

2015

# Towards DO-178C compatible tool design

Yijia Xu

*Iowa State University*

Follow this and additional works at: <https://lib.dr.iastate.edu/etd>



Part of the [Computer Sciences Commons](#)

---

## Recommended Citation

Xu, Yijia, "Towards DO-178C compatible tool design" (2015). *Graduate Theses and Dissertations*. 14740.  
<https://lib.dr.iastate.edu/etd/14740>

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact [digirep@iastate.edu](mailto:digirep@iastate.edu).

**Towards DO-178C compatible tool design**

by

**Yijia Xu**

A thesis submitted to the graduate faculty  
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:  
Robyn R Lutz, Major Professor  
Johnny S Wong  
Wensheng Zhang

Iowa State University

Ames, Iowa

2015

Copyright © Yijia Xu, 2015. All rights reserved.

## TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS .....	iv
ABSTRACT .....	v
CHAPTER 1. GENERAL INTRODUCTION .....	1
CHAPTER 2. MODEL-BASED TEST CASE GENERATION .....	5
2.1 Motivation .....	5
2.2 Related works .....	5
2.3 Background .....	7
2.3.1 Basics of testing .....	7
2.3.2 Test types .....	8
2.3.3 Functional and system testing .....	9
2.3.4 Model- Based development .....	10
2.3.5 Mathworks Simulink .....	11
2.4 Generating test sequences for Simulink models .....	13
2.4.1 Framework .....	13
2.4.2 Implementation details .....	17
2.4.3 Generation of a test case from a Simulink graph .....	20
2.4.4 Tools used for the implementation .....	27
2.4.5 Examples .....	31
2.4.6 Evaluation .....	36
2.4.7 Tool qualification .....	39
CHAPTER 3. PYDETECTOR - STATIC CODE ANALYSIS ENGINE .....	41
3.1 Motivation .....	41
3.2 Related works .....	41
3.2.1 PyLint .....	41
3.2.2 PyFlakes .....	44
3.2.3 PyChecker .....	45
3.2.4 PySonar 2 .....	48
3.3 Background .....	48

3.4 Problem definition.....	49
3.5 System architecture .....	50
3.6 Static code analysis .....	51
3.6.1 Literal.....	52
3.6.2 Variables.....	52
3.6.3 Expressions.....	53
3.6.4 Statement .....	53
3.6.5 Imports.....	53
3.6.6 Control flow.....	54
3.7 Evaluation.....	57
3.8 Tool qualification.....	59
CHAPTER 4. CONCLUSIONS AND FUTURE WORK.....	60
BIBLIOGRAPHY.....	62

## ACKNOWLEDGMENTS

I would like to thank my committee chair, Professor Robyn Lutz, and my committee members, Professor Johnny Wong and Professor WenSheng Zhang, for their guidance and support throughout the course of this research.

In addition, I would also like to thank my wife, Yuerong Wang, who always encourages and supports me. I would also like to thank my baby Charlotte Xu, who patiently stays in her mother tummy to support my graduation.

## ABSTRACT

In software development, testing often takes more than half the total development time (Pan 1999). Test case design and execution of test procedures consume most of the testing time. Thus, automatically generating test cases and automatically detecting errors in test procedures prior to execution is highly advantageous. This thesis proposes a new approach to further automate test case design and the test procedure development process.

Several open-source products exist to automate test case design, but they have limitations including test cases that do not trace back to models; test cases that are not reusable for libraries; and limiting test cases to generation on their own test environment. This limits their support for the important, new avionics standard, DO-178C (RTCA 2012).

The first contribution of the thesis is a technique for test code generation that, compared to existing products, is faster, provides improved traceability to models, and supports reusable test procedures that can be generated on any testing environment. To address the current limitations, the new approach utilizes the Simulink Design Verifier and an open-source constraint solver to generate test cases. The technique allows each test case to be traced back to an expression and to the original model.

Detecting errors in manually written test procedures before testing starts is also critical to efficient verification. It can save hours or even days if errors are detected in the early test procedure design stage. However, analysis done here of a set of open source code analysis tools shows that they cannot detect type and attribute errors effectively.

The second contribution of the thesis is to develop a static code analyzer for Python code that detects bugs that could cause automated test procedures to crash. The analyzer converts a Python code to an abstract syntax tree and detects all type and attribute errors by

performing a type-flow analysis. This approach provides improved accuracy over existing products.

Together, these two contributions, a test code generator with improved traceability and reusability, and a static code analyzer capable of handling more error types, can improve test process compatibility with DO-178C.

## CHAPTER 1. GENERAL INTRODUCTION

The motivation for the work described here is to provide better support for the testing process required by a new certification standard for aerospace. In a software development life cycle, software testing takes more than 50% of the time (Pan 1999). Test case design and execution of test procedures consume most of the time in testing. Thus, generating test cases and detecting errors in test procedures automatically are highly required. In our thesis, we propose a new approach to further automate test case design and the test procedure development process in order to improve test process compatibility with the new DO-178C standard.

The document Software Considerations in Airborne Systems and Equipment Certification, also known as DO-178C (RTCA 2012), is a software development and verification standard from RTCA (Radio Technical Commission for Aeronautics) and is a joint work with EUROCAE (European Organisation for Civil Aviation Equipment). This document replaces the previous standard DO-178B and has become the primary document used by certification authorities such as the FAA (Federal Aviation Administration), the EASA (European Aviation Safety Agency), and Transport Canada for approval of all commercial aerospace software systems. DO-178C was completed in November, 2011 and approved by the RTCA in December, 2011. It is free for downloading by all RTCA members and non-members and can be downloaded by anyone for a fee. Certification companies require avionics companies to submit evidence that they comply with DO-178C approach before releasing their products to market.



Software verification is a crucial part of DO-178C. DO-178C requires normal range test cases, robustness test cases, requirement-based test coverage analysis, and structural coverage analysis. Existing coverage-based technology does not satisfy the needs of the DO-178C standard.

On the commercial market, there are two main approaches in products for generating test cases. One is heuristic based products, such as TestWeaver and Reactis system (TestWeaver 2015) (Reactis 2015). They use heuristic-based algorithms to generate test cases in order to cover more program branches. This time-consuming process not only generates many redundant test cases. In addition, the test cases are not traceable to the model in a human-readable format.

The other type of approach in a commercial product is iterative based product such as Simulink Design Verifier (SDV). Simulink Design Verifier iterates every block in a Simulink Model and generates test cases. SDV is faster than heuristic based products. However, as a proprietary product, the test cases can only be simulated in MATLAB and cannot be read by other programs. This is a problem in the avionic industry because the DO-178C standard requires running test cases on target platform.

To better meet the testing criteria in DO-178C, in Chapter 2 we describe software we have developed that utilizes both Simulink Design Verifier and a graph-based algorithm to generate test cases. We convert a Simulink Model to a dependency graph and from such graphs generate an algebraic expression for each output. By utilizing an open source constraint solver, test cases are generated by solving the algebraic expression. Such test cases are then combined with test cases generated by a Simulink Design Verifier and stored as a table (CSV) file format. By reading the CSV file, our

program also generates reusable automatic test procedures (ATP) to be simulated on the target platform.

Compared to existing products, our solution has the following advantages:

- It is faster than existing heuristic based products
- Compared to SDV, an iterative based product, our software can export test cases to CSV file format and simulate them on any test environment.
- Test cases for library models can be reused in multiple ATPs and test cases can be traced to Simulink models.

Another major challenge in verifying airborne software systems is writing an ATP. ATPs are usually written in Python and then executed on specific testing systems. Potential bugs in an ATP could cause a system to crash, and discovering bugs before running the ATP remains a big challenge in avionics.

In Python, a dynamic language with variable types is determined by the program run-time point. Very few rules have been developed for dealing with variable types. The aim of this thesis is to propose a solution for detecting bugs that could lead to ATP crashes. Inferring the types in a static language is relatively easy since languages like C/C++ have strict rules, but nearly all dynamic languages are dynamically-typed. A language like Python is bounded by very few rules, and a test developer, therefore, may have great freedom in writing the test script; using this type of freedom may also be dangerous.

In Chapter 3, we analyze a set of widely used, open source static analysis products and discuss errors they can detect. However, those products cannot catch type errors and attribute errors effectively. These errors are the main reason for ATP to crash. We thus

provide a solution combining the power of an abstract syntax tree and type-flow analysis to detect fatal bugs in an ATP. The result produced by our tool PyDetector is a list of fatal errors including line numbers from the source code. The purpose of this tool is to detect errors that could lead to ATP crashes, thereby reducing the time-consuming load on ATPs.

In Chapter 4, we offer conclusions and describe future work.

## CHAPTER 2. MODEL-BASED TEST CASE GENERATION

### 2.1 Motivation

Model-based software development has become widely used during the past two decades in the avionics and automobile industries. Unlike directly using C / Ada code, models usually specify software designs, and executable code (C/C++) code can be generated from models. Models are written in a specific modeling language featuring very high-level abstraction that is convenient for domain engineers. In this thesis, the modeling language used will be Simulink from MathWorks. In Chapter 2, we will design an automated tool to generate test cases from a Simulink Model.

### 2.2 Related works

Simulink was originally designed for simulation purposes. DO-178C requires automated test generation to identify errors (crashing, software defects) within model implementation and model design issues (requirement / design mismatch, conflict requirements). Many publications have discussed how to generate test data to test a Simulink model. However, not many have focused on a Simulink model for verification and validation.

Zhan (Zhan 2005) proposed a mutation-based approach to cover program branches in Simulink Model. He only considered three types of mutation operators as inputs, but did not cover all operators in the model. Our approach overcomes this limitation by traversing the graph generated from the Simulink Model.

Reactive Tester (R. Systems 2012) is a guided simulation and heuristics test-case generator. It first allows a user to specify some coverage criteria and testing properties, following which it will use heuristic-based test case generation to cover as much test

criteria as possible. However, a heuristic-based algorithm may take a very long time to cover all test criteria. This approach is limited with respect to model complexity and size, and may require excessive time to generate input signals and achieve adequate model accuracy with respect to the actual software. Our approach overcomes these limitations and will also work for complex models.

Gadkari, et al., (Gadkari, et al. 2007) translate a Simulink model into Lustre, a formal language, then use a model-checking tool Lesar to generate a test case. Meng Li and Ratnesh Kumar converts Simulink model into an Input/Output Extended Finite Automata (Li and Kumar 2012), a formal model of a reactive untimed infinite-state system. Our approach will also translate the model into an intermediate representation but, rather than using a formal language, we will use a graph to represent the model.

Whalen, et al, (Whalen, et al. 2013) reveal that OMC/DC can discover more bugs than MC/DC (Hayhurst, et al. 2001), but OMC/DC is more difficult to use due to the following issues:

- OMC/DC has no definitive pattern for test suite size.
- OMC/DC obligation is more likely to be uncoverable.
- OMC/DC is sensitive to program path

In DO-178C, test cases must be written purely based on requirements. It is thus impossible to write test cases after only looking through program paths. DO-178C also requires 100% coverage unless reasonable justification for not doing so is provided. Using an uncoverable coverage-testing method is obviously not wise, so choosing MC/DC instead of OMC/DC is more suitable in safety-critical systems to be certified with DO-178C.

Panesar-Walawege, Kaur, Sabetzardeh and Briand (Panesar-Walawege, et al. 2013) proposed a safety-standards-compliant approach for verifying model-based systems. They created a conceptual model from a safety standard followed by a UML profile relating all safety standards to system development artifacts. Their example was applicable only during the development stage where no test case is needed, and they also did not specify whether their approach was compliant with DO-178C (they mentioned only IEC61508 with respect to compliance). Our work focuses on the verification stage that requires test cases and uses either automatic or non-automatic test procedures; our work will be compliant with the DO-178C standard.

## 2.3 Background

### 2.3.1 *Basics of testing*

Software testing aims to find defects in software products and such testing is considered to be the most effective and important means for reducing defects and increasing robustness in software systems. The testing process involves writing a set of test cases and observing whether the software product behaves as expected. It can be subdivided into software verification and validation processes.

Software Verification answers the question “Are we building the project right?” (Leveson 1995) By performing software verification, we attempt to make sure that the software behaves in the manner we want it to.

