Miller, S. P., Whalen, M. W., and Cofer, D. D. (2010). Software model checking takes off. Communications of the ACM, 53(2):58–64.

(5) http://www.mbat-artemis.eu