QuickCheck-Style Testing of Embedded Software using the PropEr Framework

Shivani Raina
Abstract

QuickCheck-Style Testing of Embedded Software using the PropEr Framework

Shivani Raina

Testing is an integral part of any software development. It is important for improving the quality of any product. There are several ways of software testing and one of the most common approaches being random testing. One way of realising random testing is to write generators which can produce suitable input data with the help of inbuilt and/or user defined data types. In this thesis, we have explored this method with the help of QuickCheck. QuickCheck is one of the first frameworks to support random testing written in Haskell. It has been subsequently ported to many other languages, including Erlang, Scala, and C. In this thesis, we have provided a similar framework for reactive embedded software (i.e., system testing) using Erlang version of QuickCheck called PropEr, where the test generator has to create complex program input streams in an online manner. The goal of this thesis is to create a framework by combining an existing QuickCheck implementation for Erlang called PropEr with WindRiver Simics which is an execution platform for embedded software. It involved design and implementation of an interface between the QuickCheck framework and the execution platform. This approach has been verified with the help of a case study. We have chosen an elevator case study for testing our implementation. We have evaluated the framework using the simulation of an elevator system, verifying a number of safety properties by systematic testing. It can be concluded that testing an embedded system using this embedded interface for QuickCheck is possible. There are many improvements that can be made to the interface with respect to providing more generic functions to evaluate the properties of embedded systems.
Acknowledgements

I would like to thank my supervisor Philip Rummer for providing me his guidance from time to time as needed. Helping me to structure the work and my presentation. I am really thankful to him for providing me freedom to give my inputs as the thesis was carried forward.

I would also like to thank my reviewer Bengt Jonsson for providing his support and valuable comments which led to the betterment of this project.

Finally, I would like to thank everyone else who were a small but important part of my thesis who from time to time gave some valuable comments which in turn helped me develop a better project. Last but not the least, I wish to thank my incredible parents who supported me during the project and helped me reach my goal.
Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of contents</td>
<td>iv</td>
</tr>
<tr>
<td>List of figures</td>
<td>vi</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background</td>
<td>2</td>
</tr>
<tr>
<td>2.1 Testing methodology</td>
<td>2</td>
</tr>
<tr>
<td>2.2 Testing of Reactive Systems</td>
<td>3</td>
</tr>
<tr>
<td>2.3 QuickCheck style testing</td>
<td>4</td>
</tr>
<tr>
<td>2.4 PropEr- A QuickCheck implementation- An overview</td>
<td>5</td>
</tr>
<tr>
<td>2.4.1 Property writing</td>
<td>5</td>
</tr>
<tr>
<td>3 PropEr and testing of reactive systems</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Characteristics of reactive systems</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Testing reactive systems</td>
<td>8</td>
</tr>
<tr>
<td>3.2.1 PropEr framework for Embedded Systems</td>
<td>9</td>
</tr>
<tr>
<td>4 Simulation environment for testing RealTime systems</td>
<td>11</td>
</tr>
<tr>
<td>5 Embedded software elevator case study</td>
<td>14</td>
</tr>
<tr>
<td>5.1 Overview</td>
<td>14</td>
</tr>
<tr>
<td>5.2 Safety for elevator</td>
<td>14</td>
</tr>
<tr>
<td>6 Implementation of the testing interface</td>
<td>19</td>
</tr>
<tr>
<td>6.1 Architecture</td>
<td>19</td>
</tr>
<tr>
<td>6.2 Technical details with an example</td>
<td>20</td>
</tr>
<tr>
<td>6.2.1 Evaluation function</td>
<td>20</td>
</tr>
<tr>
<td>6.2.2 Property execution and input file format</td>
<td>21</td>
</tr>
<tr>
<td>6.2.3 Parse the input file and generate output file</td>
<td>22</td>
</tr>
<tr>
<td>6.2.4 Property evaluation</td>
<td>23</td>
</tr>
<tr>
<td>7 Related work</td>
<td>24</td>
</tr>
<tr>
<td>8 Conclusions and future work</td>
<td>25</td>
</tr>
<tr>
<td>References</td>
<td>26</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example of property and a program</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Generic reactive control system</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>IO relation of a reactive system</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Real time simulation environment with PropEr testing interface</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>A property example for embedded systems</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>LED status output on Simics console</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>Example of an elevator</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>Mapping of input-outputs for elevator case study</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>Elevator case study: Property 1</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>Elevator case study: Property 2</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>Elevator case study: Property 3</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>Elevator case study: Property 4</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>Elevator case study: Evaluation</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>Elevator case study: Evaluation write function</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td>Architecture of PropEr interface</td>
<td>19</td>
</tr>
<tr>
<td>16</td>
<td>Property writing and evaluation, Example</td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td>File format generated by PropEr to be read by Simics</td>
<td>21</td>
</tr>
<tr>
<td>18</td>
<td>Example of input file for elevator case study</td>
<td>22</td>
</tr>
<tr>
<td>19</td>
<td>File format from python parser to be read by PropEr</td>
<td>22</td>
</tr>
<tr>
<td>20</td>
<td>Example of output file from elevator case study</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>Results of a test run, Elevator case study</td>
<td>23</td>
</tr>
</tbody>
</table>
1 Introduction