Software Validation answers the question “Are we building the right product?” (Leveson 1995) The software validation process will check whether there have been any mistakes leading the software developer to develop a product that does not do what the

customer asked for. Software validation usually compares either the model or the actual code against requirements.

### *2.3.2 Test types*

In this section, we will discuss several types of testing using the descriptions in Chauhan's book (Chauhan 2010).

**Black-box testing:** We test the software without knowing how its implementation details. We will write test cases based on requirements to test each functional software component. The disadvantage of black-box testing is we might be unable to effectively test all pieces of code without looking at the code.

**White-Box Testing:** White-box testing deals with the code's internal structure; test cases are developed by looking at the actual code. Using this method, the tester needs to understand the logic of the code. It also requires the tester to look deeply into the code and determine exactly which code component may be malfunctioning.

The advantages of white-box testing are that it helps detect unused code and helps optimize the software. Also, by understanding just how the software behaves, test cases can be written more thoroughly and accurately.

Testing can also be subdivided into static and dynamic testing (Chauhan 2010) based on whether a code is executed. Dynamic testing tests the code by executing it, while static testing is performed without executing the code.

**Static Testing:** Static Testing is basically testing the code without running it. This could include review, inspection, or walkthrough to discover issues. An inspection report is required under the DO-178C standard.

**Dynamic Testing:** Using dynamic testing, the code under review must be actually compiled and run. Dynamic testing is essentially the validation part of Software Verification & Validation (V&V). DO-178C requires running dynamic tests on airborne software through 5 software levels. Each level has a different dynamic testing requirement. Coverage-based testing is required for Level A airborne software.

**Unit Testing:** Unit testing is a testing method under which subsystems, related units, or software modules are “tested to determine whether they are fit for use” (Huizinga and Kolawa 2007).

**Integration Testing:** In contrast to Unit testing, Integration testing combines all software and hardware components to evaluate interactions between software units and hardware units. Black-box testing and white-box testing are both usually used to confirm that all units are working together correctly and safely. Integration testing is very important in verifying airborne systems because such systems are usually very large and a verification team must break them down and do unit testing from the very beginning. However, even if each component individually works correctly it does not necessarily mean that all components will perform correctly after everything is assembled together. To write integration test cases, test engineers must refer to both high-level and low-level design documents.

### *2.3.3 Functional and system testing*

**Functional testing:** Functional testing involves the following tests:

1. All functionalities specified in the requirement or specification work correctly.
2. The software has implemented all functionalities listed in the requirement or specification.



3. The software has no extra functionality not covered by the requirement or specification.

**System testing:** System testing is usually performed as part of integration testing while all systems are running. The following functionalities should be included in the system testing (Indian Student Association 2012):

- Graphical user interface
- Usability
- Software performance
- Exception handling
- Compatibility
- Installation
- Recovery or failover

#### *2.3.4 Model- Based development*

Model-Based Design or Model-Based Development (MBD) (Schätz, et al. 2002) is a modern way to develop complex systems. Instead of directly writing executable code, MBD focuses on high-level models of the system to be placed in the field.

MDB is extensively used in the aerospace and automotive industries. Model-based design can begin with a well-designed common framework, and throughout the design process customized features can easily be added. AUTOSAR (Autosar 2014), for example, is available to automobile manufacturers as an open and standardized software framework. Model-based design also supports most development cycles (“V” model, agile model, spiral models, etc.). Figure 2.1 is an example for the V model.

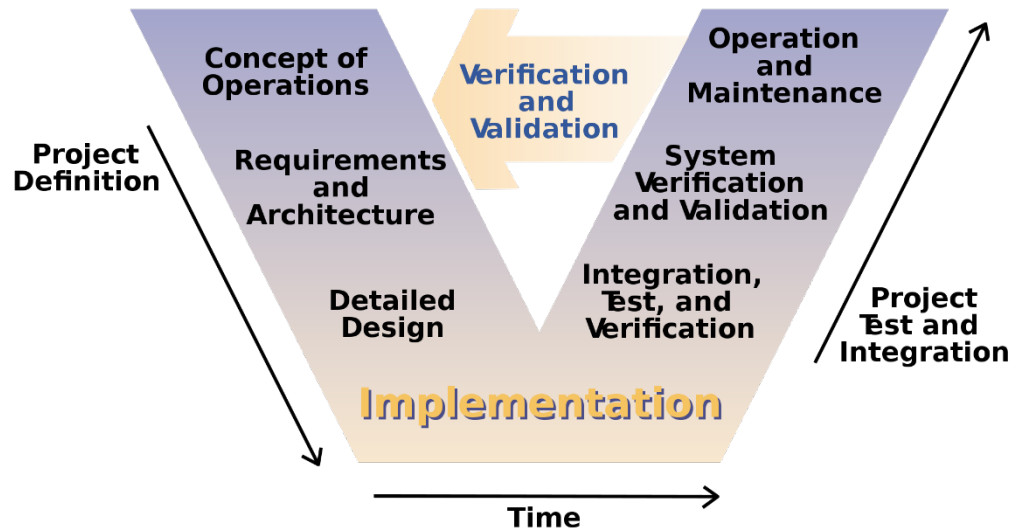


Figure 2.1: V-Model (V model 2015)

MBD usually consists of the following development steps:

- Modeling the target system
- Modeling the target system environment.
- High-fidelity simulation
- Software V & V Process
- Deployment and maintenance

### 2.3.5 Mathworks Simulink

Simulink (MathWorks 2015), developed by MathWorks, is a graphical programming language tool for developing embedded systems, now widely used in the areas of aerospace, automobile, cell phones, and even TVs. As a result, many standards have been developed to guide engineers in design, development, testing, and verification of software systems in such applications. Using MBD to design software is an important industrial trend.

Simulink is used as a standard development tool in avionics and auto design. It integrates modeling and simulating dynamic systems. A wide range of library blocks are provided in Mathworks, including an aerospace blockset, a Computer Vision system toolbox, and a DSP system toolbox, Different industries can purchase different toolboxes based on their particular needs. Simulink supports both linear and non-linear systems and can simulate software using both fixed-step and variable-step operations in its continuous-time or sample-time modeling.

The Simulink environment supports distributed development of large and complex systems and provides a variety of customizable simulation and computational blocks suitable for creating embedded systems. Executables can be automatically created from these blocks and compilable source codes can also be generated before building executables. The Simulink Code Generator supports all kinds of embedded platforms and source code generated is Misra-C (MISRA Consortium 2012)-compatible to eliminate compatibility problems when migrating to different platforms. The following are some of the block libraries:

- Sources Library: This library provides blocks that generate inputs for input signals.
- Sinks Library: This library provides blocks for displaying values during simulation.
- Aerospace Library: This library provides libraries for simulation of aerodynamics environments and other functionalities.
- Vehicle Network Library: This library provides blocks to encode/decode Control Area Network (CAN) signals and XCP signals.

## 2.4 Generating test sequences for Simulink models

### 2.4.1 Framework

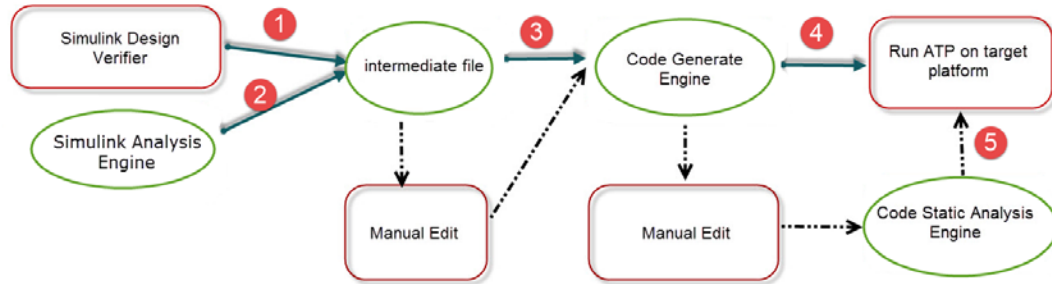


Figure 2.2 Tool Framework

Figure 2.2 shows the tool framework. The broken line in the picture indicates an alternative, non-required route. The red rectangle components are the already-existing tools. In this thesis, while I am using these tools to do a portion of the work, the oval components are the tool we designed. We will introduce all the components in the following paragraphs:

#### *Simulink Design Verifier*

The Simulink Design Verifier (MathWorks 2015) is a Mathworks tool that can generate test cases as a MATLAB mat file. Post-processing the mat file will generate an intermediate file for further processing. Details of how to use the Simulink Design Verifier to generate test cases will be discussed below.

#### *Simulink Analysis Engine*

The Simulink Analysis Engine is an engine I designed to take a Simulink Model as input to generate an intermediate file to feed into the Code Generate Engine.

The Simulink analysis Engine reads a Simulink Model and converts it to a graph structure.

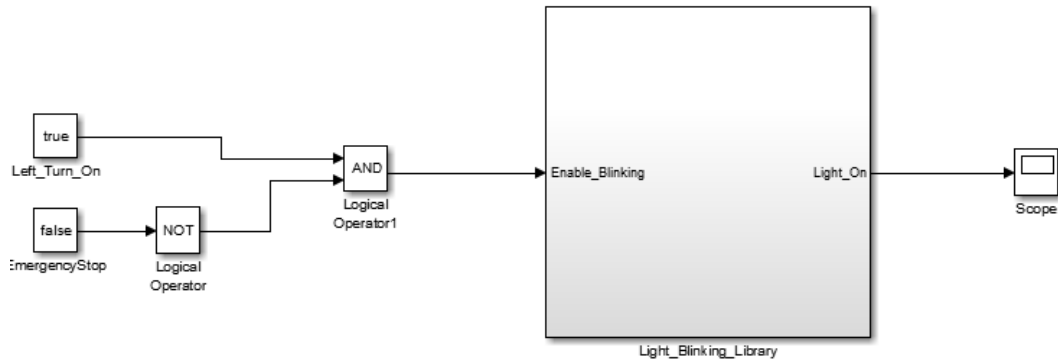


Figure 2.3 “Left Turn” Simulink Model

Figure 2.3 is a Simulink model. As can be seen in the following figure, this model can be converted to a directed graph structure in which each node indicates a Simulink block. A graph representation is equivalent to a Simulink model.

The following image is a graph representation of this Simulink model. Details of conversion of a Simulink Model to a graph representation will be discussed in Section 2.4.3.

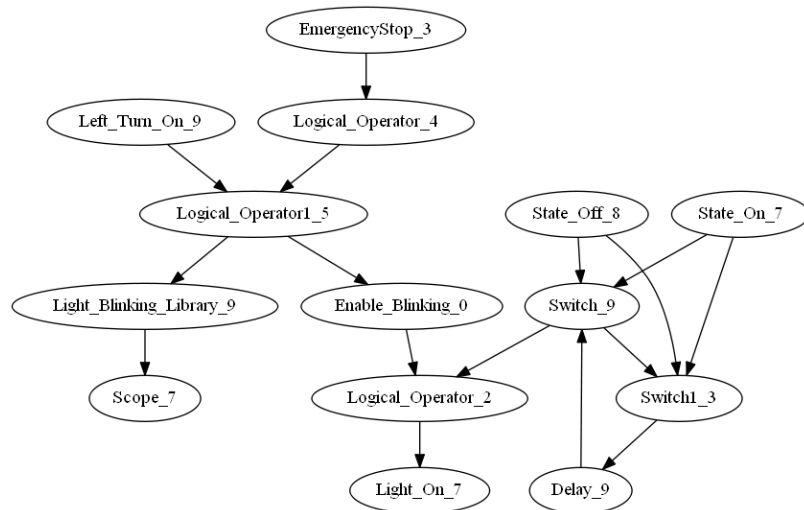


Figure 2.4 Graph representation of “Left Turn“ Model

After a graph is constructed, the following algorithm can be used to generate test cases:

1. Starting from one output signal, use Depth First Search (DFS) to begin traverse graph.
2. Once a node is reached, we perform the following process
3. Read the type of the node and see if it is supported
  - a. If the node is supported, generate a test case for the node.
  - b. Otherwise, report an exception and terminate the algorithm.
4. Terminate the process once it reaches Inport (a Simulink block type).

