Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 1, No. 1, December 2015.

USING FORMAL METHODS FOR TEST CASE GENERATION ACCORDING TO TRANSITION-BASED COVERAGE CRITERIA

Ahmad A. Saifan[1] and Wafa Bani Mustafa[2]

Computer Information Systems Department, Faculty of IT, Yarmouk University, Irbid, Jordan.
ahmads@yu.edu.jo[1], w_banimustafa@yahoo.com[2]

(Received: 15-Sep.-2015, Revised: 01-Nov.-2015, Accepted: 08-Nov.-2015)

ABSTRACT

Formal methods play an important role in increasing the quality, reliability, robustness and effectiveness of the software. Also, the uses of formal methods, especially in safety-critical systems, help in the early detection of software errors and failures which will reduce the cost and effort involved in software testing.

The aim of this paper is to prove the role and effectiveness of formal specification for the cruise control system (CCS) as a case study. A CCS formal model is built using Perfect formal specification language, and its correctness is validated using the Perfect Developer toolset. We develop a software testing tool in order to generate test cases using three different algorithms. These test cases are evaluated to improve their coverage and effectiveness. The results show that random test case generation with full restriction algorithm is the best in its coverage results; the average of the path coverage is 77.78% and the average of the state coverage is 100%. Finally, our experimental results show that Perfect formal specification language is appropriate to specify CCS which is one of the most safety-critical software systems, so the process of detecting all future possible cases becomes easier.

KEYWORDS

Formal method, Perfect developer, Test case generation, Cruise control system (CCS).

1. INTRODUCTION

Generally, the role of software has become increasingly important and is being used in many critical applications, such as aircraft flight control systems, medical device systems and nuclear systems. Such systems are called safety-critical systems, the most important property of which is dependability, which reflects the extent to which users or customers trust the software system. It also reflects the degree of user confidence that the system will not fail when operated. A failure in these systems may cause significant damage, severe economic consequences or even loss of life. Thus, the reliability, safety and correctness of these systems are important issues, and testing them is a challenging task. Tim [23] shows that the formal method is a technique that is used to reduce the cost of testing safety-critical systems by examining the behaviour of these systems in the early stages of development. One way to reduce the cost and effort of software testing is to depend on formal methods, mainly for safety-critical software, because it provides many different techniques to describe precisely and accurately the system specifications [21].

In recent years, there has been an increased tendency to use formal specification and verification methods and tools, defined by Tim [23] as "the use of ideas and techniques from mathematics and formal logic to specify and reason about computing systems to increase design assurance

and eliminate defects". In addition, they are mainly used to provide a standard software development process.

Formal methods, tools and techniques have an important advantage in that they develop formal system specifications that facilitate and provide comprehensive system analysis, design and implementation as described by Mery and Singh [17]. Mery and Singh [17] adjusted the software development life cycle - from the requirements analysis to code generation - to develop safety critical systems using formal techniques. Formal methods provide an importantly disciplined approach for complex safety-critical systems with many different formal specification languages such as Z- [12], VDM [6], B [16], SCR [20] and Perfect Developer [7]. The formal method used in our work is Perfect Developer.

In this paper, we use the Perfect language as the formal language in order to model and test a real safety-critical system; a cruise control system (CCS). The CCS must be able to maintain the current speed of the car and accelerate it upon the request of the driver. It must also measure the current car speed and display the same to the driver.

In this paper, we present our framework on test case generation using a formal method. Application of the approach starts by building the formal model for the cruise control system using Perfect formal specification language. We verify the formal model in order to check completeness, consistency and validity of the model using the Perfect Developer toolset. Next, we build a C# tool to read the Perfect Developer formal model and produce the state-based specification graph or the FSM graph in order to automatically create test cases from the model according to a transition-based coverage criterion. We evaluate the coverage to test the system's performance and effectiveness. Moreover, we use test case reduction techniques in order to reduce the redundant test cases that will be generated by a set of testing algorithms.

2. RELATED WORK

2.1 Testing Critical Software Systems

According to You and Rayadurgam [25], the failure of software systems can cause significant damage to the software or its environment, severe economic consequences or even loss of life. There are many examples of safety-critical systems, such as aircraft flight control systems, medical device systems, nuclear systems and automobile cruise control systems.

The most important property of the safety-critical system is its dependability, which is shown by the extent to which users or customers trust the software system. Moreover, this will reflect the degree of user confidence in the system's ability not to fail when operated.

Many different formal specification languages and formal verification automated tools have been used to express and verify the formal specification of automobile cruise control systems. For example, Atlee and Gannon [3], Heitmeyer *et al*. [10] and Bultan and Heitmeyer [4] used the SCR-style (Software Cost Reduction) to specify and verify the safety property of the CCS. A formal cruise control B specification is discussed by Krupp *et al*. [13] in which an adaptive cruise control B model and RAVEN model checker are presented. The model is used to improve the confidence and understand ability of the system's behaviour. Another approach presented by Iliasov *et al*. [11], in which the system's dependability is characterized by rigorous design and fault tolerance, is represented by structuring the formal specification in an abstract way in the notion of an operation mode that depends on a state-based formalism approach called (Event-B) to refine the system modes. To check the effectiveness and safety requirements, Yasmeen *et al*. [24] conducted an experimental analysis which simulated a number of indications of safety critical systems. Mishra *et al*. [18] proposed a model-based testing auto-review tool, which was used to partially automate the process of verifying safety-critical systems. You and Rayadurgam [25] proposed a constrained random testing framework on a safety-critical embedded system. They used the constraints to narrow the possible test cases and cover most of the system's

behaviour. A similar approach to ours was discussed by Nilsson *et al.* [26]. They proposed correct-by-construction control software for Adaptive Cruise Control (ACC) system that is guaranteed to satisfy the formal specification in Linear Temporal Logic (LTL). They formalized the ACC using a hybrid dynamical system model with two modes: the no lead car and the lead car. Then, they constructed two controllers as a solution for the two modes, with the first solution based on continuous state space and the other based on a finite-state abstraction. The two controllers were tested by running a simulation in Simulink and on a vehicle simulation package called CarSim. However, our work depends on the Perfect language for the formalization which is easier than LTL, since the syntax in Perfect language is close to the programming languages. Moreover, according to Zhao [27], with the use of LTL "experiences show that specifications of even moderate-sized systems are too complex".