It is important to verify an implemented software with respect to the predefined specifications. Wrong software implementations can lead to great loss like, loss of money and time. In case of embedded real time systems, it can lead to huge losses as the timing and safety constraints are very critical for ensuring the correctness of the software. To avoid this, software testing is highly recommended for any embedded software. One of the conventional ways of testing any software is manual testing. Manual testing involves creating a series of test inputs and corresponding output i.e. behaviour of the system. The system is executed for a series of inputs and is verified against the expected results. This is the straightforward method of testing a software program. However, this approach of testing software is tedious and does not guarantee that the system is working completely as per the requirements.

Another approach which can be used to test the system is property-based testing. The tester specifies the set of generators which create inputs and certain properties which define the input-output relationship of the system. A property-based testing tool applies this input to the system under test as it executes and ensures that provided input-output relationship is satisfied. PropEr (PROPerty-based testing tool for ERlang) is such a property based testing tool which tests programs written in the Erlang language. The focus is majorly on testing pure functions. Standard type annotation language incorporated in Erlang programming language to include type information is majorly required by PropEr tool. Here, programmers can write signatures for their functions, which is highly recommended since it leads to program documentation. The tester provides the properties to the tool which must hold for their programs on every execution. A random set of tests are executed on the program to falsify these properties. A set of predefined and/or user-defined types are also specified for each property.

The topic of this thesis is to explore the possibility to provide a similar framework for reactive embedded software (i.e., system testing), where the test generator has to create program inputs following a predefined protocol. These input sequences has to be run on an embedded software implementation. The interface receives a set of output sequences from the embedded software. This set of output sequences will be verified against the property written in the PropEr tool to test the correctness of the embedded software under test. The goal of the thesis is to create such a framework by combining an existing QuickCheck implementation (e.g., PropEr for Erlang) with an execution platform for embedded software (e.g., a development board or a simulator). This includes writing of python scripts for the embedded system simulator. It encompasses design and implementation of an interface between the PropEr framework and the execution platform, as well extensions to the PropEr implementation.
2 Background

2.1 Testing methodology

Testing is the most basic way applied by programmers on their code to assess the quality of their product. Manual testing is time consuming and cumbersome and is usually not very cost effective. As a result, the software industry has always been looking forward to automating the testing process. System testing was one of the initial steps towards automation of testing process where small units were combined in a whole black box and is tested for its functionality.

Software testing is done to ensure that software meets its requirements. Software testing is targeted to achieve various functions like quality assurance, verification, reliability estimation, correctness of the software etc. [21] In general, testing takes most of the time within the whole software development life cycle and hence, huge budgets are allocated for its progress.

It is practically impossible to find all the possible ways in which a software program might fail while executing. As the complexity of software increases, chances of introducing design errors and other code errors also increases. As a result, to improve the quality of software, testing of software is done at each stage of software development. This type of testing starts from unit testing, then to integration testing followed by the user acceptance testing and finally the software is sent to production. The most stable version of the software is provided to the end user. If there are still any errors found in the production installed software, these are raised as bugs and the issues are fixed in the new version of software. Once the latest version is recoded, the testing for that piece of software does not remain confined to the new additions made to the system, it has to be restarted covering all the working functionalities. This is a very important aspect of testing software as the new changes made to the software might affect the already existing functionalities and hence need to be verified for such possibilities.

Classification of testing is governed by different aspects like security, reliability or performance, if the purpose of classification is chosen. Software testing can also be divided into testing of requirements, design phase testing, evaluation testing, user acceptance testing, production implementation testing and maintenance testing. These are the phases of software design life cycle. Testing can also be divided depending on the scope of testing like unit testing, integration testing, system testing and also user testing [21]. There are various approaches to software testing. Some of them have been explained below.

Black-box testing or Specification based testing is one such method of testing software systems. Black-box testing is much simpler since it does not need to know how the system has been implemented or does not need to know the logic of the system under test. The abstract model of the system is generally considered, providing information about the functionality of the system. The system is tested and the results are compared against expected behaviour of the system under test [16]. Black box testing is also called data-driven, input-output
driven testing. In black-box testing, the inputs are given, the functionality is known to the tester, but the implementation is not taken into account by the tester. To improve the quality of software, wide range of test inputs are chosen.

