

A Case Study on the Role of Assertions in Model-Based Testing

Pritha Mahata^{*}, Paul Strooper[†], Margaret Wojcicki[‡]
School of ITEE, University of Queensland, Brisbane, Australia
^{*} pritha@itee.uq.edu.au
[†] pstroop@itee.uq.edu.au
[‡] wojcicki@itee.uq.edu.au

Abstract—This paper presents a case study regarding the effectiveness of assertions in the context of model-based testing (MBT). In particular, we use the NModel tool and analyse a cruise control simulation program to show that the traditional conformance testing (CT) can be enhanced in terms of its effectiveness if assertions are used as oracles in the test driver. Additionally we show that in this case study, the bug-detection rate improves when CT and MBT oracles are supplemented with assertions inserted in the system code.

Keywords—model-based testing; assertions; oracles; case study; NModel;

I. INTRODUCTION

Model-based testing (MBT) involves testing software based on a model of the behaviour of the software, such as a finite state machine or statechart [1]. Model-based testing has grown rapidly in terms of research interest and practical application [2]–[4], and a number of academic, public-domain, and commercial MBT tools are now available.

A widely acknowledged problem with model-based testing is the need for a test oracle to check the behaviour of the software during testing [1], [3], [5]. Assertions can at least partially address this problem.

Assertions are logical expressions that can be embedded in software and then checked when the software is statically analysed or executed. When an assertion violation is detected during such an analysis or execution, this is reported to the testing personnel, who can then investigate and fix any potential problem(s).

Although assertions have long been recognised as a potentially powerful mechanism [6] that can be used to support a variety of testing activities, they are not in widespread use in industry. Rosenblum [7] suggests several reasons for this: they are not integrated well into existing development tools, it is not fully understood what kinds of assertions are the most effective in detecting software bugs, and for some programmers, programming with assertions is similar to writing a program twice. Despite the fact that a number of modern programming languages and tools now support the definition, evaluation, and reporting of assertions, their application in industry is still limited. This may be partly due to the lack of research on and practical knowledge about the most (cost-)effective definition and use of assertions in various testing activities.

After executing tests generated by an MBT tool, the next challenge is to design *oracles* which will determine whether the *system under test* (SUT) passed the given test. It is still a research topic to create cost-effective oracles. In the absence of input/output pairs for each test case, *post-conditions* derived from the model can be used as *oracles* in the test driver. The oracles in the test driver check whether the respective *post-conditions* of the model hold after the test cases are executed by the corresponding code in the SUT.

This work presents a case study which shows that in addition to oracles, independent use of assertions in the SUT can enhance the bug-detection rate even further. The study is focused on evaluating the costs and benefits of applying MBT and assertions in combination compared to applying these technologies separately. The case study is performed on a cruise control simulation program. We run the conformance testing tool NModel [8]: (a) without any assertions, (b) with oracles in the test driver, and (c) with additional assertions (pre-conditions, post-conditions, class invariants and inline assertions) incorporated in the SUT. We found that the bug-detection rate increased when oracles were supplemented with additional assertions in the implementation.

II. BACKGROUND

A. Assertions

Assertions are used to help specify programs and to reason about program correctness. Several modern programming languages include checked assertions - statements that are checked at runtime or sometimes statically. If an assertion evaluates to false at run-time, an “assertion violation” results, which typically causes execution to abort. This draws attention to the location at which the logical inconsistency is detected.

Assertions can be broadly classified as follows [7]:

- pre-conditions: conditions or predicates that must always be true just prior to the execution of a function.
- post-conditions: conditions or predicates that must always be true after execution of a function.
- class invariants: invariants used to constrain objects of a class. Functions of the class should preserve the invariant, i.e., before and after every function call,

the invariant should be preserved. The class invariant constrains the state stored in the object.

- inline assertions: location-specific assertions that check properties at key locations in the code. For example, inline assertions can check the implicit condition of the final else in an if-else statement.

In languages such as Eiffel [9], assertions are part of the design process, and in others, such as C, C++, C# and Java, they are used to check assumptions at runtime; however they can be suppressed.

B. Model-based Testing

Model-based testing (MBT) is software testing in which test cases are derived in whole or in part from a model that describes some (usually functional) aspects of the SUT.

The model is usually an abstract, partial representation of the SUT's desired behaviour. Often it is a finite state machine or a UML state diagram. The test cases derived from such models are functional tests on the same level of abstraction as the model. These test cases are collectively known as the "abstract test suite". The abstract test suite cannot be directly executed against the system under test. Therefore an executable test suite must be derived from the abstract test suite that can communicate with the SUT. This is done by mapping the abstract test cases to concrete test cases suitable for execution. Often, a test driver (also called adapter or stepper) is written in an MBT tool which performs this mapping.

All software testing methods depend on the availability of an *oracle*, that is, some method for checking whether the system under test has behaved correctly on a particular execution. The oracle can be a human being. Relying on a human to assess program behaviors has two drawbacks: accuracy (error-prone when assessing complex behaviors) and cost. Ideally, a tester is provided with an output for each input. Finding/creating a test oracle can be an issue in MBT. Tests are generated automatically and in volume. Thus, calculating expected outputs by hand is usually infeasible. Some work has explored the automatic generation of test oracles [10]. However, it is difficult to generate test oracles automatically for an arbitrary SUT. In the absence of a good test oracle, one may need to settle for plausibility checks. Tests may be considered to have passed if their outputs are in certain ranges or they pass certain consistency checks.