2.2 Testing Coverage Criteria

Coverage criteria on software systems can be defined as the set of conditions and rules that impose a set of test requirements on a software test. Ammann and Offutt [2] mentioned that test requirements in software testing are a specific set of elements of software artefacts that the software test cases must satisfy or cover.

Categories of coverage criteria include structural coverage, data-flow coverage, decision coverage, call graph coverage and transition-based coverage. In the framework for our cruise control system, a transition-based coverage criterion is used. For transition-based coverage, every precondition in the software's formal specification should be tested at least once so that each transition must be taken as a test requirement.

Several coverage criteria are used in the literature for testing systems. For example, Offutt *et al.* [20] introduced a technique for state-based specifications to generate test cases for cruise control systems. Their technique depends on an SCR formal specification model that represents the cruise control functionality and the coverage criteria that are: transition coverage, full predicate coverage, transition-pair coverage, complete sequence and structural coverage for state-based decision testing.

An experimental study conducted by Fraser and Gargantini [9] addressed the problem of test case generation, optimization and the performance of model checkers. In their study, they depended on explicit state model checkers that use a DFS algorithm. A cruise control system was studied, with its SRC formal specification being put into the model checker and test cases generated according to condition-based and structural coverage criteria.

Liu [14] used VDL-formal specification language notations in order to automatically generate a test prediction to analyze the results from their proposed decompositional approach for automatic test case generation. The results indicated that the researcher's approach was effective in terms of branch coverage, path coverage and statement coverage. In addition, his test case generation algorithm was effective in detecting the defective test cases for his system.

Liu and Nakajima [15] aimed to improve formal specification completeness and feasibility by introducing a new method that depends on verifying completeness and feasibility in the form of pre-conditions and post-conditions. This method uses single formal specification operations to choose the automated teller machine (ATM) system. SOFL formal specification language was used to build the ATM specifications. Appropriate test case generation criteria were used to build a question checklist from the ATM SOFL specifications in the test case generation process.

The proposed method improved the ATM operation's completeness and feasibility through its ability to formally define those characteristics. Furthermore, the generated test cases covered every aspect of the ATM defined specifications, so that it added the advantage of detecting errors in the system and reducing the cost of testing the system.

Another usage for SOFL formal specification language was presented in Chen's [6] research in which he addressed the problem of specification-based testing and test case generation for concurrent software systems.

Test cases were generated to cover the produced specification suggestions according to appropriate coverage criteria related to concurrent software systems. Each one of the generated test cases was executed several times in order to traverse program paths. The proposed approach was applied to an online shipping system and proved its usability for concurrent software systems.

Tian *et al*. [22] realized the problem of automatic test case generation from the pre- and post-conditional formal specifications (PROMELA formal specification language) by obtaining the benefits from connecting specification-based testing and the usage of model checkers (Spin model checker).

Nakatsugawa *et al*. [19] aimed to facilitate formal specification readability by discussing a new specification-based testing framework for interface specifications.

As with our approach, several others have been used to automatically generate test cases from a graph using graph coverage technique, such as: [28]-[30].

Gotlieb *et al.* [28] presented two different algorithms in order to automatically generate functional tests for synchronous executable BPEL processes. The first algorithm, STRUCRUNS, was used to generate test cases covering lengthy feasible paths up to a given length. The second algorithm, RANDOMRUNS, was derived from the desired number of test cases covering a random selection of feasible paths limited by a predetermined length.

Yan *et al.* [29] implemented a prototype that is used to automatically generate test data based on a constraint solving technique. In their approach, they transformed the procedure into a constraint system using static single assignment. Then, the constraint system was solved to check whether at least one feasible path through the selected point existed. Finally, test cases were generated corresponding to one of these paths.

Jehan *et al.* [30] proposed an approach that is used to automatically generate BPEL test cases that handle concurrent features. In their approach, they represented the BPEL program in an extended control graph. After that, they generated all the sequential test paths from XCFG before combining all the sequential test paths into concurrent test paths. Finally, they used the BoNus solver to solve the constraints of the test paths and then generate feasible test cases.

Utting and Legeard [31] in their book "Practical Model-based Testing: A Tools Approach" presented a commercial tool called LTG/UML which is a model-based testing tool that can be used to automatically generate test cases from a UML state machine using different coverage criteria. The tool has been demonstrated on examples and case studies from a variety of software domains, including embedded software and information systems.

3. PERFECT DEVELOPER

Many formal methods have been used to model safety-critical systems in the literature, such as Event-B Abrial [1], Z-formal specification [12], VDM [6], B-specifications [16], SCR specification [20] and a number of other formal languages. Most formal methods only involve the specification of the system; however, some continue the development process until a running code is obtained. However, according to Crocker [7], those formal methods take a long time to produce the running program. In this paper, Perfect Developer is used as a formal method and starts its process by describing the system, verifying a formal specification, refining it to the code within the same notation using a set of algorithms, verifying the refinement by checking its correctness and completeness and translating it into a set of high-level programming languages such as C++, Java and Ada.

Perfect Developer (PD) is an automated tool developed by Escher Technology to verify the Perfect specification software system. Perfect is an expressive language that describes system specification in an object-oriented model style using object-oriented terms and concepts such as classes, functions and constant variables. Perfect, like many object-oriented languages such as Java and C++, supports object concepts, such as encapsulation and inheritance, through object classes and message passing.

Perfect language has the ability to describe software system behaviours without any details of how the behaviour will be performed. It also includes certain design principles, including design by contract, which is the main structural rule or principle which Perfect depends on; it uses the contract to define the input-output relationships for class methods. The contract has two main parts; pre-conditions and post-conditions. The pre-conditions determine what must be true for the call feature, while the post-conditions determine what is guaranteed to be true at the termination of a successful Perfect feature as presented by Carter and Monahan [5].