**White-box testing or Structural testing** In white-box testing, implementation is visible to the tester. Test cases are framed keeping in mind the code flow. White-box testing is also called glass-box, design based or logic driven testing. The test cases are derived from the functionality, the logic and/or the programming language used for the system under test. The tester tries to choose the test cases in such a way that all the nodes or paths are covered at least once. The tester has to ensure that he/she has covered all the visible paths in the system.

**Gray-box testing or Functional and Structural testing** is a combination of black-box and white-box testing [19]. Thus, specification against the program’s functionality as well as the logical paths of the system code is checked as well. There are many uses to Gray-box testing, for example, avoiding repetitive testing of similar functionality code. If a small piece of code is called at many places, then, this piece of code can be tested only once instead of testing at each place where it is called in the software program. It can save a lot of time as well as money.

**Model-based testing** Models can be used to describe the functional behaviour of the systems under test [29, 12]. The model can be an abstract representation of the system. A set of test cases are derived depending on the functional behaviour of the model in the form of say, a state chart or a state flow diagram. In some model-based testing tools, model is close enough to the system under test and can test the system functionality to a larger extent. Model based testing is usually based on heuristics. These test cases contain information about the model functionality and/or test pass/fail criteria of the system under test.

**Property based testing** Property-based testing is a high level approach to testing in the form of abstract definitions which the system must satisfy universally, for any test data generated by the testing library [23]. In this way, code can be tested against thousands of tests which otherwise would be infeasible to write manually, often results in finding hidden bugs or errors in the system which otherwise might get unnoticed. The tester has to be aware of the specification written by the programmer. The knowledge of how the system has been implemented is not necessary, hence, it is a form of random black-box testing.

### 2.2 Testing of Reactive Systems

As the name suggests, systems which are constantly interacting with environment and other reactive systems and responding with respect to it are called
reactive systems. The messages or events need to be sent and received from the other systems. There are two important characteristics of reactive systems [3]:

1. The system must react to the signals from the environment.

2. The system must respond to the environment in specified amount of time. If the specified time is elapsed, the system requirement fails even if the response is correct.

The testing for reactive systems starts from the very beginning phase of development. The system is tested rigorously with the variety of test inputs produced by the testing unit. The real time system must produce consistent results for all the possible combinations of inputs. When a test case fails, the history of behaviour of the system is checked for that particular test case to reach to the root cause of the problem. Once rectified, the system has to undergo all the test cases again to ensure there are no side effects to other parts of the system.

2.3 QuickCheck style testing

QuickCheck style of testing is random testing of software. The tester needs to provide just the structure of the valid inputs to the system under test and state properties which must hold for all the valid inputs provided. The tester needs to be aware of the functionality of the system under test but the implementation techniques need not be known. Let us formulate an example of such property based testing. Consider a simple program written in the Erlang language which deletes a given integer from a list of integers as shown in figure 1. Now to validate the program, we can define a property as: Check for the deleted integer in the new list, if not available, pass the test case, if available present the failed set of inputs. The detailed explanation of the program is given in further sections.

Such an approach to software testing might lead to less time consumption. It takes comparatively a fraction of second to specify the valid input types and write the properties for the system than writing hundreds of test cases.

Need of shrinking: Since the method involves random generation of complex valid inputs which are assessed against the behaviour of the system under test, when a set of input fails, it can be difficult to list the failed test case. It can be a very cumbersome task to manually extract the information about failed test case. This is what shrinking does for us.

Shrinking is a internal feature provided by these property based testing tool to provide the information about the failed test case. In general, parts of the failing test inputs is removed consecutively until the test case succeeds. In Figure 1 the shrinking process returns a list of two elements finally. This test input which is required for the failure of the test case is called the minimal test case. This can be a good starting point for debugging the cause of failure.
Implementations: There are various implementations of QuickCheck\[10\], written in Haskell, designed for testing pure functions. This tool has then been ported to a number of languages, for example, Python\[9\] and also in Erlang\[20\].

2.4 PropEr- A QuickCheck implementation- An overview

PropEr is a property-based testing tool for programs written in the Erlang programming language. Properties are written to test the specifications of the system under test using Erlang expressions and a few predefined macros. The input to these functions are specified using type language.

After the properties are written, the shrinking process as explained in Section 2.3 comes into scenario. In this process, a property is tried against each input to recreate the failure. The minimal test case is generated which lists the set of input values which caused the failure. These input streams can be saved and reused to ensure that the problem has been successfully fixed. PropEr offers a tight integration with the Erlang language types and specs with properties\[22\]. Generators can be as simple as native Erlang types. Any function specification can be directly converted to a simple property of the system under test. The other capabilities of PropEr tool have been exhaustively described in other publications \[20\]\[23\].

2.4.1 Property writing

In PropEr, we can specify properties with the help of Erlang expressions and a few predefined macros like ?FORALL, ?TIMEOUT, ?IMPLIES. A boolean expression is wrapped in these macros or wrappers to write the property to be tested for the system.