MBT tools can use conformance testing for checking whether SUT passes the tests. Conformance testing in the context of MBT only ensures that the SUT code corresponding to each action in the model is executable. Additionally, tools often incorporate oracles within the test driver to check whether a model's post-conditions hold after the SUT steps through the corresponding system code.

C. NModel

In this work, we use *NModel*, a model-based testing and analysis framework for programs written in C# [8]; (see <http://www.codeplex.com/NModel>). *NModel* provides solutions to the problems of expressing and analysing specifications and designs, generating test cases, and checking the results of the test runs. Only actions and their enabling conditions are required as the model, instead of an explicit state machine. This is important when handling industrial case studies with thousands of states.

NModel includes a library of attributes and data types for writing models in C#, a test generation tool *otg* (Offline Test Generator), and a test runner tool *ct* (Conformance Tester).

To execute tests using the test runner *ct*, a test driver in C# needs to be written which couples the implementation to the model. One can use the test generator *otg* to create tests from the model program in advance, or let *ct* generate the test on-the-fly from the model as the test run executes. It is also possible to write a custom strategy in C# that *ct* can use to maximise coverage according to the criteria defined.

MBT test suites are generated by *ct* by following the actions allowed by the model. In a situation where more than one action is allowed, an action is chosen randomly. The online test generator *ct* allows a lot of flexibility by supporting random test case generation and handcrafted test suites.

It is also possible to use an offline test generator *otg*, which uses the Chinese Postman algorithm [11]. This algorithm generates a tour of all transitions with a minimum total number of actions. In this case study, relatively short test sequences generated by the Chinese Postman algorithm did not catch any bugs, so we used the random test case generation provided by the *NModel* tool.

III. SYSTEM UNDER TEST: CRUISE CONTROLLER

A cruise controller system is a typical example of a software-based embedded system in automotive electronics. The system is controlled by three buttons: 'resume', 'on' and 'off'. When the automobile's engine is running the user can press 'on' which activates the cruise control system to record the current speed as the cruising speed and maintain the automobile at this speed. The accelerator, brake or 'off' button stop the cruise control system, however the cruising speed remains recorded. When the user presses 'resume' the system will accelerate or de-accelerate the automobile back to the cruising speed. To ensure an efficient development process for such systems, appropriate techniques for specification and validation have to be applied. In our case study, the model of a simplified cruise controller simulator is written in C# following the finite-state machine model (Figure 1), and the SUT is a C# translation of the Java implementation [12]. Notice that we only consider the central logic of the cruise controller and omit the graphical user interface related code.

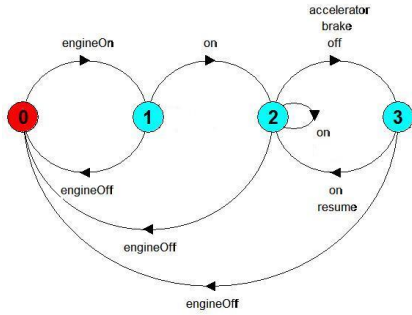


Figure 1. Cruise Controller: finite-state machine model; Figure 8.10 from [12].

The SUT, even though simple, has an interesting complexity related to synchronisation features. The cruise controller spawns a new thread each time cruise control is enabled. Thus, there is a potential source of synchronisation-related malfunctioning, e.g., a data race. Data races arise in software when separate processes or threads depend on some shared state.

A. Cruise Controller Model

The cruise controller model is encapsulated in the *Controller* class (Figure 2). The controller can stay in one of the four states: INACTIVE, ACTIVE, CRUISING, or STANDBY, which are enumerated by *cstate*. The variable *controlState* denotes the current state of the controller.

An NModel model is ‘specified’ in C# and the actions and their enabling conditions of the model are represented as C# functions. Each of the actions of the model is a C# function preceded by a keyword “[Action]”. We introduced seven functions corresponding to the events in Cruise Controller state machine (*engineOn*, *engineOff*, *accelerate*, *brake*, *on*, *off*, and *resume*). The enabling conditions specify the control states from which the actions can be performed. These conditions are written as functions postfixed with the word “Enabled”. For instance, the enabling condition for the *engineOn* action is given by the function *engineOnEnabled()* (see Figure 2).

In the following paragraph, we illustrate how we specify the *engineOn* and *engineOff* actions and their enabling conditions in NModel. Other actions are modelled in a similar manner and not shown in Figure 2.