An investigation was undertaken by Crocker and Carlton [8] to see if the automated reasoning using Perfect developer for the embedded software has the ability to provide the same degree of success in verification of a handwritten C code. The study made use of two small C programs with their specification annotations. As a result, they found that automated reasoning can discharge a very high proportion of verification conditions that arise from specification and software refinement. The number of test cases required was reduced.

4. METHODOLOGY

Our framework consists of seven phases shown in Figure 1. We first understand the cruise control requirements and then write the informal specification description for our system. The main reason for writing the informal specifications is to reflect its different states and transitions and the different conditions that enforce each transition in the system. In the second step, we develop the cruise control formal model, using Perfect formal specification language depending on the informal description of our system. The third step reflects the formal model verification using Perfect developer. After that, we manually extract the CCS formal model adjacency matrix, and then develop a C# tool to read our verified model and extract the different system paths and states in order to generate test cases using a set of proposed algorithms. The evaluation process for our generated test cases is discussed in phase seven. Moreover, we analyze the effectiveness of each algorithm used in the process of the test case generation. We will now discuss each of these steps in detail.

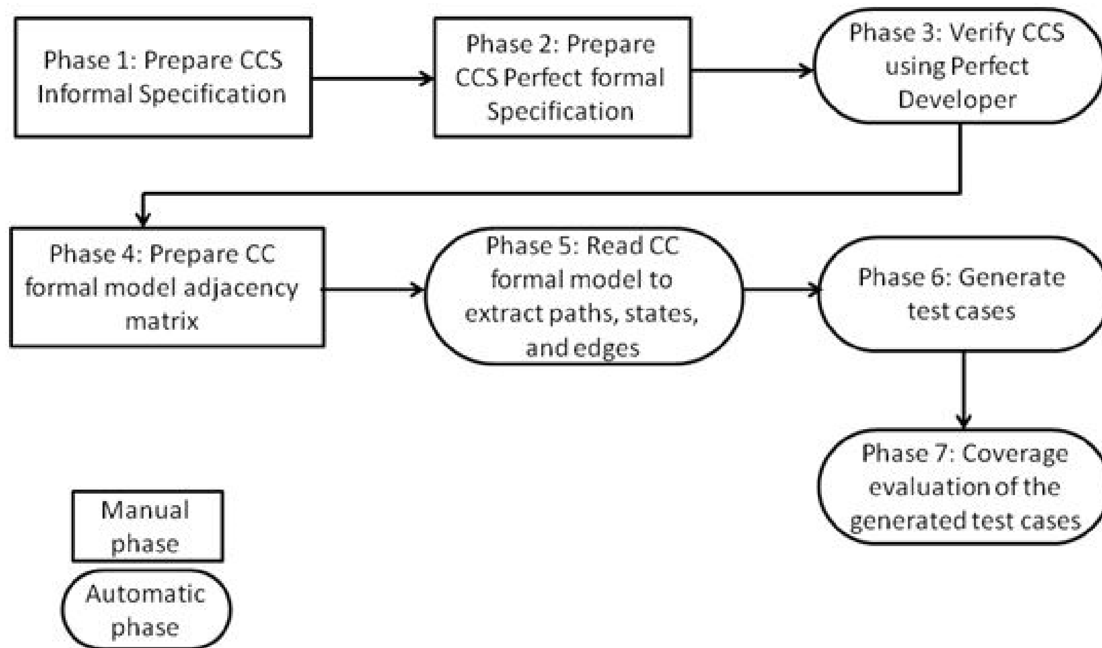

Figure 1. A framework for test case generation using the formal method.

4.1 Informal Specifications of Cruise Control System

The cruise control system (CCS) is a safety-critical system that can be described by a set of behaviours, modes, states and transition variable state quantifiers. Such quantifiers are the ignition switch, cruise control button, automobile speed and cruise control target speed.

CCS transition modes that occur between the system and the driver are started by the initial state when the system ignition is “OFF”. The current speed is at zero and the cruise state is “OFF”. When the driver switches the ignition to “ON”, then the CC state can be “ON” or “OFF” depending on whether the driver wants to switch on the CC button or not. When the CC is switched to “ON” and the driver starts driving, then the cruise state can be “active”, “cruising” or “standby”. The main role of the CCS is to maintain the automobile’s speed as close as possible to the target speed. The target speed is determined by the driver when the CCS is “ON” and the target speed can be increased or decreased by the driver.

The automobile’s actual speed should not exceed a restricted limit and should also not exceed the target speed, so that the driver has an allowance interval that defines how much the actual speed could deviate from the target speed. Where the difference between the two speeds is acceptable, the CC will maintain the current speed. If the driver’s current speed exceeds the target speed, then the CCS will take measures to maintain the target speed.

The CCS states can be affected by a set of factors like: pressing the brake, accelerator and the “ON” / “OFF” button for the CC. In addition, various failures could affect the CCS states, such as a low battery; some of these faults are handled by returning the control to the driver to take certain measures. As a result, the CC could be activated (the state will be changed from “standby” to “active”) or the driver could drive without activating the CC. Figure 2 describes the CCS finite state machine diagram.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 1, No. 1, December 2015.