Table 2.1 Java class to Simulink block mapping

Java class	Simulink Model blocks
ATPCodeBlock	The base type for all Simulink Blocks
ATPInPortCodeBlock	InPort and OutPort
ATPLogicalOperatorBlock	Logical Operators (AND, XOR, OR)
ATPReferenceCodeBlock	All user created reusable libraries
ATPRelationalCodeBlock	Relational Operators ( $>$ , $\geq$ , $<$ , $\leq$ , $=$ , $\neq$ )
ATPSwitchCodeBlock	Switch Case
ATPConstant Block	Constant
ATPSubSystemCodeBlock	Subsystem
ATPValueCodeBlock	Constants, FromWorkSpace

Theoretically, all nodes should be reached if we have traversed the graph from all output nodes; otherwise, the unreachable nodes represent dead code. Because each node has a different type with different functionality in Simulink, we will create a java class

for each type of Simulink block to customize the test-case generation process. All supported Simulink blocks are listed in Table 2.1.

#### *Test case representation*

All test cases are represented in a CSV file format. The first row states what is on the second row. If it is a “Stimulus”, then it is an input signal we are trying to set; if it is “Expected”, then it is an output signal we are trying to verify; “Condition” means a certain condition must be satisfied before verifying the output signal.

The following example shows a set of test cases for Turning\_Signal\_Left model.

Table 2.2 Test case for Left\_Turn Model

Stimulus	Stimulus	Condition	Expected
Left_Turn_On	Emergency_Stop		Light_On
TRUE	FALSE		TRUE
TRUE	TRUE	DELAY:1000	FALSE
FALSE	TRUE	DELAY:1000	TRUE

#### *Code Generation Engine*

The code generation engine generates Python ATP from the CSV file discussed in the previous section. This process maps to arrow 4 in Figure 2.2. The Code Generation Engine will read the first two rows to obtain the information for each test case. Starting from the third row, it will use the following algorithm to generate test cases:

1. Read the first row to get the stimulus
2. For each stimulus, perform the following steps:
  - a. Read the CSV file. Check whether the signal is a system input or an output signal arriving from another model

- b. If it is a system input, use the `SystemInterface.SetSignal` function to set a stimulus value.
  - c. If it is a signal arriving from another model, import the utility file from the other model and use the `set` function in the utility file to set the stimulus value.
3. Achieve all the conditions (e.g. if the condition is `DELAY: 1000`, then we will wait 1000 milliseconds before verifying the output signal)

Verify whether the output signal is the same as the expected value.

#### *Static Code analysis Engine*

The Code Static Analysis Engine is a tool for detecting issues inside a Python ATP file. Intermediate files and ATP files can be manually edited, so human errors might exist in them. Typical ATP errors are type errors and attribute errors, e.g., assigning a value to an undefined variable or providing more parameters than needed by a function. To ensure that the ATP is free of such errors, we can use this static analysis engine to scan it. Details of the Static Code Analysis Engine will be discussed in Chapter 3. This process maps to arrow 5 in Figure 2.2.

#### *2.4.2 Implementation details*

##### Generating a test case from the Simulink Design Verifier

We have two ways to generate test cases. Arrow 1 in Figure 2.2 uses Simulink Design Verifier. It generates its test cases into a `sldvData` structure containing the following information:

Table 2.3 `sldvData` structure

Entry name	Information contained
------------	-----------------------



Table 2.3 continued

ModelInformation	It is a structure containing Model Name, version, Author and time stamp for the model
AnalysisInformation	It is a structure containing all important information during the analysis. The most useful information is the InputPortInfo and OutputPortInfo. These two structures contain properties (type, default value, etc.) regarding the input signals and output signals
Model Objects	Contains all blocks that will affect out coverage, including logical blocks, Relational blocks
Constraints	A set of constraints such as the value range for an input signal
Objectives	Objectives to be met to achieve 100% coverage
TestCases	A structure containing all test cases needed to satisfy all objectives
Version	The version of the sldvData structure

The sldvData structure contains multiple test cases and is not in human-readable form, so we must write an m-script to extract all test case information from the structure and convert it into a human-readable CSV file:

We use the following m-script to decode the structure and generate a CSV file containing all test cases:

```
function sldv2tpt(sldvfile)
```

```
% sldv2tpt(sldvfile) converts Simulink Design Verifier test data into a CSV format
```

```
% Input: sldv-file Output data file generated by Simulink Design Verifier
```

**% Output: Testcase<n>.mat Files n=1..number of generated test cases. One file for each test case**

**%**

**load(sldvfile);**

**fid = fopen('te.csv','w');**

**for j = 1: length(sldvData.AnalysisInformation.InputPortInfo)**

**uname = sldvData.AnalysisInformation.InputPortInfo{j}.SignalLabels;**

**fprintf(fid, '%s,', uname);**

**end**

**for j = 1: length(sldvData.AnalysisInformation.OutputPortInfo)**

**uname = sldvData.AnalysisInformation.OutputPortInfo{j}.SignalLabels;**

**fprintf(fid, '%s,', uname);**

**end**

**fprintf(fid, '\r\n');**

**csv =**

**cell(length(sldvData.TestCases.dataValues)+length(sldvData.TestCases.expectedOutput),**

**length(sldvData.TestCases.dataValues{1}));**

**for i= 1: length(sldvData.TestCases)**

**for j = 1: length(sldvData.TestCases.dataValues)**

**val = strsplit(num2str(sldvData.TestCases.dataValues{j}), ',');**

**for k = 1 : length(sldvData.TestCases.dataValues{j})**

**csv{k,j} = val{k};**

**end**

```

    end;

end

for j = 1 : length(sldvData.TestCases.expectedOutput)

    val = strsplit(num2str(sldvData.TestCases.expectedOutput{j}), ' ');

    for k = 1 : length(sldvData.TestCases.dataValues{j})

        csv{k,j + length(sldvData.TestCases.dataValues)} = val{k};

    end

end;

for i = 1 : length(csv)[1]

    for j = 1 : length(csv)[2]

        fprintf(fid, '%s, ',csv{i,j});

    end

    fprintf(fid, '\r\n');

end

fclose(fid);

```

The CSV file will contain a set of stimuli and a set of expected outputs. The first row specifies the number of stimuli, which of the stimuli should be set, and the number and identification of signals we must test.

### *2.4.3 Generation of a test case from a Simulink graph*

Our method of generating test sequences (arrow 2 in Figure 2.2) converts a Simulink Model into an output dependency graph. Output dependency graph is a directed graph containing nodes and edges as shown in Figure 2.4. One node in the graph maps to one Simulink block in the Simulink model; one edge in the graph maps to one edge in the

Simulink Model. An edge in the graph represents transferring a signal from the source node to the destination node after performing a math operation between the two connected nodes based on their properties. An edge from node A to B signifying that the output of node B depends on node A.

The output dependency graph generation algorithm takes a Simulink model as input and generates a graph. It first generates the top level; if any subsystem is found in the top-level model, it will be pushed into a queue. Once the top level finishes, it pops the next subsystem from the queue and continues construction.

The following code describes how a Dependency graph is generated:

**Input:** *a Simulink Model file*

**Output:** *Dependency graph of the Simulink model*

BEGIN:

Initialize two empty queues, Queue\_Model and Queue\_Blocks

Initialize an empty graph

Push the Simulink model into Queue\_Model

*//First, we create an ATPCodeBlock instance for each block*

While Queue\_Model is not empty:

    Pop the model

    Extract all blocks from the model

    For each block in the model:

        If a block is a “subsystem”

            Push the block into a queue

        else

```
        Create an ATPCodeBlock object and add this node to graph
        Push the Block object into Queue_Blocks

    End if

End for

End while

//Create the graph by adding links from current block to its adjacent blocks

While Queue_Blocks is not empty:

    Pop a block

    Read all its adjacent blocks

    Create a link from this block to all its adjacent blocks

End while

Export the graph to a dotty file

END
```

Automated testing is a technique for reducing both the time and effort expended in the testing process. For constructing an automatic testing tool, coverage criteria is required; this can be path coverage, branch coverage, etc. In accordance with such criteria, test suites are automatically generated by the tool as input for given software. As discussed in section 2.2, we will use MC/DC as the criteria.

In order to generate MC/DC-compatible test cases from Simulink models covering as many paths as possible, we use a concept of hybrid testing combining static analysis (to generate test cases) with dynamic analysis (to run tests and generate coverage report).

Given a Simulink model P as input, the goal is to output a set of test cases achieving high MC/DC coverage. To do so, the model P will be converted into an output-dependency graph and paths in the graph will be iterated to generate constraints.

The Major Components for the systems are:

- Static Analyzer
- Test Case Generator
- Output Dependence Graph (ODG Module)
- Coverage Module

Figure 2.5 describes the architecture of the system:

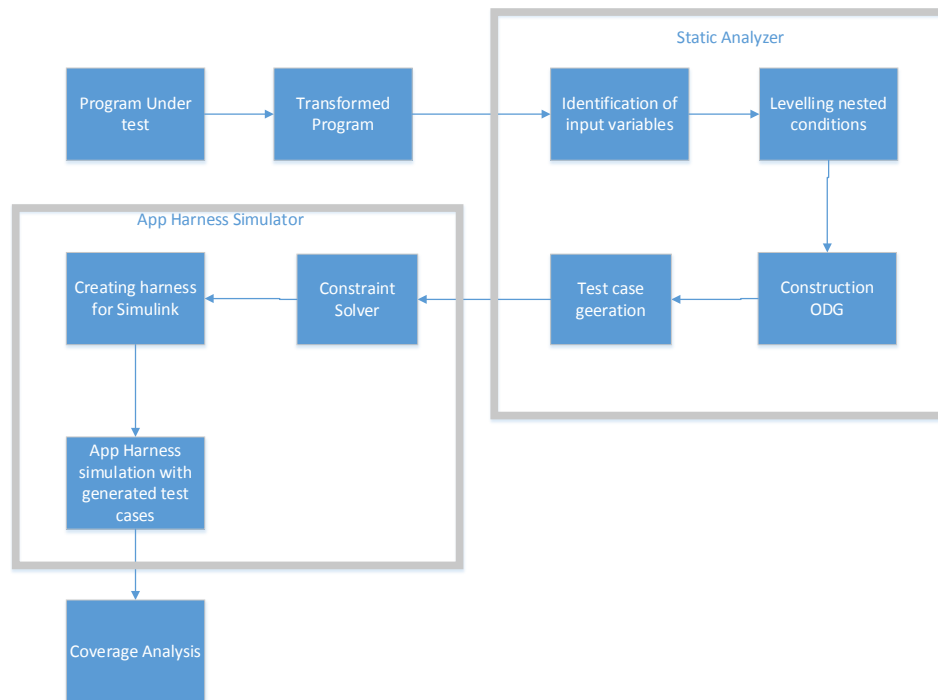


Figure 2.5 Static Analyzer & Test Case generator

The static analyzer and test case generator will work together to create the test cases. The structure of the system can be expressed as shown in Figure 2.6:

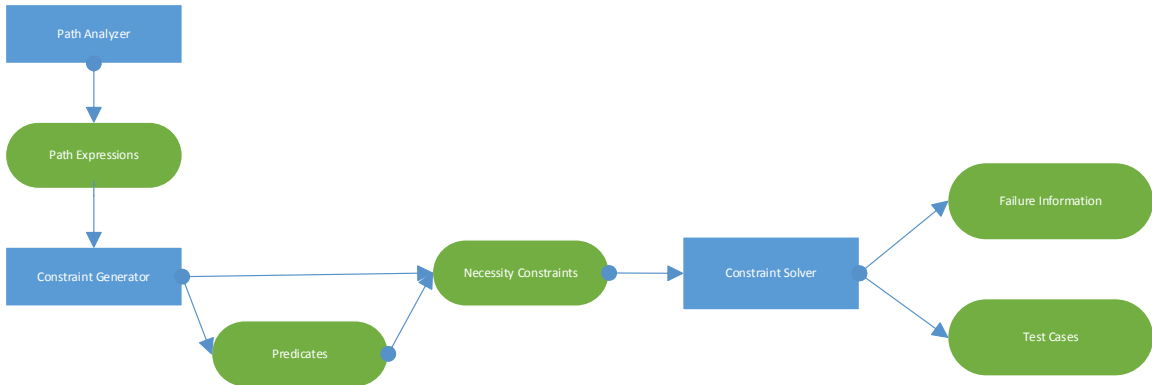


Figure 2.6 Structure of the system

The path analyzer will iterate through all the paths, and pass results to the path expressions. Then a constraint generator will subject each Simulink block to a constraint and append it as a constraint equation. If an input has constraints from another block, the predicates will know this and add them into the current equations. Then of necessity the constraints checker will check to see if there is a duplicate constraint or a useless constraint; for example, if in the previous calculation, we concluded that signal  $A > 10$ , and then in the next iteration, we encounter another constraints that say  $A > 25$ , then this constraint is a useless constraint. After this analysis, the equations will be sent to the constraint solver. Constraint solver will calculate a possible solution based on the set of equations. Once this is done, the result will be used to generate test cases.