?FORALL tests the specified property for all the instances of inputs generated for the system. The input values can be the in-built Erlang type system or it can be the user-defined type system. User defined type system can be a set of predefined values or it can be a combination of in-built existing Erlang type system \[22\]. ?IMPLIES is used in combination with ?FORALL. A precondition in the form of a boolean expression is specified for each instance of input. The property is validated only when the input satisfies this precondition. ?TIMEOUT puts an extra condition for the property to be passed. The property is considered to be passed only if it passed within a given time limit as specified in ?TIMEOUT. More detailed information can be found at \[22\]. PropEr also provides functions and macros to analyze the statistics of the SUT (System Under Test). These can provide debugging information of the failed property and hence can help improve the software. ?TRAPEXIT, ?WHENFAIL are few of such macros provided by PropEr.

Figure 1 shows an example of how property can be written to test a small program using PropEr. In this example, we wish to test a function that deletes
Figure 1: Example of property and a program

an integer from a list of integers. The property prop_delete uses ?FORALL to evaluate the function delete. Let me explain the example program in detail. The `spec` describes the input-output relation of the function delete. There are four cases for one implementation of function delete. It deletes the specified integer from the list of integers. The prop_delete is the property which we wish to test for this delete function. The property uses the ?FORALL wrapper, the first parameter $X,L$ are variables storing the created inputs. The second parameter integer(),list(integer()) provides the data type for these input instances. Here, $X$ is of type integer and $L$ is the list of integers. In third parameter, function delete is provided the input instances $X, L$ and the function verifies whether $X$ is present in the list $L$ or not.

As can be observed from the figure, the property passes initially when the number of test cases specified were '10' but as the number of test cases are increased to '100', it failed showing that the function does not delete more than one occurrences of $X$ (eg, 14) in corresponding list $L$. It also shrinks the test case and provides the minimal test case for which the property fails.[6].
3 PropEr and testing of reactive systems

3.1 Characteristics of reactive systems

As described in Section 2.2, reactive systems constantly interact with the external environment. For the reactive systems to execute, some input parameter might need to be received from the external system at any point or continuously at any point of execution. Also, it is important that the system reacts to the environment in an acceptable amount of time. The amount of time which can be tolerated to receive the response from the system governs the category of reactive systems. Systems which have tight timing constraints are called hard-realtime systems while the opposite are referred to as soft-realtime systems.

Real time systems may consist of multiple sensors and actuators. The various inputs provided to a typical embedded system are sensors, buttons, GPIO (General Purpose Input Output) port pins, analog inputs to A/D (Analog to Digital) converters, reset buttons and timers to provide delay between sequence of inputs. The analog input depends on the type of input we want to simulate.

The real time systems can receive input randomly. There is no predefined sequence in which the inputs can be received by the system. This makes the response of the system tend to be very unpredictable. Thus, rigorous testing of such systems is very important. A generalised model describing a kind of real time systems is shown in figure 2.

Figure 2 shows how a reactive system has a two-way interaction with the environment. It also does not have any definite time when the intervention/events from any of the modules arrives. Hence, considering all the possible permutation and combinations of the events need to to be addressed during testing phase.

A input-output execution trace of a general embedded system can be shown as in figure 3. The sensors 1,2 and 3 (S1, S2 and S3) are the inputs while output 1 (O1) is the output. The figure depicts a scenario in which output 1 (O1) becomes high when sensors 1, 2 and 3 (S1, S2 and S3) turn high for specific
duration of time one after the other.

Figure 3: IO relation of a reactive system

3.2 Testing reactive systems

Section 3.1 gives an overview about the requirements of a real time system which need to be addressed while testing real time systems. The tool used to test real time systems should be capable of providing discrete input sequences closely resembling the input received from sensors, port pins, A/D and other peripherals. A general block diagram of what we want to achieve by a combination of real time development environment and PropEr tool is shown below in Figure 4.

Figure 4: Real time simulation environment with PropEr testing interface

In real time embedded systems, the input sequence generally consists of bits '0' and '1'. PropEr should be capable of generating such random input sequences as per the requirement of the system. Once, the tool is capable of generating such random sequence for the system, this sequence must be provided to the simulator on which the system has been simulated. The simulator is required
to create a virtual real-time environment to test a case study. Simulators have various advantages over real hardware. Using simulators tends to be more easy, lesser time consuming and lesser expensive than real hardware. After the tests have been performed, the results from the simulator are provided to the PropEr tool. The results generated from the PropEr tool and the embedded system should be evaluated. There are plenty of simulators which can be used to create an embedded case study.