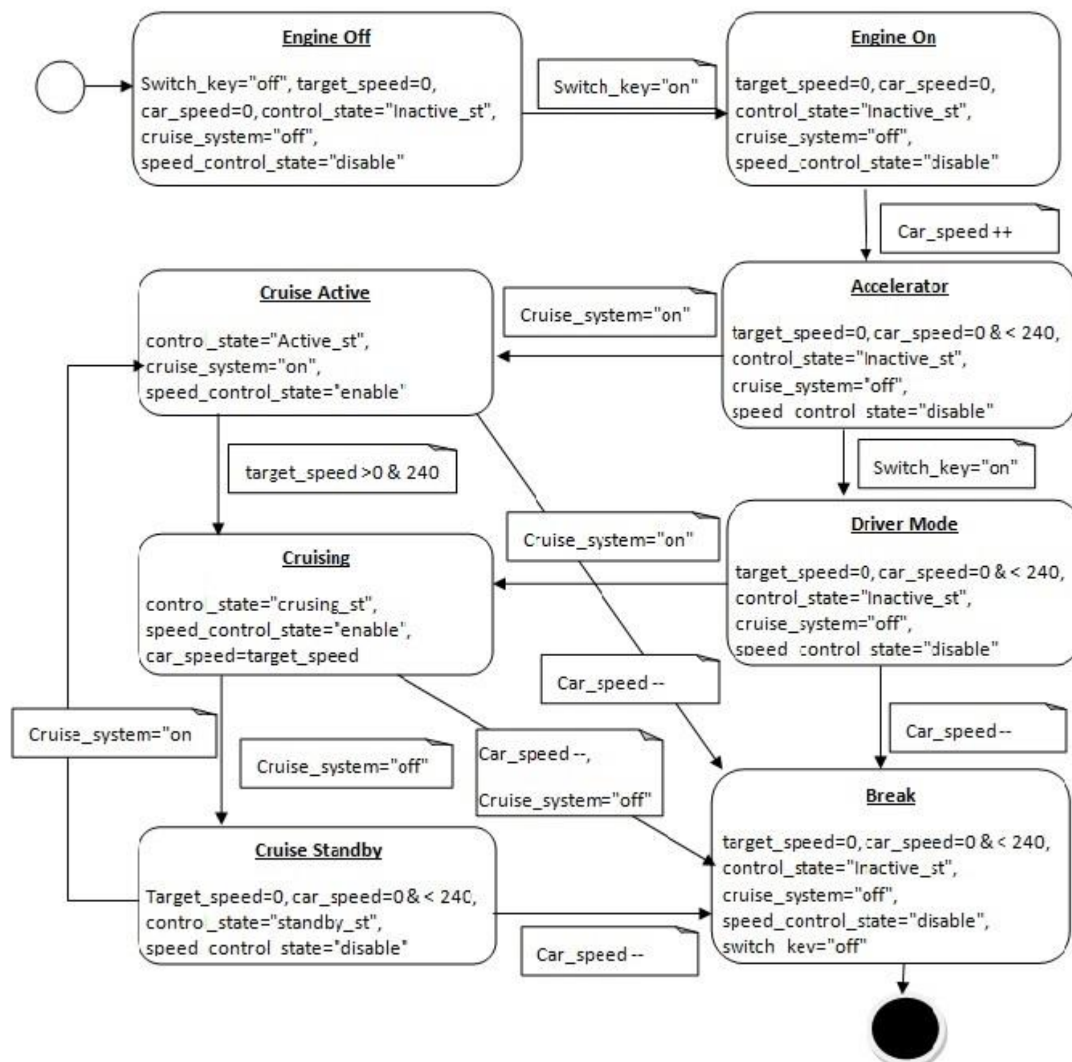

Figure 2. Cruise control system finite state machine diagram.

4.2 CCS Formal Model Using Perfect Language

In the second phase of our methodology, we build the CCS formal model from the informal specifications of the CCS using Perfect language. The formal specification consists of two main classes; the first one is the speed control class and the second is the controller class. The speed control class describes the system-related speed variables and schemas (functions). The controller class describes the related CCS variables and schemas. Due to space limitation, Table 1 shows the CCS formal model of the control class only.

4.3 Formal Model Verification Using Perfect Developer Toolset

In the third phase, the Perfect CC specification that is described in subsection 2.2 is verified by Perfect developer toolset in order to check the syntax, the semantic correctness and the validity of the model. The verifier in Perfect Developer is static analysis and automated theorem proving that collects and attempts to discharge proof obligations for the software with which it is presented.

4.4 FSM Adjacency Matrix for CCS Formal Model

The fourth phase involves the process of adjacency matrix extraction from the verified formal CCS model, which is shown in Table 2. The adjacency matrix reflects the system's states and what the adjacent state to each state in the system is. In the adjacency matrix, 'one' indicates what other states each state can go to. For example: "Engine Off" state can go to "Engine On"' state only, while "Engine On" can go to "Accelerator" state only. Zero indicates that there is no transition between the two states.

Table 1. Part of CCS Perfect formal model

```
class CONTROLLER ^=
abstract
const Inactive_st : int ^= 0;
const Active_st   : int ^= 1;
const Cruising_st : int ^= 2;
const Standby_st  : int ^= 3;
var   ControlState: int ;
var sp_con: SPEED_CONTROL ;
var getten_target_speed: int ;
var engin : bool ;
var cruise : bool;
build{!sp_con: SPEED_CONTROL }
post
ControlState != Inactive_st ,
engin != true ,cruise != false ,
getten_target_speed !sp_con.TargetSpeed;
schema !Break
pre
 sp_con.TargetSpeed <sp_con.CarSpeed ,
 engin = true ,cruise = true ,
 ControlState = Cruising_st
post
 sp_con!DisableControl ,
 ControlState != Standby_st ;
  schema !Accelerator
 pre
 engin = true ,cruise = true ,
 sp_con.TargetSpeed > sp_con.CarSpeed ,
 ControlState = Cruising_st
 post
 sp_con!EnableControl ,
 ControlState != Cruising_st ;
 schema !EnginOff
 pre
 ControlState = Active_st |ControlState
= Cruising_st  | ControlState =
Standby_st | ControlState = Inactive_st
 post
  ([ControlState = Cruising_st]:
(ControlState != Inactive_st ,
sp_con!DisableControl , engin != false)
, []: engin != false ) ,
  ([ControlState =  Active_st]:
(ControlState != Inactive_st ,
sp_con!DisableControl , engin != false)
, []: engin != false ),
  ([ControlState = Standby_st]:
(ControlState != Inactive_st ,
sp_con!DisableControl , engin != false)
, []: engin != false ) ,
  ([ControlState = Inactive_st ]:
(ControlState != Inactive_st ,
sp_con!DisableControl , engin != false)
, []: engin != false );
schema !EnginOn
pre
 ControlState = Inactive_st
```

```
post
 sp_con!ClearSpeed,
 ControlState != Inactive_st ;
  schema !CruiseOn
 pre
 ControlState = Inactive_st
st
 getten_target_speed != sp_con.TargetSpeed
  ,
 sp_con!EnableControl ,
 ControlState != Active_st ;
 schema !Cruise_Crusing
 pre
 engin = true ,
 ControlState = Active_st,
 sp_con.CheckSpeed = true
 post
 ControlState != Cruising_st ;
 schema !Cruise_Active
 pre
 engin = true ,cruise = true,
 sp_con.TargetSpeed > 0 ,
 sp_con.CarSpeed > 0
 post
 ControlState != Active_st ;
 schema !Cruise_Inactive
 pre
 engin = true ,
 cruise = false,
 sp_con.TargetSpeed = 0 ,
 sp_con.CarSpeed > 0
 post
 ControlState != Inactive_st ;
 schema !Cruise_Standby
 pre
 engin = true ,
 cruise = true,
 sp_con.TargetSpeed > 0 ,
 sp_con.CarSpeed > 0
 post
 sp_con!DisableControl ,
 ControlState != Standby_st ;
   schema !CruiseOff
  pre
  ControlState = Cruising_st | engin =
 false
   post
 sp_con!DisableControl ,
 ControlState != Standby_st ;
  schema !CruiseResume
  pre
  engin = true,
  cruise = true ,
  ControlState = Standby_st
  post
  sp_con!EnableControl,
  ControlState != Cruising_st ;
       end;  // the end of class CONTROLLER
```