The states 0–4 in Figure 1 are represented using the enumerated type *cstate* in Figure 2, and the state variable *controlState* maintains the current state. The *engineOn* action in NModel results in the transition from state 0 to state 1 in Figure 1. However, the *engineOff* action is associated with three transitions in Figure 1, since the enabling condition for *engineOff* is satisfied in all the states except state 0. These three transitions of the finite state machine model are succinctly represented by just one action with its one enabling condition (*controlState != cstate.INACTIVE*) in the

```

public static class Controller {
//Types for the state of controller.
enum cstate { INACTIVE, ACTIVE,
              CRUISING, STANDBY };

static cstate controlState = cstate.INACTIVE;

static bool engineOnEnabled() {
return (controlState == cstate.INACTIVE);
}

[Action] public static void engineOn() {
controlState = cstate.ACTIVE;
Console.WriteLine("engineOn");
}

static bool engineOffEnabled() {
return (controlState != cstate.INACTIVE);
}

[Action] public static void engineOff() {
controlState = cstate.INACTIVE;
Console.WriteLine("engineOff");
}
....

[AcceptingStateCondition]
static bool Accept() {
return (controlState = cstate.INACTIVE);
}
}

```

Figure 2. Model of Cruise Controller. Includes enabling conditions for actions and the action methods.

model written using NModel. This is particularly useful for large models with thousands of transitions, which can often yield much shorter models with smaller number of actions in NModel. In NModel, a run of the model may only terminate in an accepting state. The predicates that determine whether a given state is an accepting state are given by the “[AcceptingStateCondition]” attribute (Figure 2). Note that there can be several accepting states in a given model and that usually results in a large number of test sequences as well as many short test sequences.

In NModel, a run of the model may only terminate in an accepting state. The predicates that determine whether a given state is an accepting state are given by the “[AcceptingStateCondition]” attribute (Figure 2).

B. Implementation

Our Cruise Controller implementation consists of four C# source files : *Controller.cs*, *SpeedControl.cs*, *CarSpeed.cs*, and *CarSimulator.cs* (see Appendix for the code listing of *Controller* and *SpeedControl*).

The *Controller* class closely follows the model’s state transitions and is a direct translation from the Java code [12]. Each method modifies the control state and invokes speed control actions using the functions from the *SpeedControl*

class. The Controller class is implemented as a monitor with each model action becoming a synchronised method. More precisely, the Controller class contains the definitions of its four states: *INACTIVE* (0), *ACTIVE* (1), *CRUISING* (2) and *STANDBY* (3). It also contains a variable called *controlState* which is set to *INACTIVE* initially. The variable *controlState* is updated in the seven functions *engineOn()*, *engineOff()*, *on()*, *off()*, *resume*, *accelerator()* and *brake()*, corresponding to the seven actions in the model.

Another attribute of the Controller class is the SpeedControl instance *sc* for the speed controller. The SpeedControl class interacts with the car simulation provided by the class *CarSimulator* via the interface *CarSpeed*. This interface provides methods to set the throttle and to get the current speed. When SpeedControl is enabled, a thread is created which periodically obtains the current car speed from the car simulator and sets the throttle so that the target cruising speed is maintained. Also, the SpeedControl class defines a variable *state*, which takes two values: *DISABLED* and *ENABLED*. SpeedControl's state is changed within the Controller's functions.

IV. CASE STUDY

A. Goal

The goal of this case study is to provide empirical evidence regarding the effectiveness of using MBT oracles and assertions in the SUT for testing purposes.

B. Objects

1) *System Under Test*: The system under test consists of two C# source files from the CruiseController implementation.

2) *Bugs*: We asked three postgraduate students to seed bugs in the implementations' Controller class and SpeedControl class. Each introduced five bugs. Table I contains a detailed description of the seeded bugs in these two classes.

The bugs can be syntactically classified as follows:

- changes in assignment to boolean/integer/float/double variables - bugs 2,3,4,5,7,9, and 10
- change in control flow (removal of "if" statements, change in "if" condition, inclusion/exclusion of statements to/from the existing "if" statement)- bugs 6, 8, 11, 12, 13, and 15
- synchronisation bugs - bugs 1 and 14

Some of the bugs turned out to be "equivalent" in the sense of "observational equivalence" (i.e. bugs 4, 5, 6, and 11).

Furthermore, some bugs turned out to be "out of the scope of MBT"; for example, bug 3 and bug 14. By saying "out of scope", we mean that our analysis shows that the test sequence required to catch such bugs will never be generated by the given model we used.

C. Methodology

The general procedure for our study is as follows.

- 1) Instrument the code with bugs, run *ct* and record how many bugs are detected. This step corresponds to simple conformance testing.
- 2) Incorporate oracles within the test driver (the code used to call functions from implementation for the given actions in the model). Run *ct* and record how many bugs are detected by the oracles.
- 3) Incorporate assertions (the process is detailed below) in the SUT and perform testing with *ct* and record how many bugs are detected by assertions in the SUT.

Incorporation of assertions essentially followed Rosenblum's guidelines (in short, incorporate specifications of function interfaces using pre- and post-conditions and incorporate specifications of function bodies using inline assertions; see [7] for details). More precisely, our approach is to:

- 1) use model's pre- and post-condition where the functions in the SUT are the same as that in the model;
- 2) enhance model's pre- and post-conditions by examining the functions;
- 3) add pre- and post-conditions for the additional functions in the implementation;
- 4) add inline assertions and class invariants appropriately.

The approach detailed above is the 'ideal'; however, during the initial phase the approach needed to be accommodated to deal with the exploratory nature of the study (for example, the need for familiarisation with MBT in combination with assertions). For instance, the bug seeding took place on a version of the code with a few pre- and post-conditions inserted in the Controller class. However, those assertions were not derived from the model, and rather mimicked the code in the SUT. Later, we decided to insert pre- and post-conditions according to the MBT model for those functions (e.g., the function in the Controller class) which occur both in the model and the SUT.