The basic element in a constraint is an algebraic expression consisting of a set of variables, logical/relational operators. Expressions are calculated directly from the output dependency graph. A constraint is expressed using variables and one or more of the following operator{ $<$ ,  $>$ ,  $=$ ,  $\geq$ ,  $\leq$ ,  $\neq$ }; in MATLAB, it maps to a relational library block.

When iterating the graph, each Simulink library block will be mapped to a constraint using the following table:

Table 2.4 Simulink block to constraint mapping

Block Type	Description	Constraint
OR	Logical OR	$input1 \vee input2$
AND	Logical AND	$input1 \wedge input2$
NOT	Logical operator	$input1 \neq input2$
Compare	Relational compare	$input1 < input2$ ; OR $input1 > input2$ ; OR $input1 = input2$
Math	Math operators ( -, +, *, / )	$input1 - input2$ ; $input1 + input2$ ; $input1 * input2$ ; $input1 / input2$ ;

Once a constraint is generated, the expression will be sent to JaCoP, an open-source constraint solver that will analyze the expression and output a feasible result. Once we have this answer, we will be able to use it to generate test cases.

During the iteration, when a node with a Simulink Output Block type is reached, the constraint will be stored on that node because the output signal will serve as an input to another model. Just as we may call a function A and, inside function A, we may call function B to obtain a returned value, this returned value will be used to perform some further calculation in function A. In this example, the returned value is the output signal. Because function A is dependent on function B. the constraint we set for the output node in function B will be set as the initial condition for function A. In this manner we can



ensure that, when generating a test case for a subsystem, the combination of all the test cases will still work on the whole system.

### **Reusable test procedure generation**

In the Simulink model, we can also have reusable subsystems. For example, if we are developing software for a car, the front light left-turning signal and right-turning signal, the rear light left-turning signal the right-turning signal, and the emergency-stop signal will all use the same logic. To test these models, we would like to use a single standard test procedure so that the tests would be consistent throughout all the different models.

We can use the following algorithm to generate test case:

1. **Function** Generate\_TestCase
2. **Input:** *a Simulink Model file*
3. **Output:** *Test Case*
4. BEGIN:
5. CALL function in Section 2.4.3 to convert Simulink Model file to Simulink graph
6. Initialize two empty list, list\_constraint and list\_testcase
7. For each OutPort in the graph
8.     Clear list\_constraint
9.     CALL Function Generate\_Logical\_Expression(OutPort, list\_constraints)
10.    If OutPort is boolean:
11.       Solve list\_constraints using constraint solver with OutPort = true and false
12.    Elseif OutPort is numeric:

13. Solve list\_constraints by giving OutPort = Minimum, Middle, Maximum of its valid range
14. Else:
15. Throw unsupported exception
16. END IF
17. Push the result to list\_testcases
18. END
19. **Function** Generate\_Logical\_Expression
20. **Input:** A Simulink block and a list
21. **OutPut:** A list of constraints for this block
22. BEGIN:
23. If input block is a InPort:
24. END
25. END IF
26. Convert all connected blocks into a logical expression for the input block
27. Push the logical express (which is the constraint for the block) to the input list
28. For each connected block:
29. CALL Generate\_Logical\_Express(connected\_block, list)
30. END

#### *2.4.4 Tools used for the implementation*

In this section, we will introduce the tools we used to implement all functions.

The following tools have been used to implement the software:

- Eclipse

- Window Builder Pro
- Graphviz

### *Eclipse*

Eclipse (Eclipse 2015) is an open-source integrated development environment. By installing different plug-ins, Eclipse can be adapted to develop applications for various programming languages. The latest official release version, Eclipse Luna, is the first official support version for Java 8.

With respect to the Eclipse version, there are still different IDEs corresponding to different Java Developers. The following IDEs are currently supported by Eclipse Luna:

- Eclipse IDE for Java Developer
- Eclipse IDE for Java EE developer
- Eclipse IDE for C/C+ Developer
- Eclipse for Tester
- Eclipse for Java and Report Developers
- Eclipse IDE for Automotive Software Developer

The difference between Java EE IDE and Java IDE is that Java EE IDE provides all functionalities in Java IDE as well as support for Java EE, JPA, JSF, etc. In this project, the Java EE additional functionalities are unnecessary, so we will use the standard java IDE.

Eclipse also provides us a convenient way in which to write Java Code. Designing a GUI, for example, can be a very painful process. To hand-code a beautiful GUI, a developer must declare all the controls and add them for accurately positioning to a panel. Arranging these positions for all controls by hand coding would take hours and it

would be hard to trace between the code and the control. We can use the following plugin to improve the GUI design methodology.

### *Window Builder Pro*

Window Builder Pro (WindowBuilder 2015) is a powerful GUI design plugin for Eclipse IDE. It is a bi-directional GUI designer that allows one to trace an event handler to a button and vice versa. With Window Builder, a developer can drag and drop controls, design completed windows and add event handlers in minutes; the same process could take hours if programming it by hand.

The following user interface components are supported in Window Builder:

- Design View
- Source View
- Structure View
- Palette
- ToolBar
- Context Menu

The above user interface components include all major user interfaces. More importantly, the plugin can create custom, reusable components for a developer to use in multiple projects. It also has a Property Pane that helps a developer set and update properties.

### *Graphviz*

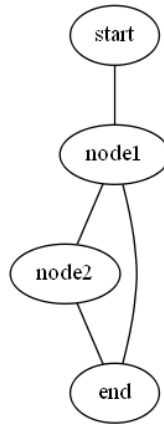
Graphviz (Graphviz 2015) is an open-graph visualization tool. In our thesis, we convert a Simulink Model to a graph. To visualize the graph, we convert it to a dot file format. Graphviz can visualize a graph from the dot file representation.

The dot graph description language can be used to describe a directed or undirected graph.

The following example describes an undirected graph:

```
graph graphname {  
    start -- node1 -- node2;  
  
    node1 -- end;  
  
    node2 -- end;  
  
}
```

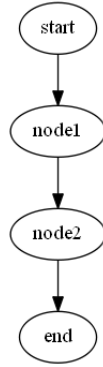
The bold format graph statement states that the graph is an undirected graph. In such a graph, only “--” is syntactically allowed; “--” connects two nodes with a undirected edge. The following figure shows the graph after rendering by Graphviz:



The directed graph is also easy to describe:

```
digraph graphname {  
    start -> node1 -> node2;  
  
    node2 -> end;  
  
}
```

The designation Digraph at the beginning states that the dot file is a directed graph. Unlike an undirected graph, digraph uses “->” to connect two nodes. The following figure shows the directed graph after rendering by Graphviz:



### 2.4.5 Examples

To evaluate our theory, we will use two Simulink Models:

Model AirbrakeControl includes all common Simulink Blocks. Generate test cases for this model shows our tool can support most engineering needs.

Model “Left Turn” includes reference and library blocks. Generating test cases for this model shows our tool can support reusable models.

#### Model AirbrakeControl

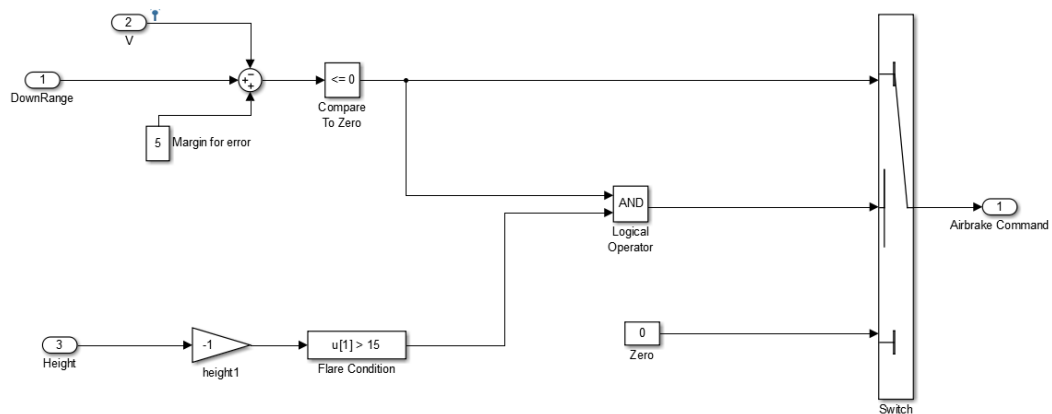
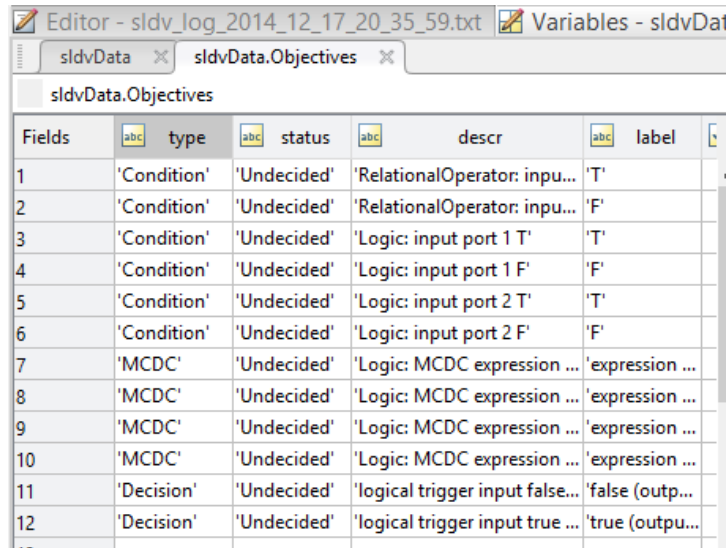


Figure 2.7 AirbrakeControl Model

Step 1, we use Simulink Design Verifier (arrow 1) in Figure 2.2 to generate temp file, the temporary file

The temporary file generated from Simulink Design Verifier is shown below:

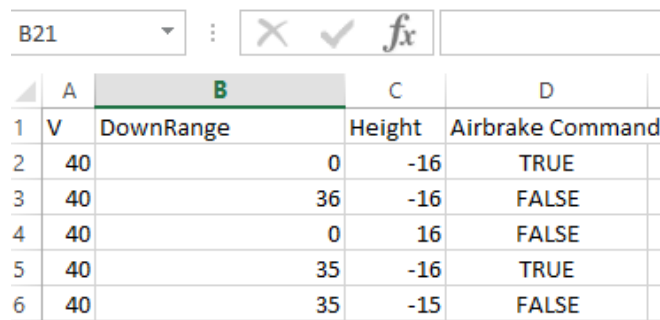


The screenshot shows a window titled 'Editor - sldv\_log\_2014\_12\_17\_20\_35\_59.txt' with a tab for 'sldvData.Objectives'. The table below is a representation of the data shown in the screenshot.

Fields	type	status	descr	label
1	'Condition'	'Undecided'	'RelationalOperator: input...	'T'
2	'Condition'	'Undecided'	'RelationalOperator: input...	'F'
3	'Condition'	'Undecided'	'Logic: input port 1 T'	'T'
4	'Condition'	'Undecided'	'Logic: input port 1 F'	'F'
5	'Condition'	'Undecided'	'Logic: input port 2 T'	'T'
6	'Condition'	'Undecided'	'Logic: input port 2 F'	'F'
7	'MCDC'	'Undecided'	'Logic: MCDC expression ...'	'expression ...'
8	'MCDC'	'Undecided'	'Logic: MCDC expression ...'	'expression ...'
9	'MCDC'	'Undecided'	'Logic: MCDC expression ...'	'expression ...'
10	'MCDC'	'Undecided'	'Logic: MCDC expression ...'	'expression ...'
11	'Decision'	'Undecided'	'logical trigger input false...	'false (outp...
12	'Decision'	'Undecided'	'logical trigger input true ...'	'true (outpu...