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 1, No. 1, December 2015.

Table 2. CCS adjacency matrix.

State	Engine Off	Engine On	Accelerator	Driver Mode	Cruise Active	Cursing	Cruise Standby	Break
Engine Off	0	1	0	0	0	0	0	0
Engine On	0	0	1	0	0	0	0	0
Accelerator	0	0	0	1	1	0	0	0
Driver Mode	0	0	0	0	1	0	0	1
Cruise Active	0	0	0	0	0	1	0	1
Cursing	0	0	0	1	0	0	1	1
Cruise Standby	0	0	0	0	1	0	0	1
Break	0	0	0	0	0	0	0	0

4.5 Extraction CC Formal Model Paths, States and Edges

The testing C# tool develops an algorithm called “extractPath” in order to extract the CCS finite state machine paths, states and edges. The “extractPath” algorithm depends on the CCS adjacency matrix and the pseudo code shown in Figure 3. In this phase, the verified formal Perfect specification is put into our C# testing tool and the structure of the cruise control system is prepared using the adjacency matrix that is shown in the previous phase. From the matrix, the algorithm in Figure 3 is used to extract the different system states, paths between system states and edges between states.

The extraction process depends on transition-based coverage criteria that focus on the transition sequence between the system states. The transition-based coverage criteria are represented by 1. all system states, 2. all pair-transitions where each state can go and 3. all system paths.

The algorithm shown in Figure 3 starts by identifying a list of strings (P) that contains all system state paths extracted from the adjacent matrix for a given state. Each state is given a number; for example, Engine Off =1, Engine On=2 … and so on. Then, we find all the adjacent states (by checking the adjacent matrix) for the current state, calling the function find_adjacent (). This function will add the adjacent state numbers to the paths in list P. For example, when list P contains an item (1,2), it means that there is a path from state 1 to state 2. Then, the algorithm continues by finding the adjacent state to the previous state in the path, in this example state 2. Adjacent to state 2 is state 3 (Accelerator). So, state 3 is added to the path (1,2) and the new path (1,2,3) is added to P ; P=[(1,2), (1,2,3)]. In this case, P has two paths. Next, the algorithm finds the adjacent states for state 3 which are states 4 and 5, which are both added to the current path. List P is now equal to [(1,2), (1,2,3),(1,2,3,4), (1,2,3,5)]. The algorithm continues until all paths that each state can reach are found.

Finally, the algorithm extracts the test paths (TP) from list P. A test path is a path that starts from the initial state which is state 1 and ends with the final state which is state 8.

```
For each state in the adjacent matrix Do
        define P as a list of strings contains all paths from the current state
        P= find_adjacent(currentstate)
        IF P.length === 0 then
                there is no path from the current state
        Else
                count=0
                While counter < = P.length DO
                        String currentPath = P.getitem [counter ]
                        //get the last index of the current item in the list P
                        I= currentPath[currentPath.length - 1].tostring()
                        P= find_adjacent(I)
                        counter ++
                End DO
        End IF

From P get all test paths TP that starts with state 1 and ends with state 8


Function list <String> find_adjacent (i)
        J=1
        String S=P[i]
        For state =1 to the last state DO
                IF adj_Matrix [i,state]== 1 then
                        // add the state # to the path P[i]
                        S= P[i] + state
                        add the pathe S to the list P
                End IF
        return P
        End For //list filled with the adjacent states
```

Figure 3. The algorithm of path extraction.

4.6 Automatic Test Case Generation

After the system paths are extracted and system states counted, we automatically generate test cases from the verified cruise control system formal specification through the developed C# testing tool. Test cases will be generated from the test paths generated in the previous phase. For the process of test case generation, we use three algorithms:

Algorithm (1): random test case generation process that generates test cases without any restrictions. So, some of the test cases are considered to be redundant test cases. Algorithm 1 is presented in Figure 4. In the algorithm, TCR1 is a list that contains a set of test cases that are randomly generated from the test path (TP).

```
Algorithm 1
define TCR1 as a list of string that contains test cases generated using Alg 1
For I=1 to N // where N is the number ot test cases to be generated
        //get a random number
        item = Random (1, TP.length)
        TCR1.add( TP.ElementAt(item))
End For
```

Figure 4. Algorithm1.

Algorithm (2): random test case generation with pure restrictions. This algorithm depends on a random generation process with partial restrictions on the generated test cases. This algorithm restricts the results by which a set of randomly unique system paths must be generated for the results each time. In Figure 5, the algorithm randomly generates N test cases from the test path (TP) in such a way that M of them must be unique paths; where M is less than N. The others (N-M) could be redundant.

```
Algorithm 2
define TCR2 as a list of string that contains test cases generated using Alg 2
countDis=0 //represents the number of unique test cases
countAll=0 //to count the number of the generated test cases
// M is the number of unique test cases, M should be less than N
While (countAll <= N)  // where N is the number of test cases to be generated
        //get a random number
        item = Random (1, TP.length)
        IF (countDis <= M  )
                IF( TP.ElementAt(item) is not in TCR2 list )
                        TCR2.add( TP.ElementAt(item))
                        countDis++
                        countAll++
                End IF

        else
                TCR2.add( TP.ElementAt(item))
                countAll++
        End IF
End While
```

Figure 5. Algorithm 2.