D. Execution

Each run in NModel consists of 30–60 actions and 10 such runs were used in each conformance testing phase. NModel generated action sequences with both action coverage and transition coverage as the coverage criteria. We also increased the total number of actions and the number of runs to hundreds in our experiments, but the larger number of actions/runs did not change the results.

Oracles in the test driver of the Cruise Controller: For the seven controller functions belonging to the Controller class, we incorporated seven assertions (*post-conditions* for the relevant actions) as oracles in the test driver. These post-conditions are derived from the finite state model of the controller and check that the target state is reached by a given action.

Table I
BUGS SEEDED IN SYSTEM.

No.	Class	Function	Line	Description
1	SpeedControl	Run()	75	SpeedController = null;
2	Controller	Accelerator()	28	controlState = ACTIVE instead of controlState = STANDBY
3	Controller	EngineOn()	53	The code for setting the speed to 0 is deleted.
4	SpeedControl	clearSpeed()	30	Speed is unconditionally set to 0.
5	SpeedControl	clearSpeed()	30	Added disableControl().
6	Controller	EngineOff()	43,46,61	The if statement is deleted.
7	SpeedControl	GetSpeed()	38	wrong speed is returned if state == ENABLED
8	SpeedControl	EnableControl()	46,51	Removed if statement.
9	SpeedControl	Run()	70	Replaced 6 By 12 in the denominator of the "error" expression.
10	SpeedControl	Run()	71	Replaced 12 By 6 in the denominator of the expression for "steady".
11	Controller	EngineOff()	44	Remove "if" condition
12	Controller	EngineOff()	45	If statement also groups the assignment of controlState = ACTIVE
13	Controller	On()	61	Conditional expression is changed.
14	SpeedControl	GetSpeed()	37,39	Removed synchronization.
15	SpeedControl	EnableControl()	43,54	Added an additional "if" condition to spawn a new speedController thread.

Assertions in the SUT: The code listing in the Appendix contains all the assertions discussed in this section.

Pre- and post-conditions and invariants:

- 1) We incorporated pre- and post-conditions in the Controller class. Most post-conditions are the same as those in the oracles used in the test driver. Only the *engineOn()* function's post-condition additionally performs a check to ensure that the SpeedControl instance has its speed set to 0.
- 2) We also incorporated pre- and post-conditions in the SpeedControl class and the CarSimulator class. In the following, we give an example of a non-trivial assertion (post-condition) in the SpeedControl class's *enableControl()* function. This assertion states that
 - the SpeedControl object is in the state "ENABLED", and
 - either of the following holds:
 - at the entry to this function, the state was ENABLED and the speedcontroller thread is the same before and after the execution of this function, or
 - at the entry to this function, the state was DISABLED and the speedController thread's references are different before and after the execution of this function.
- 3) We added one class invariant in the Controller class which states that the SpeedControl instance is non-null. Also, in the SpeedControl class, we added another class invariant which states that either the SpeedControl instance is not in the enabled state or its speedController thread is not null.

Inline assertions:

The SpeedControl class published in Magee and Kramer's [12] first edition of 'Concurrency: State Models and Java Program' had a statement in the *run()* function, which sets its speedController thread to null in a non-synchronous manner. This turns out to be a bug. This bug is exposed by an

assertion in the *enableControl()* function that checks if any previous assignment suddenly makes the speedController thread null, as it is supposed to be non-null after it is started. See bug 1 for details (Table I) and also the Appendix for the implementation of the SpeedControl class.

For the case study, we add the statement "speedController = null" in the *run()* function of the SpeedControl class as bug 1, and remove it from our SUT for the other implementations. Overall, we inserted 28 assertions in the three classes with 20 functions.

E. Results

Table II summarises the results. Out of the 15 bugs, 4 of them (4,5,6, and 11) are equivalent and not shown. Two of them are out of the scope of MBT (3 and 14) with respect to the given model.

Use of just conformance testing did not detect any of the 9 remaining bugs. Out of these 9 bugs, the oracles in the test driver caught bugs 2, 12 and 13. The assertions (pre- and post-conditions and inline assertions) inserted independently within the SUT catch 5 bugs (the 3 caught by oracles and bugs 1 and 8), i.e., 5 out of 9.

Models can be incomplete, or even incorrect and sometimes, testers may need to adjust a given model based on their experiences in executing test runs of the implementation. For instance, a simple modification of the model enabled us to catch bug 3 (a deletion of code that sets the speed to 0). If we allow the action *accelerate* to be used before *on* in the model, then we can create a scenario in which the engine can be turned on from a non-zero speed. This will then violate the *engineOn()* function's post-condition. We also need to change the pre-condition of the accelerator function to be able to catch this. After these modifications were made, *ct* furnished us with a long test-sequence that violates the post-condition of the *engineOn()* function. We mention this just to show that given an appropriate test sequence, assertions will be able to catch some of the out

of scope bugs as well, if those test sequences are run in a stand-alone fashion in the SUT. Since bug 3 is caught, 6 out of 11 non-equivalent bugs get revealed by assertions in total.

It is also interesting to note that the uncaught bugs were either numerical ones (bugs 7, 9, and 10) or synchronisation-related (bug 15), or out of scope of MBT (bugs 3 and 14). Unless we write some assertions replicating the assignments, the numerical bugs will not be caught. However, we believe that incorporating such assertions may not be cost-effective as it could easily complicate the code and slow the execution of the program during testing. On the other hand, putting specific delays in the functions belonging to the SpeedControl class may reveal bug 15, but that is outside the scope of this paper.