Figure 2.8 Simulink Design Verifier temporary file

Step 2, we run algorithm sldv2tpt in section 2.4.2 to convert a temporary file to a CSV file format:



The screenshot shows a table with columns A, B, C, and D. The data is as follows:

	A	B	C	D
1	V	DownRange	Height	Airbrake Command
2	40	0	-16	TRUE
3	40	36	-16	FALSE
4	40	0	16	FALSE
5	40	35	-16	TRUE
6	40	35	-15	FALSE

Figure 2.9 CSV test case

Step 3, at last we use algorithm 2.4.1 (arrow 4 of Figure 2.2) to generate the ATP from CSV file. The following code is ATP generated from Figure 2.9:

**import** system

**#Test step 1**

```
system.setValue('V', 40);  
system.setValue('DownRange',0)  
system.setValue('Height', -16)  
system.verifyValue('Airbrake_Command', True)
```

**#Test step 2**

```
system.setValue('DownRange',36)  
system.verifyValue('Airbrake_Command', False)
```

**#Test step 3**

```
system.setValue('DownRange',0)  
system.setValue('Height', 16)  
system.verifyValue('Airbrake_Command', False)
```

**#Test step 4**

```
system.setValue('DownRange',35)  
system.setValue('Height', -16)  
system.verifyValue('Airbrake_Command', True)
```

**#Test step 5**

```
system.setValue('DownRange',35)  
system.setValue('Height', -15)
```



```
system.verifyValue('Airbrake_Command', False)
```

Model Left Turn

Step1 (Convert Simulink to a graph) and Step 2 (algorithm 2.2) have already been discussed in Section 2.4.1 (Figure 2.3, Figure 2.4 and Table 2.2). In this section, we will show the ATP generated from the test cases.

```
SimulinkLibrary.py
```

```
import system
```

```
import time
```

```
def Test_Light_Blinking_Library(TurningLight, EmergencyLight):
```

```
    #Test step 1
```

```
    system.setValue('TurningLight', True)
```

```
    system.setValue('Emergency_Stop', False)
```

```
    system.verifyValue('Light_On', True)
```

```
    #Test step 2
```

```
    system.setValue('Emergency_Stop', True)
```

```
    time.delay(1000)
```

```
    system.verifyValue('Light_On', True)
```

```
    #Test step 3
```

```
    system.setValue('TurningLight', False)
```

```
    time.delay(1000)
```

```
system.verifyValue('Light_On', True)
```

### **LeftTurnModel.py**

```
import system
```

```
import SimulinkLibrary
```

```
'''
```

```
Test Light Blocking block is a Simulink Library block, thus the test script is generated in  
Simulink Library for reusable purpose
```

```
'''
```

```
SimulinkLibrary.Test_Light_Blinking_Library(Left_Turn_On, EmergencyStop)
```

For every reusable block in Simulink Model, our tool will generate a function in SimulinkLibrary.py. The ATP will call this function to test the block. The advantage of this approach is that if we have another model calling this library block, then instead of writing duplicate code to test this library block, we will call the test function in SimulinkLibrary.py to cover the library block.

To illustrate the reusability, the example below is an ATP generated for “Right Turn Model”, which uses the same library as the “Left Turn Model” shown above:

### **RightTurnModel.py**

```
import system
```

```
import SimulinkLibrary
```

```
'''Test Light Blocking block is a Simulink Library block, thus the test script is generated  
in Simulink Library for reusable purpose'''
```

SimulinkLibrary.Test\_Light\_Blinking\_Library(Right\_Turn\_On, EmergencyStop)

#### 2.4.6 Evaluation

To evaluate the correctness of the software, we consider the following criteria from DO-178C section 6.4.2:

1. Does the software generate normal range test cases and robustness test cases?
2. Does the test cases provide structural coverage (MCDC in our case)?

We export all test cases as CSV format. We can view the test cases in Notepad++ or Microsoft Excel to check if normal range test case and robustness test case are generated.

For the structural coverage evaluation, we use MATLAB to generate a harness model and import the CSV file to the harness model. We select an open source project from MATLAB file exchange (Jeppu 2015). The open source project has blocks commonly used in flight control laws (Jeppu 2015).

The project has 7 models. For each model, we first create a harness model. A harness model is a model with all input connecting to a signal builder. We can feed all test cases to the signal builder, so that MATLAB can calculate the model coverage from the test cases.

A harness model can be easily created by the following command:

```
open_system('Simulink model name');  
load_system('Simulink model name');  
harnessModelFilePath = sldvmakeharness('Simulink Model name',loggedSignalsPlant);  
[~,harnessModel] = fileparts(harnessModelFilePath);
```

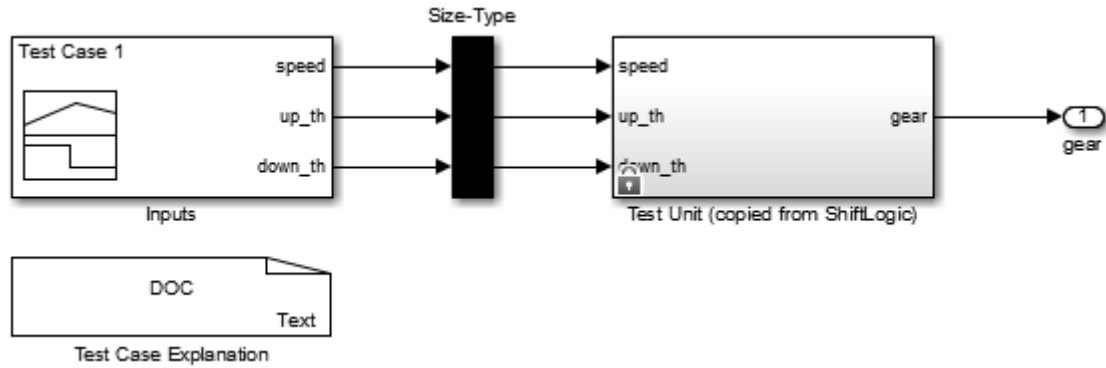


Figure 2.10 Simulink harness model (Mathworks, Simulink Design Verifier 2015)

A harness model looks like Figure 2.10. The left side is a signal generator and it connects to all the input signals in the test unit. The test unit is the model we try to verify.

If we click the “Inputs” on the left side, we can open the signal builder and import our test cases into it. The signal builder looks like the following image, each test case takes one tab in the window:

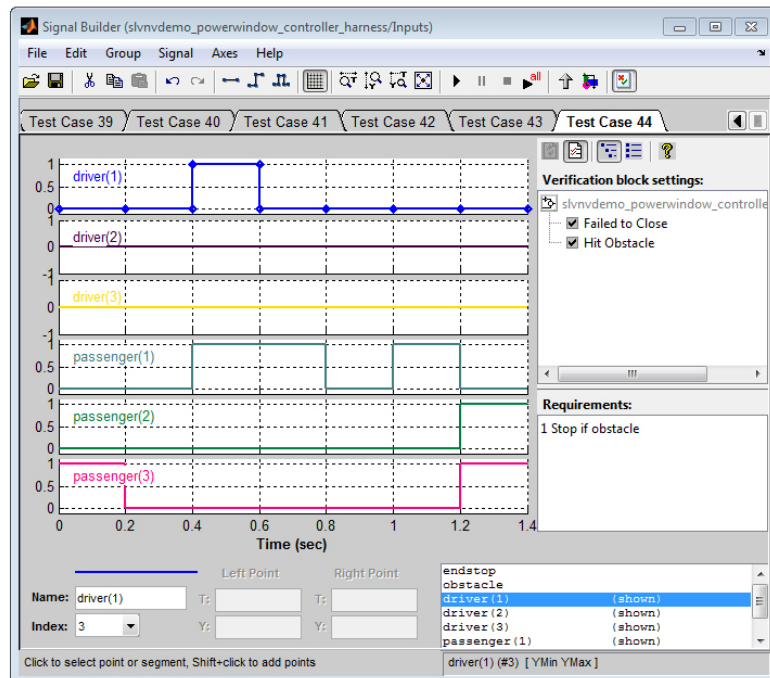


Figure 2.11 Signal builder (Mathworks, Mathworks 2015)

The following table shows the number of test cases generated from each model using our technique and the coverage obtained through these models.

Table 2.5 Simulink Model test case and coverage result

Model ID	Model name	Normal test cases	Robustness test cases	Coverage
1	Autodestruct	3	3	100%
2	Delay OnOff	3	0	100%
3	Anti-windup Integrator	3	3	100%
4	Hysteresis	4	9	100%
5	Priority	32	0	100%
6	Window Counter	4	0	100%
7	On ground circuit	68	0	100%

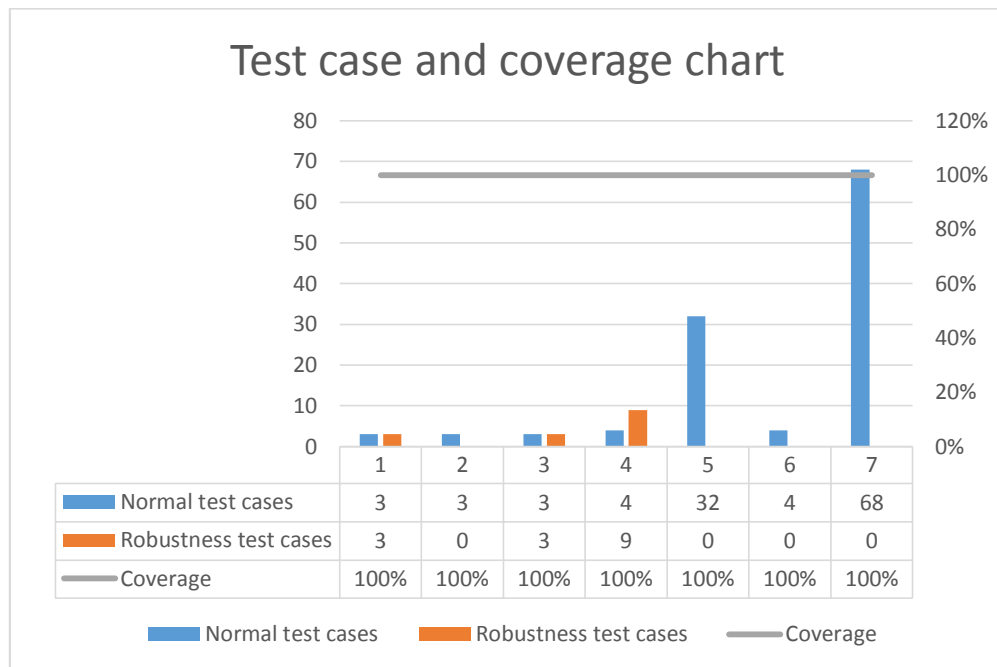


Figure 2.12 Simulink Model test case and coverage result

Table 2.5 shows that all models have 100% coverage which means we did achieve the coverage goal specified in DO-178C.

Figure 2.12 shows that our tool generated robustness test cases for the model. The missing robustness test cases for 2, 5, 6, and 7 is because all the inputs in these models are Boolean, and there is no way we can assign an out of range value to a Boolean value. Thus, our tool also achieved the normal range and robustness test case criteria.

#### *2.4.7 Tool qualification*

To save time and achieve better accuracy, software tools are widely used to help engineers develop code, compile code, and even find code bugs. In this thesis, a tool is an automated program to help engineers automate part of the verification work defined in Section 6.0 of the DO-178C standard.

DO-178C distinguishes two types of tools:

1. Development tools: a development tool is a tool for helping engineers develop target systems. Such tools include IDE, MATLAB, Simulink, and compilers.
2. Verification tools: A verification tool is a tool for detecting potential bugs inside a software system. Such tools include Pylint, VectorCast, and other bug-detection tools. In this thesis, both tools described in Chapters 2 and 3 are verification tools.