3.2.1 PropEr framework for Embedded Systems

As described in Section 2.4.1 properties can be written in the PropEr tool to test functionality of the system under test. To test embedded systems in particular, I have designed an interface between PropEr tool and embedded system environment using Erlang language [8, 4, 14]. This interface lets the PropEr tool communicate with the embedded systems to perform testing operations on it. We will refer to figure 5 to explain the list of commands provided by the interface. The interface commands are written inside the ?FORALL macro. These commands are run for every set of test inputs produced by the macro.

```erlang
prop_test() ->
    ?FORALL([l,D],[proper_es:input(),proper_es:delay()]),
    begin
        start(),
        input(l,"d1"),
        delay(D),
        end1(),
        N = length1(),
        eval1(N)
    end).

eval1(N) ->
    case N of
        0 -> false;
        _ -> addoffset(10),
            P1 = not getpin("e.1") and not getpin("e.2")
            case P1 of
                true when N>1 -> eval1(N-1);
                true when N>1 -> true;
                false -> false
            end
    end.
```

Figure 5: A property example for embedded systems

start(): starts the embedded system environment for communication with the interface.
**input(I,Pin_number):** writes the generated input value I to corresponding port.pin_number, say "1" to pin d.1 of the embedded system.

**delay(n):** provides a delay of n time units say, milliseconds to the embedded system.

The random inputs, I and D are generated using specified data types input() and delay() in the module proper_es. In Erlang, it is possible to define your own set of data types such that when a variable is specified with that type, it takes values only from defined values[8]. For example, as in this case, input() can take only two values "0" or "1".

**end1():** shuts down the embedded system environment and waits for the output values to be available.

**length1():** provides the number of output samples received in one test run from the embedded system.

Now, at this point we have received the continuous output stream from the embedded system. The output data can be received by a simple polling mechanism such that the output samples are received every T time units, say 10 ms. Now, the next step is to test the received output data. In this example, we are evaluating a condition such that the pins e.1 and e.2 cannot be true at any point of execution. The function eval1() has been written to test this property P1 for the received output data. The embedded interface provides some commands to facilitate the testing of output data as listed below.

**addoffset(DT):** This can be used to traverse the output data stream. Initially, the system is at the first sample. This command moves the system by time DT, say 10 ms such that it points to the next sample.

**getpin(Pin_number):** provides the current value of Pin_number, say e.1 at which the system is pointing that time.

**getpin(Pin_number,T):** provides the value of Pin_number, say e.1 at T(100 ms).

**getanalog(Pin_number):** provides the current analog value of Pin_number, say e.3 at which the system is pointing that time.

**setoffset(T):** sets the system to the corresponding location T. Using command getpin after this command, provides the value of that pin at time T.

**getoffset():** retrieves the current time position of the system.

P1 receives the evaluated result from the logical expression using commands getpin and then the case clause checks whether the evaluation was true. This evaluation is repeated for all the output samples. In certain scenarios, where analog data is required to be tested against a property, getanalog command can be used.
4 Simulation environment for testing RealTime systems

Software written for an embedded application needs to be copied to the hardware or development boards to test its functionality and compatibility on the actual hardware. If a buggy software is burnt on the hardware, the hardware can become non-functional and can lead to permanent damage to the connected components. Such hardware can be very expensive at times to replace. There are few more advantages to simulation environment over actual hardware. Testing the software on the simulation environment saves a lot of time. It is possible to increase the speed of simulation and test the system behaviour. This is very valuable to the developer.

To avoid this, in order to test the embedded software, there are numerous simulation environments available in the market. By simulation environment, we mean that the actual hardware is replicated virtually in a possibly simpler form. The embedded software is burnt on these simulators and testing rigorously. There are two sides to this testing. One that the development board/hardware can be tested for bugs and second that embedded software can be made closer to free from elusive bugs. This also helps to reduce time-to-market time, makes the system more reliable and cost effective.

Such fast and accurate virtual system enables integration, testing and engineering approaches to altogether different level which may not be possible on actual hardware. There are plenty of simulation environments available in the market. Once the software passes the simulator testing phase, it can be burnt onto specific development boards. These boards can be connected to the host computer using the JTAG interface to test the software on primitive hardware set up.

The choice of execution environment for our case study is guided by the following factors:

1. First of all, the chosen environment should be able to replicate the targeted development board.
2. Secondly, the simulation environment should be able to communicate with the PropEr tool.

Below, I am listing a few of the various simulation environments available on the market:

1. Keil uVision,
2. WindRiver's Simics environment.
**Keil uVision**  uVision IDE and Debugger has been developed by ARM[2]. It combines project management, compilation, developing source code, debugging and complete simulation. The development platform provided by uVision helps the user to create the projects and embedded programs. The development and debugging are integrated in a single tool for embedded project development.

There is an active support and documentation about Keil APIs. Keil uVision tool can combine the simulation environment to work with the development boards by making simple changes to the project variables. The code can be burnt and run on the development boards which is very useful in embedded systems as it allows the programmer to test the code on real hardware and not only with the simulation.