Algorithm (3): random test generation with full and optimal restrictions. The algorithm restricts the selected paths by which all system paths must be chosen in the algorithm random process. Moreover, the algorithm forces the generation process to uniquely generate a number of test cases. In other words, if we need N test cases, then we have to generate them randomly and each one should be unique (redundancy in test cases is not allowed). Figure 6 shows algorithm 3.

```
Algorithm 3
define TCR3 as a list of string that contains test cases generated using Alg 3
countDis=0 //count represents the number of unique test case
 // where N is the number of test cases to be generated
While (countDis <= N && countDis < Tp.length)
        //get a random number
        item = Random (1, TP.length)
        IF ( TP.ElementAt(item) is not in TCR2 list )
                TCR3.add( TP.ElementAt(item))
                countDis++
        End IF
End While
```

Figure 6. Algorithm 3.

4.7 Coverage Evaluation for the Generated Test Cases

Finally, the test cases generated in the previous phase are evaluated to determine the system path coverage, state coverage (node coverage) and transition coverage (edge coverage). A set of coverage measurements is used here to evaluate path, state and transition coverage.

5. EXPERIMENTS AND EVALUATION

In our experiments, we use our framework to automatically generate the test cases using the three algorithms for the CCS. Subsection 5.1 discusses the coverage evaluation measures used in our framework. Subsection 5.2 makes a comparison between the three developed test case generation algorithms in terms of coverage and performance evaluation used in this framework.

5.1 Evaluation Measures

The evaluation process of our C# testing tool depends on three matrices in terms of coverage evaluation measurement. The three matrices shown in Figure 7 are state (node) coverage matrix, path coverage matrix and edge coverage matrix. The coverage value for each of the three types depends on the related value of executed state, executed paths and executed edges, respectively, in which the smaller execution value type achieves a smaller coverage value. The number of extracted states, paths and edges represents the number of states, paths and edges, respectively,

from the formal model. The number of executed states, executed paths and executed edges represents the number of states, paths and edges, respectively, that will be processed according to a specific number of test cases. For example, if we have eight states extracted from the system but the test cases that we have used only executed six of the states, then the coverage will be 6/8=75%.

Coverage Type	Coverage Matrix
State (node) coverage =	$\frac{\text{\# Of executed states}}{\text{\# Of extracted states}}$
Path coverage =	$\frac{\text{\# Of executed paths}}{\text{\# Of extracted paths}}$
Edge coverage =	$\frac{\text{\# Of executed edges}}{\text{\# Of extracted edges}}$

Figure 7. Coverage matrix measurements.

5.2 Comparisons and Evaluation Results

5.2.1 CCS Test Case Evaluation

The process of test case generation from the CCS model depends on three algorithms, as described in the previous section. The developed algorithms depend on a random generation process and are evaluated to five numbers of fixed test cases. Moreover, the three algorithms are compared with each other to determine their effectiveness.

As we have seen from Figure 2, the CCS extracted states were eight, the extracted test paths were nine and the extracted edges were twelve. The three random test case generation algorithms were evaluated to five numbers of test cases with 3, 5, 9, 12 and 15 test cases. The evaluation of state coverage results is shown in Table 3.

Table 3. CCS State coverage algorithm results.

T.C#/Algo.#	Algo.#1	Algo.#2	Algo.#3
3 Test Cases	75%	87%	100%
5 Test Cases	100%	100%	100%
9 Test Cases	87.5%	100%	100%
12 Test Cases	100%	100%	100%
15 Test Cases	100%	100%	100%
Average	92.5%	97.4%	100%

Table 3 shows that, for example, when we generate nine test cases by algorithm #1, the CCS state coverage is 87.5%. This means that the nine test cases execute seven system states from all eight extracted system states. When the nine test cases are generated using algorithm #2 (as well as algorithm #3), the state coverage is 100% which means that the nine test cases pass all system states. The averages of the state coverage for algorithm #1, algorithm #2 and algorithm #3 are 92.5%, 97.4% and 100%, respectively, which represents the average from generating 3, 5, 9, 12 and 15 test cases by each algorithm.

Table 4 shows the evaluation results for path coverage using the three algorithms.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 1, No. 1, December 2015.

Table 4. CCS path coverage algorithm results.

T.C#/Algo.#	Algo.#1	Algo.#2	Algo.#3
3 Test Cases	11.12%	33.34%	33.34%
5 Test Cases	33.34%	55.55%	55.55%
9 Test Cases	33.34%	55.55%	100%
12 Test Cases	44.45%	66.66%	100%
15 Test Cases	55.56%	77.77%	100%
Average	35.558%	57.774%	77.778%

Table 4 shows that, for example, when we generate nine test cases by algorithm #1, the CCS path coverage is 33.34%. This means that the nine test cases execute only three test paths from all nine extracted test paths. With the nine test cases generated using algorithm #2, the path coverage is 55.55% which mean that only five test paths from all extracted test paths were executed. For algorithm #3, the path coverage with nine test cases is 100%. This means that those nine test cases execute all nine test paths in the system. The averages of the path coverage for algorithm #1, algorithm #2 and algorithm #3 are 35.558%, 57.774% and 77.778%, respectively. Table 5 shows the evaluation results for edge coverage using the three algorithms.

Table 5. CCS edge coverage algorithm results.