Qualifying a tool is a time-consuming process, and even a minor update of the tool requires re-qualification. Thus, when we design a tool, we try to avoid the necessity of qualifying the tool.

DO-330 (RTCA 2012) describes the processes and objectives for qualifying tools. However, we can use Figure 2.13 to determine whether tool qualification is needed. The first branch is “YES”, and the second branch is “NO”.

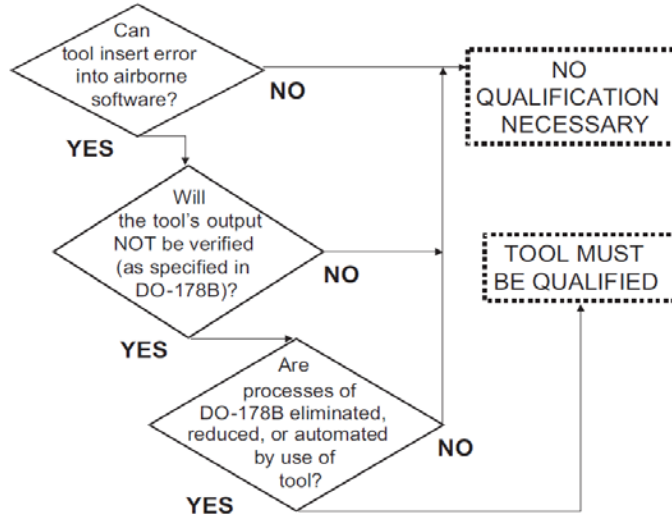


Figure 2.13 Tool qualification process (Kornecki and Zalewski 2006)

The test cases generated from our tool will still be sent to peer review. Thus, the tool in Chapter 2 does not need to be qualified.

## CHAPTER 3. PYDETECTOR - STATIC CODE ANALYSIS ENGINE

### 3.1 Motivation

Second only to writing test cases, writing a test procedure and making it pass is also a time-consuming process. Usually, one requirement points to one test case and each test case will have multiple test steps. One Automated Test Procedure (ATP) usually contains several test cases and could take a great deal of time to execute. For example, according to Elbaum's work (Elbaum, Malishevsky and Rothermel 2002), 135 test cases were generated to cover 69% of functionality in QTB (an embedded real-time system) and it took 27 days to execute those test cases. During execution, an error in the ATP would result in a crash and the ATP would have to be rerun from the beginning. Thus developing a static code analysis tool to detect errors before execution would be very useful and could save thousands of hours of testing.

### 3.2 Related works

Python is an easy-to-learn dynamic typed language. Because of its more flexible and less strict coding style. Python is the most popular language for writing ATPs.

We examined various open-source Python code analysis tools, and selected the three most popular tools among the Python community: PyLint, PyFlakes and PyChecker, PySonar 2. In the next few subsections, we will introduce these three tools:

#### 3.2.1 *PyLint*

PyLint ( Logilab and contributors 2014) is a Python semantic error checker and also a coding standard checker. This tool is implemented in Python and was first created in 2003 at Logilab. Many open-source projects have been created to integrate PyLint into many IDEs (e.g., Eclipse, Sublime text).



It is easy to install PyLint on all kinds of system distributions ( Logilab and contributors 2014).

On OpenSUSE, for example, we can run the following commands

```
sudo zypper install pylint # Python2.7  
  
sudo zypper install Python3-pylint
```

On Mac OS X and Windows, we can use pip to install Python with the following command:

```
pip install pylint
```

PyLint supports the following features:

- Coding Standard (PEP 8 style)
- Error detection
- Refactoring help
- Fully customizable
- Editor integration
- IDE integration
- UML diagrams
- Continuous integration
- Extensibility

The PyLint tool is a static code analyzer. The common issue for a static code analyzer is significant difficulty in capturing type errors. From PyLint's specification, it claims rule R0923 can detect whether or not an interface is implemented, and many other rules for detecting various error messages in the Python code are also supported.

However, we show that in some cases PyLint fails to detect the bug, but that our approach does find it. We will thus use the following example to test PyLint:

```
class signal:

    def __init__(self):

        self.rate=100

        self.name = ""

    def initialize(self):

        self.initialized=True

a=signal()

print a.test1 # Error showed, OK

a.test2=10

print a.test2 # no warning showed, OK

print a.test3 # no warning showed, ERROR

a.test3=20

print a.initialized #no warning showed, ERROR

print len(a) # len(a) does not make sense, however, no warning showed
```

The result produced by the PyLint analysis was a disappointment. Despite the claims stated on its website, this tool could not properly analyze the simple (though not trivial) example above. What good does PyLint probably do? A dynamic addition of an attribute to an object is detected without any problems. A reading context of such a dynamically-added attribute is also properly processed. What is PyLint's issue in this example? Almost certainly, PyLint does not know in what sequence these statements will

be executed, and in some cases it cannot even determine which statements are executed. This is a very important observation that will be used later in proposing our solution.

### 3.2.2 PyFlakes

PyFlakes (PyFlakes 2014), similar to PyLint, is another static code analyzer. The installation of PyFlakes is as easy as that of PyLint. There have also been multiple projects trying to integrate PyFlakes into IDEs.

The aim of pyFlakes is to quickly detect common errors without dynamically executing them. In addition to detecting common errors in Python, pyFlakes brother project Flake8 can also check PEP8 code style in a manner very similar to PyLint.

PyFlakes can detect the following issues:

1. Unused imports
2. Redefinition of an unused variable
3. Global variable shadowed in a loop or function
4. Undefined variable
5. Syntax error in doctest
6. Local variable defined in encoding scope referenced before assignment
7. Local variable assigned but never used
8. Duplicate argument in function definition

Although these errors seem naive, they are rather common and could show up when a variable name is mistyped, when an import is missed, or when a function with the same name and argument is overwritten. We show in some cases that PyFlakes fails to detect an unused local variable. We will test PyFlakes using the following example:

```
import sys
```

```
def foo(a, b):  
    c = 10  
    return a - b  
  
d = 20  
  
foo(a)
```

PyFlakes detected unused import, the unused variable `d` and the undefined variable `a`. However, the unused variable `c` was not detected. Both PyLint and PyFlakes use similar technology to detect errors, and neither needs to execute the Python script to detect these issues. The next tool we introduce will use a different principle for error detection.

### 3.2.3 PyChecker

In contrast to PyLint and pyFlakes, PyChecker (PyChecker 2015) is a dynamic code analyzer. PyChecker overcomes the defects of static code analyzers by dynamically running the code. Although it is very effective in detecting type errors, its drawbacks are also obvious.

Consider the following example as an input for PyChecker:

```
import sys  
import pygear  
  
raw_input("The following test will examine if terrain warning will alert when altitude is  
below 1000 feet")  
  
while altitude >= 0:  
    pygear.set_altitude(altitude)
```

```
raw_input("Press Y if alert has sound")
altitude = altitude + 1000
```

PyChecker will wait on the raw input statement for an infinite time. The reason for this is that PyChecker tries to execute the above statements and when it reaches any statement that requires user input, PyChecker will stop and wait for the user's response.

Another example shows that, even without any input function, PyChecker may still run into an infinite loop.

```
i = 10
while i > 1:
    print Developer accidentally used x>1 instead of x < 1
    i = i + 1
k = i + j
```

When checking the above code, PyChecker will be stuck in the while loop and will not give us any information about the undefined variable j.

The following example shows still another problem:

```
import sys
import pygear
def get_Altitude(FMS)
    if(FMS == "FMS1"):
        return pygear.FMS1.get_Altitude();
    elif(FMS == "FMS2"):
        return pygear.FMS2.get_Altitude();
    elif(FMS == "FMS3"):
```

```
return pygear.FMS3.get_Altitude();
```

```
else:
```

```
raise Exception("Please select a correct Flight management system")
```

Altitude = get\_Altitude("FMS5"); #pycheck reports an exceptio here which is what we designed for

Altitude = get\_Altitude("FMS1", "FMS2") #too many arguments, pychecker didn't report

Because PyChecker reports an error when the Python interpreter returns an error exception, the first occurrence of an error is correctly logged. However, PyChecker has no recovery system to continue execution after an exception is reported, so the next error message will not be generated until all previous errors are fixed. The advantage of PyChecker is that it gives a very sound result, i.e., if a code has passed on PyChecker, the code will not fail during run time.

As a dynamic code analyzer, PyChecker also has the following disadvantages:

1. It can only detect errors in the code it has executed
2. It cannot detect dead code
3. Only the first fatal error can be captured. PyChecker does not recover the execution after an exception has occurred.
4. PyChecker might take a long time to run the code or even run in an infinite loop

In the next section, we show the motivation for our work that is strongly related to insufficient results produced by related current work.

### 3.2.4 PySonar 2

PySonar2 (Wang 2014) is a widely used Python type inference and indexer. It infers types of variables by performing inter-procedural analysis. PySonar had been tested on million lines of code when internally used at Google (Wang 2014)

The purpose of PySonar is to infer the type of a variable at a certain program point, but the tool does not report any type error. Our tool PyDetector performs both type analysis and attribute analysis. PyDetector can do the following analysis that PySonar does not:

- PyDetector analyzes for the logical operation ( $>$ ,  $<$ ,  $=$ ) and Math operation ( $+$ ,  $-$ ,  $\div$ ,  $\times$ ) that two or more variables are of the same type.
- PyDetector analyzes that a function call from an imported library exists.
- PyDetector analyzes that parameters passed into the function are correct (correct number of parameters and correct type for each parameter)

With the above analysis, our tool PyDetector is capable of detecting code issues defined in Section 3.4.

## 3.3 Background

From Section 3.2, we can see the static analysis and dynamic analysis each has both advantages and disadvantages. In general, static analysis is more useful and safer for users (programmers). On the other hand, dynamic analysis provides the most accurate results in exchange for safety and running the entire body of code while checking it. Moreover, it may cause unpleasant side effects, such as when working with a database.

In avionics, it is very difficult to perform dynamic analysis, because simulating a test script requires execution on a special simulation system such as Virtual integrated

software testbed for avionics (VISTA) (Magor and Stodola 1994), and it might take a test script up to 10 hours or more before it completes. Also, if a fatal error occurs or a failure is captured, we must restart the entire script. We aim to design a static Python-based analysis tool to detect potential issues inside an ATP.

### 3.4 Problem definition

To avoid crashes or failures in test scripts, we design a static analysis tool PyDetector to detect issues possibly leading to a software crash. We categorize the issues to be examined into the following types:

- Syntax issue
- Non-existent import
- Globals are not found (using a global import before importing it)
- Wrong number of parameters passed into functions
- Too many or too few arguments passed for string format
- Using methods and attributes that don't exist in imports
- Changing signature when overriding a method
- Redefining local functions
- Using an undefined or uninitialized variable
- Variables / imports / functions defined but not used
- Argument in a function defined but not used
- Missing doc string for functions

Moreover, according to DO-330, any tool that automates any process of DO-178B must be qualified, and qualifying a tool requires significant efforts. We thus design a



rule-based static code analysis tool and implement each rule as a plugin. In this way, when a new rule is added, only that rule requires qualification.

### 3.5 System architecture

Once a user submits an ATP, our analyzer will first generate an AST (Abstract syntax tree) from the file. The AST will be sent to the Python compiler to check syntax issues, and the file will then be passed through a chain of rules. Each rule will detect issues in the file to be attached to a global issue list.