**Wind River’s Simics environment**  Wind River Simics is one of the very powerful simulators[1, 13, 7]. Simics can simulate any target hardware ranging from a single processor to the other electronic systems connected to this target hardware. By using Simics, the project risks can be reduced to a larger extent and hence, improving the quality of the system. Also, the simulation is fast enough for developer to use it interactively.

Simics has a feature called checkpointing which can store a state and use it at a later stage. Thus, the system state is deterministic which means any state of the system can be reached at any point in time. It is possible to provide the initial state to the system which will give similar results each time. Hence, the simulator is repeatable and reliable. It is also possible to inject faults in the system in Simics. Hence, a small part of the system can be tested for failures and not the whole system needs to be changed. It is possible to communicate with the environment using ethernet [28, 27].

```
running> gpio.status
Status of gpio [class ep93xx_gpio]
--------------------------------------------------------
Green LED : on
Red LED : on
running> gpio.status
Status of gpio [class ep93xx_gpio]
--------------------------------------------------------
Green LED : off
Red LED : off
running> gpio.status
Status of gpio [class ep93xx_gpio]
--------------------------------------------------------
Green LED : on
Red LED : on
running> gpio.status
Status of gpio [class ep93xx_gpio]
--------------------------------------------------------
Green LED : off
Red LED : off
```

**Figure 6:** LED status output on Simics console

Simics-TS7200 target board is one of the embedded system board available on Simics. Figure [3] shows the result of a small program written to blinking a red and green LED on TS7200 target board by writing different values to the
LED’s respective memory addresses. It is a simple program which runs hundred times and turns LEDs ‘on’ and ‘off’ alternately after every 1 sec. The change in the LED’s status can be observed on Simics console as shown in figure 5. On the Simics console, the status can be checked by using commands gpio.status. As can be seen in the figure, the LEDs are turned ‘on’ and ‘off’ alternately.
5 Embedded software elevator case study

5.1 Overview

We have planned a simple elevator case study to test it for different inputs and validation of timing constraints. Since Simics license for creating the case study completely was not available, I have created a simple elevator using python script which acts as a parser to the input file and produces a output file by running these inputs on the elevator case study. I have designed a simple elevator for three floors as shown in figure 7.

In the elevator, buttons 1, 2, 3 are available to go to respective floors. There is also a stop button which is available to the user to stop the elevator. This button can be pressed any time. The elevator must be stopped immediately as soon as the stop button is pressed. An elevator should have two sensors, the first one indicates if the elevator is on a floor or in between two floors, the second one indicating if the elevator’s door is closed or not. The elevator is moved up and down by a motor. The motor is controlled by two signals, one for moving the elevator up and one for moving it down. There is also a output pin which provides the current position of the elevator. We have mapped these input output values to the pins of the development board. This mapping can be changed depending on the requirement of the development board. Figure 8 shows the mapping for this particular case study. The goal of designing this case study is to implement and test the software controlling the elevator using PropEr interface. This script reads the input file, receives the input sequences and runs it through the elevator case study. It then receives the output sequences and sends it back to the PropEr embedded interface.

5.2 Safety for elevator

An elevator is a safety-critical system, hence, we need to formulate safety requirements for the system. We have formulated some safety requirements for our elevator case study [26].

Safety requirements:

1. If stop button is pressed, the elevator motor should stop in 1 second.
2. The motor indicators for upwards and downwards movement are active one at a time.

3. The elevator must not pass the end positions of the floors.

4. The elevator reaches the first floor within 10 seconds after pressing the button for first floor.

These safety requirements will be tested using the PropEr interface developed for the embedded system testing. I will now explain how these properties can be tested using embedded interface provided in this thesis. There can be different ways in which the properties can be formulated, this is one possible way of formulating it.

**Property 1: If the stop button is pressed, the motor is stopped within 1 second:** In figure 9, eval1(N,N1) evaluates the property 1. The arguments passed to it N and N1 is the number of output samples produced by the simulation run and the number of samples we wish to evaluate the results for. Here, we wish to evaluate the result for 1 second, so, N1 is 100 considering the sampling rate to be 10 milliseconds. The getoffset() command stores the current offset value for later reuse. Now, if e.4 returns true using getpin() command is true for any sample, we try to evaluate the property for each sample and store the result in Eval1_res. This is checked using function pin_occur. The pin_occur function loops through the samples and checks whether the property P1 was true at any instance. If true, the stored offset is retrieved back and next output entry for pin e.4 is evaluated. If false, then the property fails. The addoffset() command is...
used to traverse the output samples to receive the values of specified pins using getpin() command.

Property 2: The motor signals for upwards and downwards movement are not active at the same time: In figure 10, eval2() function traverses the whole stream of output samples using addoffset() command and evaluates the property for each instance. Once failed, the property code is false and hence, the property fails.