F. Analysis

The inconsistency between the model and the implementation leads to non-equivalent bugs (like bug 3) that are difficult to catch with MBT. MBT oracles detected 3/11 bugs whilst assertions detected 6/11 bugs (which include the 3 bugs detected by MBT oracles). One of the additional bugs was detected by a post-condition in the context of our model.

Furthermore, inline assertions were added to the code with independent test suites and this resulted in the detection of bug 1 (a synchronisation bug). Assertions depend on the code that is executed and if this means that particular sequences that would check the assertions are not executed because of the model (or the NModel settings) then an issue can arise in which assertions are not being used to their full potential with MBT (e.g., in the case of bug 3, which is caught after changing the model). In the end, there were five bugs that were not detected at all.

G. Threats to Validity

Since this study does not include statistical analysis, the focus of threats to validity is on internal and external validity.

1) *Internal:* Assertions already present in the code used for bug seeding could confound results as the subjects may have been influenced in their task by the presence of the assertions to attempt to seed bugs that would not be caught by the assertions; however only the Controller class included some experimental assertions (almost mimicking the SUT code) prior to bug seeding (assertions were added to the other classes following bug seeding). Later we also changed the assertions in the Controller class according to the method of inserting assertions mentioned in Section IV-C. We have since developed guidelines for inserting assertions which are currently being used in an industrial case study that follows up on this work.

2) *External:* In ‘real-life’ situations practitioners are not aware of the bugs in the code when deriving the assertions so therefore assertions cannot be ‘tailored’ to find particular

bugs in practice as they are in this case; however, it helps to see if one can derive an assertion that can find these bugs at all. This information can be used to establish a set of guidelines (see Section V-A) that can be used to derive assertions systematically to detect particular types of bugs.

The selection of a representative SUT is not a trivial task and the case study SUT is not representative of all software systems; however it is a well-known example of an embedded software system with enough complexity to provide useful observations. It is also difficult to seed representative bugs; however the bugs do cover a diverse range of issues as seen in the classification (see Section IV-B2).

V. DISCUSSION

The case study focused on the effectiveness of using assertions in MBT; however we also examine the use of NModel itself and the process of deriving assertions.

A. Effectiveness

In this work, we evaluated the effectiveness but not the cost of using assertions in the context of MBT. Analysing cost is more challenging, since it requires one to come up with metrics to judge how easy it is to incorporate assertions of different categories which probably depends on the skills of the assertion writer. Such analysis could involve an experiment that examines whether or not assertions can be derived systematically in a cost-effective manner and whether or not this derivation would be based on the model, the specification within code (if any), and/or the implementation itself (this is planned in future work).

The results of the study depend on MBT (and NModel) and the choice of the model (especially with respect to its consistency with the implementation). Since there were five bugs that were not found, it is of interest to determine whether better models that handle more test cases can influence the detection of bugs.

B. Using NModel

In our experience, NModel is easy to use, since it is well-documented, especially for modelling. It also supports flexible configuration options for test coverage and test environment set-up, and a friendly UI. However, model actions must be hooked up (via the test harness) to .NET functions. Like many MBT tools, NModel provides limited support for test oracles. For example, the cruise controller SUT offers a graphical user interface (GUI). However, there is no mechanism to specify a test oracle in NModel for checking whether the GUI actions were performed as expected or not. In fact, this is currently a research topic. In absence of such facilities, we concentrated our study only on the core operating logic of the cruise controller, by eliminating the GUI part.

Table II
RESULTS FOR NON-EQUIVALENT BUGS.

Bug No.	Out of Scope of MBT	Detected By Oracle (Y/N)	Detected by Assertions added to the implementation		
			Y(Type)/N	Assertion	File:Function
1	N	N	Y (inline assertion)	SpeedController != null	SpeedControl:enableControl()
2	N	Y	Y (post-condition)	ControlState == STANDBY	Controller:accelerator()
3 [^]	Y	N	Y (post-condition)	ControlState == ACTIVE && sc.getSpeed() == 0	Controller:engineOn()
7	N	N	N		
8	N	N	Y (post-condition)	checks if additional thread is spawned in ENABLED state	Speedcontrol:enableControl()
9	N	N	N		
10	N	N	N		
12	N	Y	Y (post-condition)	ControlState == INACTIVE	Controller:engineOff()
13	N	Y	Y (post-condition)	ControlState == CRUISING	Controller:on()
14	Y	N	N		
15	N	N	N		

C. Deriving Assertions

In this case study, deriving assertions from the model was straightforward. However, from our experience with the SpeedControl class, we find that introducing assertions involving threads is non-trivial (as also observed by others [9], [13], [14]).

VI. RELATED WORK

A. Evaluating Model-based Testing tools

Model-based testing is an active area for empirical study as observed by Neto et al. [15] in their systematic review of MBT studies up to mid-2006. Of the 85 studies they reviewed, 7 empirically evaluate effectiveness of MBT methodologies. Empirically, model-based testing continues to be active as shown in the 15 case studies that have been reported following mid-2006. Six of these studies took place in an academic setting and included evaluation of MBT with varied technologies such as UML [16] and the use of SpecExplorer [17] which found MBT to be effective, as well as an approach called MODEST with its accompanying tool MODESToo [18]. An evaluation of MODEST showed cost reduction through decreased test effort and benefit to test quality through increased defect detection. The context of use also varied including an open-source GUI [19] for which the approach was useful in detecting severe faults and concurrent programs validated with predicate sequence constraints [20].