The architecture of our tool PyDetector conforms to Figure 3.1:

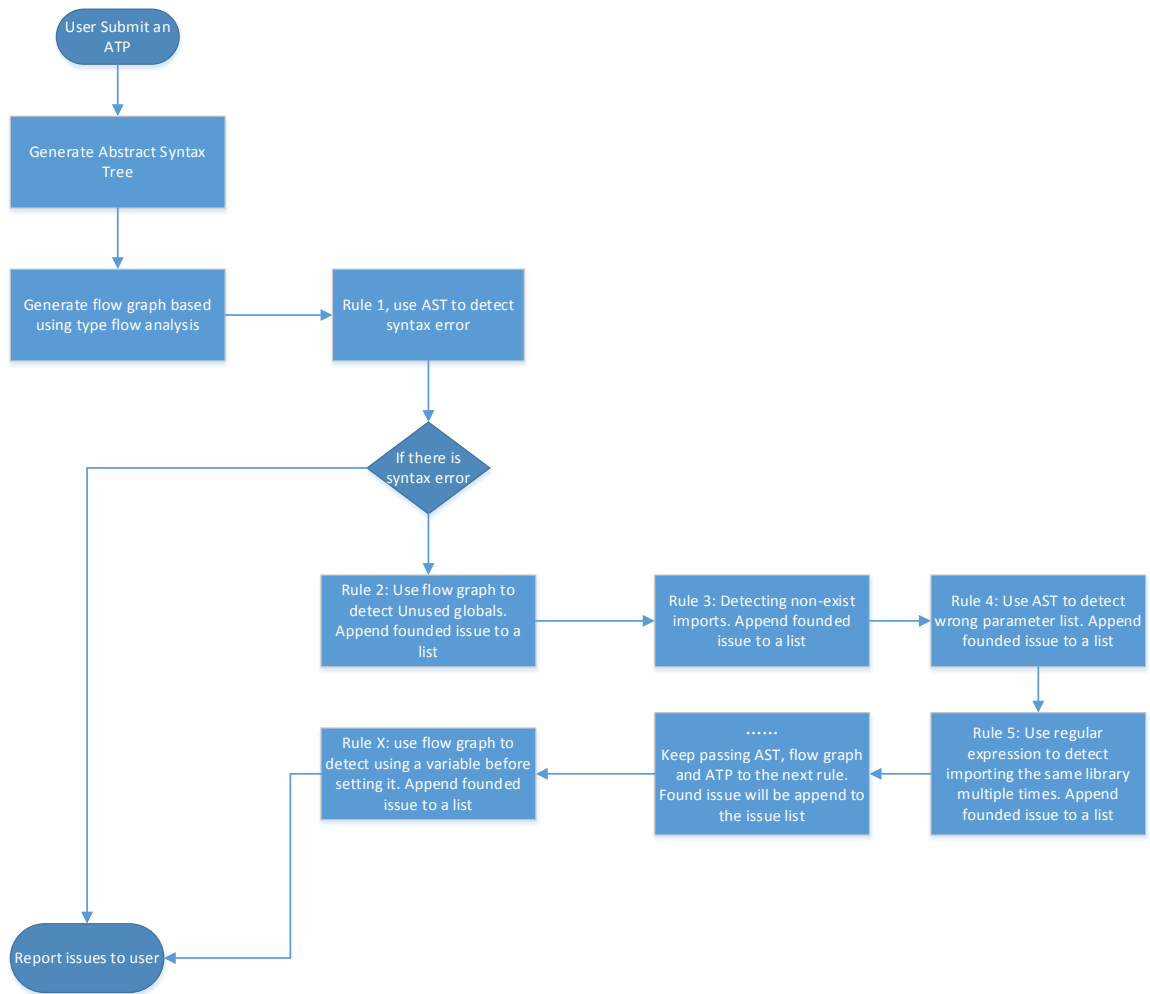


Figure 3.1 PyDetector architecture

It is very important to convert a Python file into AST. With AST, we can iterate every line of code and check the type of each variable using type flow analysis. We will discuss what information AST has and how type flow analysis is performed on AST in next section.

### 3.6 Static code analysis

The fatal issues listed in Section 3.3 are caused by type or attribute errors. We will use "type-flow analysis" to detect such errors before actually running the ATP. Type-flow analysis is similar to data-flow analysis, but before performing type-flow analysis, we will convert the Python program to an abstract syntax tree (AST) so that we can easily iterate the code.

The abstract syntax tree (AST) is a tree representation of the Python source code. Each node is an abstract syntactic structure. Python library has a built-in AST module that developers can use to obtain the AST from the source code. In the next few sections, we introduce how a Python file can be mapped to an AST and what is in the AST.

There are three available compilation modes for each Python file:

- Exec
- Eval
- Single

The root node of the AST will differ based on the different compilation modes. For exec mode, the root node will be an `ast.module`; for eval mode, the root will be an `ast.expression`; and for single mode the root will be an `ast.Interactive`.

All other nodes in the AST fall into one of the following groups:

- Literals

- Variables
- Expressions
- Statements
- Control flow
- Function and class definitions

### *3.6.1 Literal*

Literals are a set of nodes that deal with constants in the AST.

- `ast.Num`

This node will represent all numbers in the Python file. This includes integer, float, or complex values.

- `ast.Str`

This node represents any string in the Python file

- `ast.Bytes`

This node is a bytes object. A bytes object can be returned by reading a file in binary mode. This node only exists in Python 3 or higher versions

- `ast.List`, `ast.Tuple`, `ast.Set`, `ast.Dict`

These nodes represent List, Tuple, Set and dictionary types respectively.

- `ast.Ellipsis`

This node represents the “...” syntax for an Ellipsis singleton.

### *3.6.2 Variables*

- `ast.Name`

This node will store a variable name

- `ast.Starred`

This node stores a variable reference. Variable references can load the value of another variable.

### *3.6.3 Expressions*

Expressions are an extensive category of nodes that encapsulate all the work with expressions in Python. The main node is `ast.Expr`. Within this node, all operations are executed. The kind of operations may be recognized by a nested node of the operation, with the following possibilities:

- `ast.UnaryOp` - unary operation
- `ast.UAdd`, `USub`, `Not`, `Invert` - unary operator tokens. `Not` is the `not` keyword in Python. `Invert` is “~” operator
- `ast.BinOp` - Binary operations
- `ast.Add`, `Sub`, `Mult`, `Div`, `FloorDiv`, `Mod`, `pow`, `LShift`, `RShift`, `BitOr`, `BitXOR`, `BitAnd` - Binary operator tokens
- `ast.BoolOp` - Boolean operator.

### *3.6.4 Statement*

The group of the statements consists of:

- `ast.Assign` - This represents assigning value to variable operation in Python
- `ast.AugAssign` - This represents Python syntax like `a+=1`
- `ast.Print` - This represents a print statement
- `ast.Raise` - This represents raising an exception syntax
- `ast.Pass` - This represents a pass statement in Python

### *3.6.5 Imports*

- `ast.Import`

- `ast.ImportFrom`
- `ast.alias`

These three nodes represent Python syntax “import library”, “from library import module” and “import library as alias”

### 3.6.6 Control flow

- `ast.If`
- `ast.For`, `ast.While`
- `ast.Break`, `ast.Continue`
- `ast.TryFinally`, `ast.TryExcept`
- `ast.ExceptHandler`

From the name of the node it is easy to see that these nodes represent the “if”, “for”, “while”, “break”, “continue”, “try... except ... finally” statements in Python.

Each line of Python code can be broken into a multiple expression or statement and each statement is a node in the tree. Each node also stores all AST information listed above: type information is most important for us during type flow analysis. We can figure out the type of a current node from its adjacent nodes.

Consider the following example:

```
var1 = "Yijia"
```

```
var2 = 3
```

```
var3 = var1 * var2
```

```
var4 = var1 + var2
```

From the above statements, PyDetector constructs the abstract syntax tree shown below:

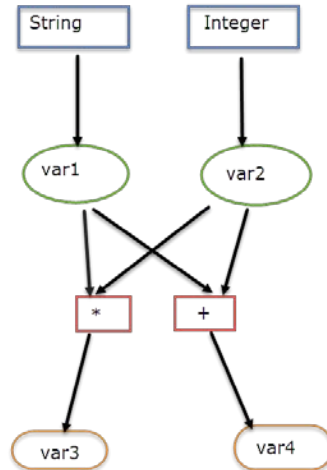


Figure 3.2 Graph converted from the code above

Nodes var3 and var4 in the bottom of Figure 3.2, converted from string “Yijia” and integer number 3, are built-in Python types. The green nodes on the first row are the defined variables. We have assigned a string type to var1 and an Integer type to var2. The two red color nodes marked as “\*” and “+” on the third row are expression nodes; they store the kind of expression required from the two input nodes. The brown nodes on the fourth row are the final output nodes.

When we do type flow analysis, we can see  $\text{var3} = \text{String} * \text{Integer}$ . This operation is supported by Python and the returned type is String, so we will store type = “String” at node var3 and  $\text{var4} = \text{String} + \text{Integer}$ . This operation is NOT supported by Python; an error will be reported for this operation.

Next consider an example involving a function:

```

def func1(x):
    return x*2

f1 = func1("a")
f2 = func1(1)
  
```

The above code will generate the following abstract syntax tree shown as in

Figure 3.3:

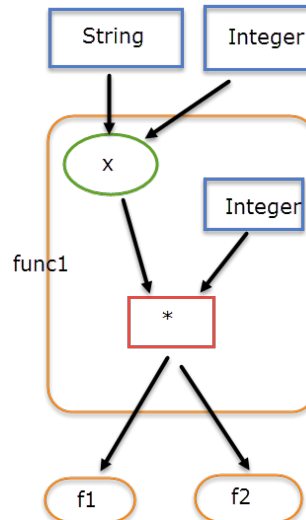


Figure 3.3 Graph converted from the code above

The big box marked as func1 in the middle of Figure 3.3 represents function “func1” and the green node “x” is its parameter. This function has been called twice and a string and then an Integer are passed as parameters for the two calls. Then node x will multiply by another Integer and return a value to f1 and f2 respectively. From type-flow analysis, we can see that  $f1 = \text{String} * \text{Integer}$  and  $f2 = \text{Integer} * \text{Integer}$ . Thus f1 is a string and f2 is an Integer.

By this method, all Python built-in types can be correctly analyzed. However, if an engineer defined two custom classes and overwrote the “+”, “-”, “\*”, “/” operators, our approach cannot successfully perform an analysis. At this time, none of the open source static code analyzers can yet perform this analysis. The way to mitigate this drawback is to apply a dynamic code analyzer technology method and let the analyzer

become a hybrid analyzer. However, this represents future work not included in this thesis.

### 3.7 Evaluation

In this section, we will set up experiments to compare the performance of our tool to that of PyFlakes. As discussed in Section 3.2. PyFlakes is also a static code analyzer. We use a set of sample test scripts to compare the performance of the two tools.

The experiment analyzed 139 Python scripts with a total of 40840 lines of code. These scripts are test scripts designed in open source project pyChecker (Neal 2013) to test Python Code Analyzer. We run PyFlakes against each file and categorize issues it reported. Table 3.1 shows all the errors reported by PyFlakes.

Table 3.1 PyFlakes analysis result

Python Script	Number of issues	Type of issue
exceptions1.py	1	UnusedVariable
func1.py	1	UnusedImport
sequence.py	4	UnusedVariable
test1.py	9	UnusedImport UndefinedName UnusedVariable
test13.py	3	UndefinedName
test14.py	1	UnusedImport
test2.py	1	UnusedImport
test22.py	5	UnusedVariable
test24.py	1	ImportStarUsed
test3.py	1	UndefinedName
test33.py	2	UndefinedName UnusedVariable
test35.py	1	UnusedImport
test36.py	2	UnusedImport
test37.py	1	UnusedImport
test39.py	2	UnusedVariable
test41.py	3	RedefinedWhileUnused
test44.py	1	ImportStarUsed
test45.py	5	UnusedVariable
test46.py	4	UnusedImport RedefinedWhileUnused ImportStarUsed
test54.py	3	UndefinedName
test55.py	1	UnusedVariable
test58.py	1	Invalid syntax
test64.py	3	UnusedImport
test73.py	3	UnusedVariable
test8.py	1	Invalid syntax
test83.py	8	UnusedVariable
test97.py	1	UnusedImport



Table 3.1 shows that lacking type flow analysis, PyFlakes only detected a total of 33 issues, out of which, only 6 of them can cause a program to crash.