Property 3: The elevator may not pass the end positions, that is, go through the roof or the floor: In figure 11, eval3(), similar to eval2
function traverses through the whole stream of output samples using addoffset() command and evaluates the property for each instance. The getanalog() command retrieves the position of elevator at that point in time and value is stored in Pos. Pos is evaluated for respective floor positions, 20 cms for floor 3 and 0 cms for floor 1. As the property specifies, the elevator must move within these boundary positions. If the elevator passes these positions, the property code returns false and hence, the property fails.

Figure 11: Elevator case study: Property 3

Property 4: The elevator reaches first floor within 10 seconds after pressing the button: In figure 9, eval4(N,N1) evaluates the property 4. The implementation is similar to property 1. N1 here is 1000, since we need to evaluate the output stream for 10 seconds.

Figure 12: Elevator case study: Property 4
Evaluation function: The properties described above must be evaluated for each instance of output value produced from the simulation environment. These properties are wrapped into a single evaluation function. The overall evaluation function written to evaluate these properties is shown in figure 13. The function creates a list of input values and pin numbers randomly and writes it to the input file using function writeip as shown in figure 14. The writeip function is required such that all input instances are written into a input file before sending to the simulation environment. The writeip function traverses through the lists and creates proper inputs using each value. Once the input file is created, the end1 command is used to send the file and retrieve the output stream of data. This output stream of data is evaluated using eval functions as described above. After each evaluation is complete, the setoffset command sets the cursor to the beginning of the output stream such that the next evaluation begins at the start of the data.

The pin_occurs function could be converted into a generic macro, say, ?will_happen which accepts a list of pins and evaluates the property without being repeated for each property. Also, function eval2() and eval3() can be converted into a generic macro, say, ?bound_forever such that any property provided to this macro has to evaluate true for each instance of output data stream.
6 Implementation of the testing interface

6.1 Architecture

The overall architecture for this implementation is shown below in figure 15. The second level "Low level API" is written to establish a complete cycle of communication from PropEr side to the simulation environment side. An example of such communication is shown in figure 16. This is converted into a lower level format by the PropEr interface implemented as part of this thesis. The lower level follows a particular format for files being used for communication as shown in figure 15.

![Figure 15: Architecture of PropEr interface](image)

The most high level of the architecture can be devised which wraps the second level into more formal way of presenting the property. This means instead of the user defining loops to check certain port pins, we can just provide these pin numbers to these wrappers and they get expanded to this lower level which loops over these port pins. For example, ?FOREVER provided with specific pin numbers will get converted to second level and will perform this task for the whole system execution, ?SINCE will provide a signal if the event specified in the form of a property has occurred since some specified amount of time.
6.2 Technical details with an example

Below is provided a complete cycle of how the testing occurs using this PropEr framework. The testing interface comprises of the following parts as listed below.

1. Evaluation function for testing embedded system.
2. Run the property and send the input file generated by PropEr to the embedded system under test.
3. Read the input file generated in Step 2, run the input sequence on the case study and provide the output file with a defined format.
4. The output file produced from the interface is sent back to PropEr. The evaluation function uses the output file to testify the property.

6.2.1 Evaluation function

The functions described in Section 3.2.1 are used within a property to send the input file to the embedded system and receive the corresponding outputs from it. These outputs can be further evaluated and tested using output functions provided by the interface. An example of such property and evaluation function is described in figure 16.

The function prop_testled() uses macro ?FORALL to test the property written in eval1() for all the inputs I1, I2, D generated randomly from type inputs input() and delay() defined in module properes. The start command starts the embedded system and prepares an input file to be sent to the embedded system simulator. The command input provides input I1 and I2 to port pins d.1 and d.2 respectively. The delay provides a randomly generated delay input D to the embedded system. This delay is required to let the embedded system run for certain amount of time D. The command end1 ends the embedded system once the set of input and delay commands have been performed on the system. The end1 command also receives the set of output data from the simulator and sends it back to the PropEr interface. The PropEr interface now is ready to analyse the output data for the expected properties.

To traverse through the entire set of output data sampled at certain amount of time, the command lengthI provides the number of output data samples we have received during this run. Now, we call the function eval1 with the number of output data samples say, N. Now, in eval1, we are traversing through each set of output data using command addoffset. This command increments the state of the system to point to the next output set of data. The commands getpin provides the value of the specified port pin, say, e.1, e.2 and e.3 for current set of output data.
6.2.2 Property execution and input file format

The general file format generated from PropEr tool to be read by Simics is shown in figure 17. The figure also depicts an example of such an input file.

```
prop_testled() ->
  FORALL(I1,I2,0),
  (proper_cs:input(),proper_cs:input(),proper_cs:delay()),
begin
  start(),
  input(I1,"d.1"),
  delay(0),
  input(I2,"d.2"),
  delay(0),
  end1(),
  N = length1(),
  eval1(N)
end).

eval1(N) ->
case N of
  0 -> false;
  _ -> addoffset(10),
      P1 = not getpin("e.1") and not getpin("e.2"),
      P2 = not getpin("e.1") and not getpin("e.3"),
      P3 = not getpin("e.2") and not getpin("e.3"),
      Res = P1 or P2 or P3,
      case Res of
        true when N>2 -> eval1(N-1);
        true when N=2 -> true;
        false -> false
      end
end.
```