T.C#/Algo.#	Algo.#1	Algo.#2	Algo.#3
3 Test Cases	41.66%	66.67%	75%
5 Test Cases	83.33%	91.67%	100%
9 Test Cases	66.67%	100%	100%
12 Test Cases	83.33%	100%	100%
15 Test Cases	83.33%	100%	100%
Average	71.66%	91.7%	95%

Table 5 shows that, for example, when we generate nine test cases by algorithm #1, the CCS edge coverage is 66.67%. This means that eight edges of the 12 extracted system edges are passed or executed. With the nine test cases generated using algorithm #2 and algorithm #3, the edge coverage was 100% which means that all the 12 extracted system edges are executed. The averages of the edge coverage for algorithm #1, algorithm #2 and algorithm #3 are 71.66%, 91.7% and 95%, respectively.

5.2.2 Test Case Generation Algorithm Results

In this section, we compare the three test case algorithm results. Figure 8 shows the coverage averages obtained from the three algorithms.

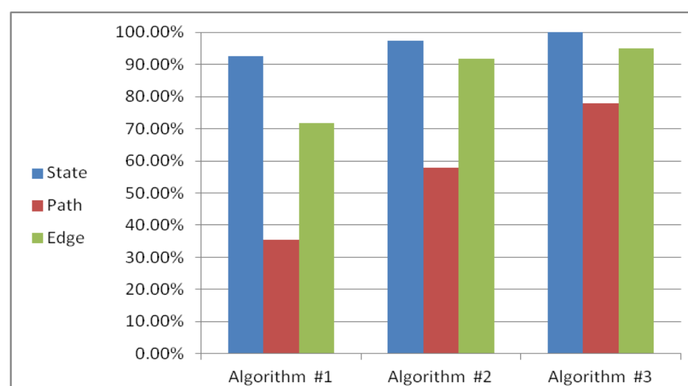

Figure 8. A comparison between the three test case algorithms.

From the results, we can see that algorithm #3 is the best, achieving the most effective coverage values in terms of state, path and edge coverage. Algorithm #2 is better than algorithm #1 in its average results, from which we can conclude that more restriction in the process of test case generation provides more effective coverage results.

6. LIMITATIONS AND FUTURE WORK

Our framework suggests there are benefits in the use of the Perfect formal model for efficiently testing a safety-critical system. However, it is important to note the limitations of our evaluation experiments and framework. First, our framework accepts only the CCS Perfect model and no other safety-critical systems. Moreover, more experimentation is necessary to strengthen the evaluation of our framework, ideally using a safety-critical system used in industry. Second, the finite state machine adjacency matrix was built manually. However, the automation of this process should be possible.

7. CONCLUSIONS

In this paper, we have reflected the importance and role of formal methods in developing safety-critical system formal models. The cruise control system was chosen for this study as a safety-critical system and its state machine diagram was built to reflect the system’s states, transitions and variables.

Perfect formal specification language was used to develop the CCS formal model due to its reliability and effectiveness in presenting transition state systems’ formal models, which is important in reducing software development costs. We built a consistent and reliable formal model that will play a role in increasing system quality and reducing system testing cost and effort.

We proposed three algorithms to generate test cases from the CCS Perfect formal model. The generated test cases were evaluated according to three coverage matrices; state coverage matrix, path coverage matrix and edge coverage matrix. We proved that using formal methods through the safety-critical software development life cycle plays a significant role in improving the testing stage, making it more effective in terms of both effort and cost.

REFERENCES

[1] J. R. Abrial, Modeling in Event-B: System and Software Engineering, 1st Edition, Cambridge University Press, New York, NY, USA, 2010.

[2] P. Ammann and J. Offutt, "Introduction to Software Testing," 2008, Available at: http://www.cs.gmu.edu/~offutt/softwaretest/.

[3] J. Atlee and J. Gannon, "State-Based Model Checking of Event-Driven System Requirements," IEEE Transactions on Software Engineering, vol. 19, issue 1, pp. 24-40, Jan. 1993.

[4] T. Bultan and C. L. Heitmeyer, "Applying Infinite State Model Checking and Other Analysis Techniques to Tabular Requirements Specifications of Safety-Critical Systems," Design Automation for Embedded Systems, vol. 12, issue 1, pp. 97-137, June 2008.
URL http://dx.doi.org/10.1007/s10617-008-9014-2.

[5] G. Carter and R. Monahan, "Software Refinement with Perfect Developer," 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM’05), pp. 363-373, 2005.

[6] Y. Chen, "Generation of Test Cases for Concurrent Software Systems Based on Data-Flow-Oriented Specifications," IEEE 2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering (CNSI), pp. 170-177, 23-25 May.

[7] D. Crocker, "Teaching Formal Methods with Perfect Developer," 2003, http://www.eschertech.com/papers/teaching_formal_methods.pdf.

[8] D. Crocker and J. Carlton, "Verification of C Programs Using Automated Reasoning," in: SEFM '07: Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods, 2007 IEEE Computer Society, Washington, DC, USA, pp. 7-14.

[9] G. Fraser and A. Gargantini, "An Evaluation of Model Checkers for Specification Based Test Case Generation," 2013 IEEE 6th International Conference on Software Testing, Verification and Validation, pp. 41-50, Denver, Colorado, 1-4 April 2009.

[10] C. Heitmeyer, J. Kirby and B. Labaw, "Tools for Formal Specification, Verification and Validation of Requirements," Proceedings of the IEEE 12th Annual Conference on Computer Assurance (COMPASS '97), pp. 35-47, 16-19 June 1997.

[11] A. Iliasov, A. Romanovsky and F. Dotti, "Structuring Specifications with Modes," 4th IEEE Latin-American Symposium on Dependable Computing (LADC '09), PP. 81-88, 1-4 Sept. 2009.

[12] S. Kanwal and N. Zafar, "Formal Model of Automated Teller Machine System Using Z Notation," IEEE International Conference on Emerging Technologies (ICET 2007), pp. 131-136, 12-13 Nov. 2007.

[13] A. Krupp, O. Lundkvist, T. Schattkowsky and C. Snook, "The Adaptive Cruise Controller Case Study Visualization, Validation and Temporal Verification," in UML-B System Specification for Proven Electronic Design, 9 Dec. 2004, Kluwer Academic Publishers 2005.