Similar to MODEST, two industry-based case studies presented an increase in defect detection with MBT [21], [22]. Three of the industry-based studies were in the automotive industry [22]–[24] and one was used on health-care products [25], suggesting a need for an effective and reliable testing approach in such contexts. UML based MBT was evaluated in two studies [26], [27] and showed benefits as test cases could be derived at early stages of development and maintained with lower cost, whilst the use of the SAL language and tool suite provided an all-in-one solution with fewer costs in guiding the design of time-triggered systems

[28]. Interestingly, Aydal et al.'s [29] case study reported that MBT did not detect all defects therefore formal verification would need to be combined with other testing technologies for completeness. In an MBT approach that focuses on test case generation from assertions, Aydal et al. [30] examined how to write useful validation scenarios for assertions in the Object Constraint Language (OCL) for the Mondex Smart Card system.

B. Evaluating assertions

Empirical studies evaluating assertions are not as numerous as those evaluating MBT, however they have shown that effort can be reduced and more commonly the reliability of programs can be increased [31], [32]. Assertions have been evaluated in the multi-threaded context resulting in fewer false negatives [33] and in the object-oriented contexts, by examining which types and what placement of assertions are effective [34]. Assertions derived from formal specifications have been shown to be effective when evaluated on real-world systems injected with faults if these faults adversely affected program state [35]. Empirical evidence for assertions in commercial systems is even more scarce, however Microsoft performed a study that showed increased assertion density resulted in programs with reduced fault density [36]. Additionally assertions resulted in detection of a large percentage of faults in the bug database. The automation of annotation development has shown promising preliminary results with a tool called Houdini [37].

VII. CONCLUSION

In this work, we consider the case study of a cruise controller simulator program to show how bugs in the SUT can be better detected if simple conformance testing is supplemented with (a) oracles in the test driver, and (b) assertions independently introduced by the developers in the SUT. We empirically evaluate this approach by introducing 15 bugs in the SUT; 3 out of 11 non-equivalent bugs get detected by the oracles in the test driver and 6 out of the 11 bugs were detected by assertions.

Similar to the majority of the studies reviewed by Neto et al. [15], the case study presented in this paper deals with a relatively small example with seeded bugs that do not serve as a complete set of defects that may occur in the program development; however, it is difficult to derive a characterisation of all possible bugs. Furthermore, unlike Neto et al.'s studies, our case study did not detect 100% of defects and focused on the combination of MBT with implementation-based assertions.

In future work, we plan to take up an industrial case study to determine if our results can be replicated in an industrial setting and compare the effectiveness of our approach to the standard test suite. We will need to determine a metric to evaluate how difficult it is to incorporate assertions in the SUT, and perform a cost-effectiveness study. Moreover, we plan to study how assertions can be generated systematically.

ACKNOWLEDGMENTS

We acknowledge Nguyen Hoai Duc for sharing his previous experiences with SpecExplorer (an earlier MBT tool from Microsoft Research) and Qtronic. Additionally we wish to thank Nguyen Hoai Duc as well as Niusha Hakimipour for their assistance in seeding defects for the study.

REFERENCES

- [1] I. El-Far and J. Whittaker, *Encyclopedia on Software Engineering*. Wiley, 2002, ch. Model-based software testing.
- [2] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer," *Lecture Notes in Computer Science*, vol. 4949/2008, pp. 39–76, 2008.
- [3] S. Dalal, A. Jain, and J. Poore, "ICSE 2005 Workshop Advances in model-based testing (A-MOST)," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 1–3, 2005.
- [4] A. Hartman, "Model based test generation tools." AGEDIS report, Tech. Rep., November 2005.
- [5] L. Briand, M. D. Penta, and Y. Labiche, "Assessing and improving state-based class testing: a series of experiments," *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 770–793, 2004.
- [6] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [7] D. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.
- [8] J. Jacky, M. Veanes, C. Campbell, and W. Schulte, *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
- [9] B. Meyer, *Object-oriented software construction*. Prentice Hall, 2000.
- [10] M. S. Feather and B. Smith, "Automatic generation of test oracles: From pilot studies to application," in *Fourteenth IEEE Automated Software Engineering Conference (ASE-99)*, 1999, pp. 63–72.
- [11] K. Mei-Ko, "Graphic programming using odd or even points," *Chinese Mathematics*, vol. 1, pp. 273–277, 1962.
- [12] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. Wiley, 1999.
- [13] P. Nienaltowski, B. Meyer, and J. S. Ostroff, "Contracts for concurrency," *Form. Asp. Comput.*, vol. 21, no. 4, pp. 305–318, 2009.
- [14] M. Katrib, D. Fernandez, and E. Pimentel, "Synchronizing java threads using assertions," in *31st International Conference on Technology of Object-Oriented Language and Systems*, 1999, p. 51.
- [15] A. Neto, R. Subramanyan, M. Vieira, G. Travassos, and F. Shull, "Improving evidence about software technologies: A look at model-based testing," *Software, IEEE*, vol. 25, no. 3, pp. 10–13, May-June 2008.
- [16] S. Kansomkeat, J. Offutt, A. Abdurazik, and A. Baldini, "A comparative evaluation of tests generated from different UML diagrams," in *Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008 (SNPD '08)*, Aug. 2008, pp. 867–872.
- [17] M. Botincan and V. Novakovic, "Model-based testing of the conference protocol with spec explorer," in *9th International Conference on Telecommunications, 2007 (ConTel 2007)*, June 2007, pp. 131–138.
- [18] P. Santos-Neto, R. F. Resende, and C. Pádua, "An evaluation of a model-based testing method for information systems," in *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 770–776.
- [19] Q. Xie and A. Memon, "Model-based testing of community-driven open-source GUI applications," in *22nd IEEE International Conference on Software Maintenance, 2006 (ICSM '06)*, Sept. 2006, pp. 145–154.
- [20] P. Wu and H. Lin, "Model-based testing of concurrent programs with predicate sequencing constraints," in *Fifth International Conference on Quality Software, 2005 (QSIC 2005)*, Sept. 2005, pp. 3–10.
- [21] J. Boberg, "Early fault detection with model-based testing," in *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, 2008, pp. 9–20.
- [22] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One evaluation of model-based testing and its automation," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 392–401.
- [23] E. Bringmann and A. Kramer, "Model-based testing of automotive systems," in *2008 1st International Conference on Software Testing, Verification, and Validation*, April 2008, pp. 485–493.