Figure 17: File format generated by PropEr to be read by Simics

One such example of input file for our elevator case study is shown below in figure 18. This case study example is a bit different from the case study explained in Section 5. Here, we provide the input to the elevator motor through pin d.1 and d.2. A '1' at d.1 moves the elevator upwards while a '1' at d.2 moves the elevator downwards. During delay command, the previous run command executes moving the elevator up or now depending on the input value.

The file as it shows has a set of input to pin d.1 and d.2 which will be executed during the respective delay commands following it. When d.1 is high,
the elevator starts moving upwards and when d.2 is high, the elevator starts moving downwards. When both are low, the elevator does not move and remains still.

6.2.3 Parse the input file and generate output file

After the files are provided to Simics or the python script in our case, the output for each input command will be provided to the user so that it can be compared against the PropEr evaluation. A general format of such a file is shown in figure 19. Also, the example output file is shown in the figure.

An example output file for our case study is shown below in figure 20.

An output file is generated with the time stamp on it. The output file has a list of output pins which correspond to the position of elevator. Pin e.1, e.2 and e.3 corresponds to floor 1, 2 and 3 respectively. A high at any pin shows that elevator is at that floor. None of the pins can be high at the same time.
6.2.4 Property evaluation

Once the PropEr interface has received the output file, it uses the evaluation function as shown in figure 16. The output of a couple of such test runs is shown below in figure 21.

The `eval1()` evaluation function as described in figure 16 uses the commands `addoffset(N)` and `getpin(Pin_number)` to evaluate that the elevator is at one floor at a time. The `eval1()` function browses through the whole list of output values generated for that set of inputs and checks the condition. This condition is evaluated for each output sample. If the property evaluates true, the PropEr gives that message. When the property does not evaluate true, PropEr tool applies the shrinking mechanism and finds the minimal test case for which the property fails as shown in figure 21.
7 Related work

QuickCheck has been used in various case studies to evaluate the random test case generation approach. One such case study was carried out to test an industrial implementation of Megaco protocol [5]. Quviq QuickCheck was satisfactory in finding the faults with the application. QuickCheck was applicable to black box random testing of the synchronous part of real telecommunication protocols.

One typical usage of lightweight formal methods and property based testing of QuickCheck was done to test a radiotherapy support system [30]. Properties were written to test and verify the system. It was useful in random generation of test inputs and also to reduce the complexity of failed test cases. The use of QuickCheck to help attain the high demands on software validation was achieved. QuickCheck has also be used to test Monadic code[11].

Another automatic testing style for reactive systems is using GAST [18, 17]. GAST is a fully automatic test system. In this system, appropriate test values are generated for the specified logical property and results are evaluated after execution. Reactive systems are usually represented as models instead of property. In this paper, model based testing and logical based testing were integrated in a single automatic system.

There has been various papers which address the requirement of automation of testing for reactive systems. One of such papers propose the use of synchronous observers so that both the correctness and the relevance of the test cases can be determined [25]. LURETTE is one such prototype which has been developed to work on observers in the LUSTRE programming language.

One way of online test case generation for automated test oracles has been discussed here [15]. The test oracles report the behaviour of the system depending on the model of the system under test. In this paper, it tests the safety requirements which are described in temporal logic. The logic expresses quantitative timing properties. This technique has been used in cruise control module and throttle module of Volvo Technical Development Corporation. This paper provides a translation which can automatically construct test oracles from specifications.
8 Conclusions and future work

In this thesis, an interface for testing embedded software has been implemented. The interface is responsible for interacting between PropEr and an Embedded software simulation environment. The communication follows a predefined protocol to maintain uniformity.

The thesis contributes to improve testing coverage of embedded software by exploiting the use of randomized property based testing for embedded software. Using random property based testing can improve the testing coverage especially in the areas of black box testing.

The interface implementation has been tested using the elevator case study. The communication with the case study, retrieving the results and then testing the property for the case study gives appropriate and expected results. I can say that the set up for testing a embedded software case study is achievable with this implementation.

The major obstacles in carrying the thesis was the simulation environment availability. The licenses took more than two months to be available. The installation of Simics environment for TS7200 embedded environment is not complete and needs a lot of work to be done. The different packages like modelling were not installed and hence, took lot of time to be ready for use.

The long term work for this thesis would be to develop python scripts to communicate with simulation environment. A good model of simulation environment for testing embedded software on Simics can also be developed. A more high level language can be implemented to write properties for testing embedded software.
References