[14] S. Liu, "Utilizing Test Case Generation to Inspect Formal Specifications for Completeness and Feasibility," 9th IEEE International Symposium on High-Assurance Systems Engineering (HASE'05), pp. 349-356, 2005.

[15] S. Liu and S. A. Nakajima, "A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications," IEEE 2010 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI), pp. 147-155, Singapore, 9-11 June 2010.

[16] Q. Malik, J. Lilius and L. Laibinis, "Scenario-Based Test Case Generation Using Event-B Models," IEEE 1st International Conference of Advances in System Testing and Validation Lifecycle (VALID '09), pp. 31-37, 20-25 Sept. 2009.

[17] D. Mery and N. K. Singh, "Critical Systems Development Methodology Using Formal Techniques," in: Proceedings of the 3rd Symposium on Information and Communication Technology (SoICT '12), pp. 3-12, Viet Nam, 23 – 24 August 2012. URL http://doi.acm.org/10.1145/2350716.2350720.

[18] A. Mishra, M. Rao, C. Cu, V. Rao, Y. Jeppu and N. Murthy, "An Auto-Review Tool for Model-Based Testing of Safety-Critical Systems," in: Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation, JAMAICA 2013, pp. 47-52, ACM, New York, NY, USA, 2013.

[19] Y. Nakatsugawa, P. Kurita and K. Araki, "A Framework for Formal Specification Considering Review and Specification-Based Testing," 2010 IEEE Region 10 Conference-TENCON 2010, pp. 2444 – 2448, ISBN: 978-1-4244-6889-8, Fukuoka, 21-24 Nov. 2010.

[20] J. Offutt, Y. Xiong and S. Liu, "Criteria for Generating Specification-Based Tests," 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99), pp. 119-131, Las Vegas, Nevada, USA, 18-21 October 1999.

[21] I. Sommerville, Software Engineering, 9th Edition. Addison-Wesley, Haelow, England, 2010.

[22] C. Tian, S. Liu and S. Nakajima, "Utilizing Model Checking for Automatic Test Case Generation from Conjunctions of Predicates," Proceedings of the 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2011), pp. 304-309.

[23] G. Tim, "Object-Z to Perfect Developer", 2007, Available at: http://www.doc.ic.ac.uk/~tk106/ObjectZ_project.pdf.

[24] A. Yasmeen, K. M. Feigh, G. Gelman and E. L. Gunter, "Formal Analysis of Safety-Critical System Simulations," in: Proceedings of the 2nd International Conference on Application and Theory of Automation in Command and Control Systems (ATACCS '12), pp.71-81, London, UK, 29-31 May 2012.

[25] D. You and S. Rayadurgam, "Practical Aspects of Building a Constrained Random Test Framework for Safety Critical Embedded Systems," in: Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation (MoSEMInA 2014), pp. 17-25, ACM, New York, NY, USA, 2014.

[26] P. Nilsson, O. Hussien, Y. Chen, A. Balkan, M. Rungger, A. D. Ames, J. Grizzle, N. Ozay, H. Peng and P. Tabuada, "Preliminary Results on Correct-by-Construction Control Software Synthesis for Adaptive Cruise Control," Proc. 53rd IEEE Conference on Decision and Control (CDC), Dec. 2014.

[27] Y. Zhao, "Intuitive Representations for Temporal Logic Formulas," in Proc. of Forum on Specification and Design Language (FDL'03), pp. 405–413, Frankfurt, Germany, September 2003.

[28] A. Gotlieb, B. Botella and M. Rueher, "Automatic Test Data Generation Using Constraint Solving Techniques," in Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98), Will Tracz (Ed.), pp. 53-62, FL, USA, 02 – 04 March 1998.

[29] J. Yan, Z. Li, Y. Yuan, W. Sun and J. Zhang, "BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach," IEEE 17th International Symposium on Software Reliability Engineering (ISSRE '06), pp. 75 – 84, 7-10 Nov. 2006.

[30] S. Jehan, I. Pill and F. Wotawa, "BPEL Integration Testing," 18th International Conference on Fundamental Approaches to Software Engineering (FASE), pp. 69-83, London, UK, 11-18 April 2015.

[31] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufman Publishers Inc. San Francisco, CA, USA, 2007, ISBN:0123725011 9780080466484.

ملخص البحث:

تلعب الطرق الرسمية دوراً مهمّاً في زيادة جودة البرمجيات و إعتماديتها و متانتها وفعاليتها. أضف إلى ذلك أن استخدام الطرق الرسمية، وبخاصة في أنظمة السلامة الحرجة، من شأنه أن يساعد في الكشف المبكر عن أخطاء البرمجيات و عيوبها، الأمر الذي يؤدي إلى إختصار الكلفة و الجهد و الوقت التي يتطلبها اختبار البرمجيات.

يهدف هذا البحث إلى إثبات دور المواصفات الرسمية و فاعليتها في دراسة حالة لنظام من أنظمة قيادة السيارات و التحكم بها. و قد تم بناء نموذج رسمي لهذا النظام باستخدام لغة بيرفكت للمواصفات الرسمية, والتحقق من صحته باستخدام أدوات تطوير لغة بيرفكت. كذلك، تم بناء أداة لفحص البرمجيات جرى استخدامها لإنتاج حالات فحص عن طريق ثلاث خوارزميات مختلفة. وتمّ تقييم حالات الفحص تلك من أجل تحسين تغطيتها وفعاليتها.

أشارت النتائج الى أن خوارزمية التوليد العشوائي لحالات الفحص مع وجود المحدّدات كانت الأفضل من حيث النتائج الخاصة بالتغطية؛ فقد كانت نسبة تغطية المسار 77.78%, في حين بلغت نسبة تغطية الحالة 100%.

و أخيراً، فقد بينت النتائج التجريبية التي تم الحصول عليها أنّ لغة بيرفكت للمواصفات الرسمية ملائمة للاستخدام في نظام قيادة السيارات و التحكم بها، الذي يعّد من أهم برمجيات أنظمة السلامة الحرجة. كما مكّن ذلك الاستخدام من أن تصبح عملية الكشف عن جميع الحالات المستقبلية أكثر يسراً و سهولة.

This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).