- [24] S. Kandl, R. Kirner, and P. Puschner, "Development of a framework for automated systematic testing of safety-critical embedded systems," in *2006 International Workshop on Intelligent Solutions in Embedded Systems*, June 2006, pp. 1–13.
- [25] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, and B. Hasling, "Applying model-based testing to healthcare products," in *ACM/IEEE 30th International Conference on Software Engineering, 2008 (ICSE '08)*, May 2008, pp. 669–672.
- [26] B. S. Andaloussi and A. Braun, "A test specification method for software interoperability tests in offshore scenarios: A case study," in *International Conference on Global Software Engineering, 2006 (ICGSE '06)*, Oct. 2006, pp. 169–178.
- [27] E. Cartaxo, F. Neto, and P. Machado, "Test case generation by means of UML sequence diagrams and labeled transition systems," in *IEEE International Conference on Systems, Man and Cybernetics, 2007 (ISIC'07)*, Oct. 2007, pp. 1292–1297.
- [28] K. Sakurai, P. Bokor, and N. Suri, "Aiding modular design and verification of safety-critical time-triggered systems by use of executable formal specifications," in *11th IEEE High Assurance Systems Engineering Symposium, 2008 (HASE 2008)*, Dec. 2008, pp. 261–270.
- [29] E. G. Aydal, R. F. Paige, M. Utting, and J. Woodcock, "Putting formal specifications under the magnifying glass: Model-based testing for validation," in *International Conference on Software Testing Verification and Validation, 2009 (ICST '09)*, April 2009, pp. 131–140.
- [30] E. Aydal, R. Paige, and J. Woodcock, "Observations for assertion-based scenarios in the context of model validation and extension to test case generation," in *IEEE International Conference on Software Testing Verification and Validation Workshop 2008 (ICSTW '08)*, April 2008, pp. 11–20.
- [31] B. Korel, Q. Zhang, and L. Tao, "Assertion-based validation of modified programs," in *International Conference on Software Testing Verification and Validation (ICST '09)*, April 2009, pp. 426–435.
- [32] M. Muller, R. Typke, and O. Hagner, "Two controlled experiments concerning the usefulness of assertions as a means for programming," in *International Conference on Software Maintenance*, 2002, pp. 84–92.
- [33] A. Dantas, F. Brasileiro, and W. Cirne, "Improving automated testing of multi-threaded software," in *1st International Conference on Software Testing, Verification, and Validation*, April 2008, pp. 521–524.
- [34] M. Satpathy, N. Siebel, and D. Rodriguez, "Assertions in object oriented software maintenance: analysis and case study," in *20th IEEE International Conference on Software Maintenance*, Sept. 2004, pp. 124–133.
- [35] D. Coppit and J. Haddox-Schatz, "On the use of specification-based assertions as test oracles," in *29th Annual IEEE/NASA Software Engineering Workshop*, April 2005, pp. 305–314.
- [36] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical investigation," in *17th International Symposium on Software Reliability Engineering (ISSRE '06)*, Nov. 2006, pp. 204–212.
- [37] C. Flangan, K. Rustan, and M. Leino, "Houdini, An Annotation Assistant for ESC/Java," *Lecture Notes in Computer Science*, vol. 2021/2001, pp. 500–517, 2001.