Figure 3.4 compares how many errors our tool PyDetector detects from the Python scripts as compared to how many PyFlakes detects:

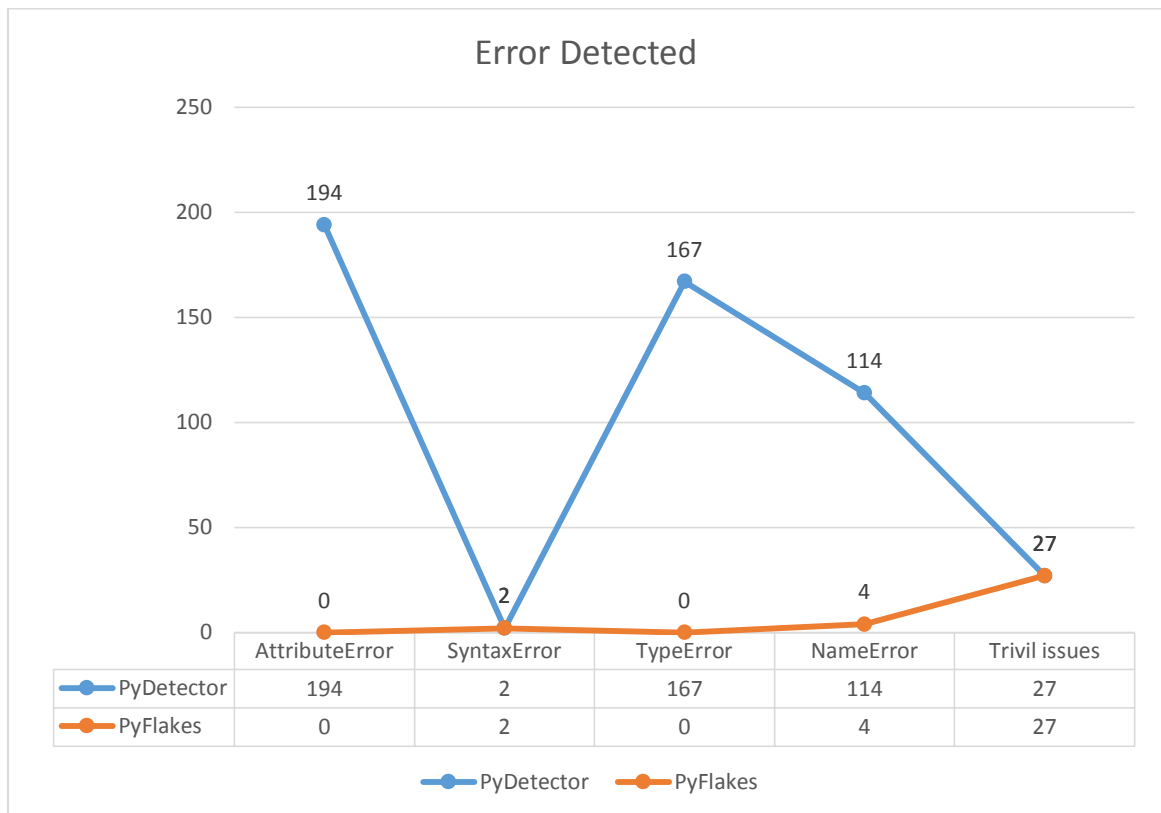


Figure 3.4 PyFlakes and PyDetector evaluation result

After applying type flow analysis, our program, PyDetector, can find a total of 502 issues. And of these, 475 can cause a program to crash. The above figure shows our program PyDetector can detect 15 times more errors than PyFlakes.

PyFlakes cannot deal with most issues in the file because PyFlakes only scans the file we submitted. It does not load the imported library file but simply assumes the library file exists and has all the parameters passed in correctly. In contrast, our PyDetector tool

will search and load the library file to make sure that the expected function is there and has the correct parameters.

### 3.8 Tool qualification

In DO-178C, after the test developer writes the test script, a reviewer must review the ATP to make sure it is working correctly. By applying PyDetector, a reviewer can get a list of issues in the ATP. However, a reviewer still needs to go through all the issues to make sure that none of them are false positive. More importantly, a reviewer must review ATP to make sure all test cases have been implemented in the ATP. Thus, the tool does not need to be qualified since none of the DO-178C processes has been replaced by PyDetector.

## CHAPTER 4. CONCLUSIONS AND FUTURE WORK

In Chapter 2, we proposed a new methodology to generate test cases from a Simulink Model. First, we designed an m-script to decode test cases generated by a Simulink Design Verifier and exported them into a CSV-format test-case document. Second, we converted the model into a dependency graph and from this graph we developed our test cases.

To avoid requirement conflicts, for each output signal, we must store a constraint calculated from its input. When generating a test case for another model that depends on that constraint, we submit it to a constraint solver and determine whether the stimulus can be achieved under the current constraint.

For future work, we state two goals:

1. In our current work, we use a Simulink Design Verifier to provide us with test cases for state flow. Our Simulink Analyze Engine does not analyze state flow, an important part of Simulink. We expect to develop state- flow support in the near future to provide more accurate test case results.
2. In our current implementation, we only handle a limited number of blocks. While our analysis indicates that the existing blocks already support most models in display systems, we will develop support for more blocks in the future for use in analyzing control systems.

In Chapter 3, we developed a tool, PyDetector, to deal with dynamically-typed languages. The purpose of that tool was to detect errors that could crash the execution of Automated Test Procedure (ATP). Three technologies have been introduced in our solution.

We used Abstract Syntax Tree (AST), a tree representation of the abstract syntactic structure of source code. In AST, “each node of the tree denotes a construct occurring in the source code” (Wikipedians 2012). If any syntax issue exists in an ATP, an error will occur when constructing the AST. Constructing the AST is essential for the subsequent static code analysis.

Type flow analysis is an enhancement to the Abstract Syntax Tree. It does not require running the Python program to obtain the variable type at each program point. We combined the safety of static analysis with the power and accuracy of dynamic analysis and, as a result, achieved a powerful approach with significant potential. The demonstration of the correctness of our idea was accomplished by implementation of this solution in Python.

Our tool provides a wide range of error detection for Python scripts. In Chapter 3, we compared our tool with PyFlakes, another well-known static code analysis engine. We listed 12 test items supported by our tool, PyDetector, only 4 of which are supported in PyFlakes. This comparison shows the relative power of our tool and, within the discipline of error detection that our tool improves on other open source solutions.

The future plans for PyDetector include support of a larger number of Python standard libraries. Unlike a user library file, Python standard libraries are in pyc format, and we have no access to the source code. However, we can analyze the library files from Python source code and dump them into a saved file. Although it is still not clear whether or not this solution is feasible, it should be very interesting and challenging to support such functionality in the future.

## BIBLIOGRAPHY

- Autosar. 2014. *Autosar*. Accessed 05 15, 2015. <http://www.autosar.org/>.
- Chauhan, Naresh. 2010. *Software Testing Principles and Practices*. New Delhi: Oxford University Press.
- Deißenböck, Florian. 2015. *Simulink Library for Java*.  
<https://www.cqse.eu/en/products/simulink-library-for-java/overview/>.
2015. *Eclipse*. Accessed 05 15, 2015. <https://eclipse.org/>.
- Elbaum, Sebastian, Alexey G. Malishevsky, and Gregg Rothermel. 2002. "Test Case Prioritization: A Family of Empirical Studies." *Software Engineering, IEEE Transactions on* 28.2 159-182.
- Etienne, J-F, S. Fechter, and E. Juppeaux. 2010. "Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain." *Complex Systems Design & Management* 61-72.
- Gadkari, Ambar A., Swarup Mohalik, K.C. Shashidhar, Yeolekar Anand, J. Suresh, and S. Ramesh. 2007. "Automatic generation of test-cases using model checking for sl/sf models." *MoDeVVa workshop Model-Driven Engineering, Verification and Validation* 33.
2015. *Graphviz*. Accessed 05 15, 2015. <http://www.graphviz.org/>.
- Hayhurst, Kelly J., Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. 2001. "A Practical Tutorial on Modified Condition/ Decision Coverage." NASA.  
<http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf>.
- Huizinga, Dorota, and Adam Kolawa. 2007. *Automated Defect Prevention: Best Practices in Software Management*. John Wiley & Sons.
- Indian Student Association . 2012. *Software testing*. <http://isa.unomaha.edu/wp-content/uploads/2012/08/Software-testing.pdf?>
- Jeppu, Natasha. 2015. *Exploring Design Verifier*.  
<http://www.mathworks.com/matlabcentral/fileexchange/48858-exploring-design-verifier>.
- Kornecki, Andrew J., and Janusz Zalewski. 2006. "The Qualification of Software Development Tools From the DO-178B Certification Perspective." *CrossTalk* 19.4 19-22.
- Leveson, Nancy. 1995. *Safeware: System Safety and Computers*. Addison-Wesley Professional.
- Li, Meng, and Ratnesh Kumar. 2012. "Model-based automatic test generation for simulink/stateflow using extended finite automaton." *Automation Science and Engineering (CASE)*,. IEEE International Conference on. IEEE,. 857-862.
- Logilab and contributors. 2014. "Pylint User Manual." *Pylint 1.2.0 documentation* ». Accessed 05 15, 2015. <http://docs.pylint.org/>.

- Magor, Wayne E., and Ken M. Stodola. 1994. Virtual integrated software testbed for avionics. USA Patent US 5541863 A. Sep 30.
- MathWorks. 2015. *Simulink - Simulation and Model Design*. Accessed 05 15, 2015. <http://www.mathworks.com/products/simulink/?refresh=true>.
- MathWorks. 2015. *Simulink Design Verifier - Identify design errors, generate test cases, and verify designs against requirements*. Accessed 05 15, 2015. <http://www.mathworks.com/products/sldesignverifier/>.
- MathWorks. 2015. *Simulink Design Verifier*. Accessed 05 15, 2015. <http://www.mathworks.com/help/sldv/examples/using-existing-coverage-data-during-subsystem-analysis.html?prodcode=DV&language=en>.
- MathsWorks. 2015. *Simulink Design Verifier Features*. Accessed 05 15, 2015. <http://www.mathworks.com/products/sldesignverifier/features.html>.
- MISRA Consortium. 2012. *MISRA C:2012*. Accessed 05 15, 2015. <http://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/Default.aspx>.
- Neal. 2013. *PyChecker*. Accessed 05 15, 2015. <http://pychecker.sourceforge.net/>.
- Pan, Jiantao. 1999. "Software testing." *Dependable Embedded Systems*.
- Panesar-Walawege, Rajwinder Kaur, Mehrdad Sabetzadeh, and Lionel Briand. 2013. "Supporting the verification of compliance to safety standards via model-driven engineering: Approach, tool-support and empirical validation." *Information and Software Technology* 836-864.
- PyChecker. 2015. "PyChecker: a python source code check tool." Accessed 05 15, 2015. <http://pychecker.sourceforge.net/>.
- PyFlakes. 2014. "PyFlakes." Accessed 05 15, 2015. <https://github.com/pyflakes/pyflakes/>.
- R. Systems. 2012. *Software Testing and Validation with Reactis*. Accessed 05 15, 2015. <http://www.reactive-systems.com/>.
2015. *Reactis*. Accessed 05 15, 2015. <http://www.reactive-systems.com/>.
- RTCA. 2012. *DO-330 Software Tool Qualification Considerations*. RTCA.
- RTCA. 2012. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA SC-205.
- Schätz, Bernhard, Alexander Pretschner, Franz Huber, and Jan Philipps. 2002. "Model-based development of embedded systems." *Advances in Object-Oriented Information Systems* (Springer) 298.
2015. *TestWeaver*. Accessed 05 15, 2015. <http://www.qtronic.com/en/weaver.html>.
2015. *V model*. Accessed 05 15, 2015. [http://en.wikipedia.org/wiki/V-Model\\_\(software\\_development\)](http://en.wikipedia.org/wiki/V-Model_(software_development)).

- Wang, Yin. 2014. *PySonar2 - a type inferencer and indexer for Python*.  
<https://github.com/yinwang0/pysonar2>.
- Whalen, Michael, Gregory Gay, Dongjiang You, and Matt Staats. 2013. "Observable modified condition/decision coverage." *Proceedings of the 2013 International Conference on Software Engineering* 102-111.
- Wikipedians. 2012. *Compiler Construction*. PediaPress.
2015. *WindowBuilder*. <https://eclipse.org/windowbuilder/>.
- Zhan, Yuan. 2005. "A search-based framework for automatic test-set generation for matlab/simulink models." *Software Eng. SE-10, vol. PhD thesis*.