APPENDIX

The code listings for Controller and SpeedControl are presented on the following pages including bugs, assertions and line numbers.

```
1  #define DEBUG
2
3  using System;
4  using System.Diagnostics;
5  using System.Collections;
6  using System.Data;
7  using System.Text;
8
9  namespace CruiseControllerImpl{
10
11  public class Controller {
12      static int INACTIVE = 0; // cruise controller states
13      static int ACTIVE = 1;
14      static int CRUISING = 2;
15      static int STANDBY = 3;
16      private int controlState = INACTIVE; //initial state
17      private SpeedControl sc;
18      private Boolean isfixed;
19
20      public Controller(CarSpeed cs, Boolean b){
21          sc=new SpeedControl(cs);
22          isfixed=b;
23      }
24
25      public void brake(){
26          Debug.Assert(controlState == CRUISING);
27          if (controlState==CRUISING ) {
28              sc.disableControl(); controlState=STANDBY;
29          }
30          Debug.Assert(controlState == STANDBY);
31      }
32
33      public void accelerator() {
34          Debug.Assert(controlState == CRUISING);
35          if (controlState==CRUISING ){
36              sc.disableControl(); controlState=STANDBY;
37          }
38          Debug.Assert(controlState == STANDBY);
39      }
40
41      public void engineOff(){
42          Debug.Assert(controlState != INACTIVE);
43          if(controlState!=INACTIVE) {
44              if (controlState==CRUISING) sc.disableControl();
45              controlState=INACTIVE;
46          }
47          Debug.Assert(controlState == INACTIVE);
48      }
49
50      public void engineOn(){
51          Debug.Assert(controlState == INACTIVE);
52          if(controlState==INACTIVE) {
53              sc.clearSpeed();
54              controlState=ACTIVE;
55          }
56          Debug.Assert(controlState == ACTIVE && sc.getSpeed() == 0);
57      }
58
59      public void on(){
60          Debug.Assert(controlState != INACTIVE);
61          if(controlState!=INACTIVE){
62              sc.recordSpeed(); sc.enableControl();
63              controlState=CRUISING;
64          }
65          Debug.Assert(controlState == CRUISING);
66      }
67
68      public void off(){
69          Debug.Assert(controlState == CRUISING);
70          if(controlState==CRUISING )
71              {sc.disableControl(); controlState=STANDBY;}
72          Debug.Assert(controlState == STANDBY);
73      }
74
75      public void resume(){
76          Debug.Assert(controlState == STANDBY);
77
78          if(controlState==STANDBY)
79              {sc.enableControl(); controlState=CRUISING;}
80          Debug.Assert(controlState == CRUISING);
81      }
82
83      public int getControlState(){
84          return controlState;
85      }
86  }
87 }
88
```

bug 2: replace controlState = STANDBY by controlState = ACTIVE

bug 6: delete line 43 and 46

bug 11: remove if

bug 12: insert this statement as the last statement in the preceding if block.

bug 3: delete line 53

bug 13 : change != INACTIVE to == ACTIVE

```

1  #define DEBUG
2
3  using System;
4  using System.Diagnostics;
5  using System.Threading;
6
7  namespace CruiseControllerImpl{
8
9      public class SpeedControl: Object {
10         static int DISABLED = 0; //speed control states
11         static int ENABLED = 1;
12         int state = DISABLED; //initial state
13         int setSpeed = 0; //target cruise control speed
14         Thread speedController;
15         CarSpeed cs; //interface to control speed of engine
16
17         public SpeedControl(CarSpeed cs){
18             this.cs=cs;
19         }
20
21         public void recordSpeed(){
22             lock(this) {
23                 setSpeed=cs.getSpeed();
24             }
25         }
26
27         public void clearSpeed(){
28             lock(this) {
29                 Debug.Assert(state == DISABLED);
30                 if (state==DISABLED) {setSpeed=0;}
31                 //sc.disableControl();
32                 Debug.Assert(setSpeed == 0);
33             }
34         }
35
36         public int getSpeed(){
37             lock(this) {
38                 return setSpeed;
39             }
40         }
41
42         public void enableControl(){
43             lock(this) {
44                 int prestate = state;
45                 Thread prethread = speedController;
46                 if (state==DISABLED) {
47                     speedController= new Thread(new ThreadStart(run));
48                     speedController.Start();
49                     Debug.Assert(speedController != null); // inline assertion for bug 1
50                     state=ENABLED;
51                 }
52                 Debug.Assert(state == ENABLED && ((prestate == ENABLED && speedController == prethread) ||
53                     (prestate == DISABLED && speedController != prethread)));
54             }
55         }
56
57         public void disableControl(){
58             lock(this) {
59                 if (state==ENABLED) {state=DISABLED;}
60                 Debug.Assert(state == DISABLED);
61             }
62         }
63
64
65         public void run() { // the speed controller thread
66             try {
67                 while (state==ENABLED) {
68                     Thread.Sleep(500);
69                     if (state==ENABLED) lock(this) {
70                         double error = (float)(setSpeed-cs.getSpeed())/6.0;
71                         double steady = (double)setSpeed/12.0;
72                         cs.setThrottle(steady+error); //simplified feedback control
73                     }
74                 }
75             } catch (Exception e) { Console.WriteLine(e.StackTrace);}
76         }
77     }
78 }
79
80
81

```

bug 4: delete line 30 and
replace with setSpeed=0;

bug 5: add sc.disableControl(); after line 30

bug 14: remove locking lines 37 and 39

bug 7: replace line 38 with: if
(state==DISABLED) return setSpeed; else
return setSpeed*6;

bug 15: add additional condition: if (state==DISABLED) to lines 43 and 54

bug 8: remove if condition lines 46, 51

bug 9: replace 6 by 12

bug 10: replace 12 by 6

bug 1: include speedController = null;
after line 75