Black-Box Test Case Generation from TFM Module Interface Specifications and Usage Statistics

Katsuya Iwata

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Iwata, Katsuya, "Black-Box Test Case Generation from TFM Module Interface Specifications and Usage Statistics" (2012). Graduate Theses and Dissertations. 12613.
http://lib.dr.iastate.edu/etd/12613

This Thesis is brought to you for free and open access by the Graduate College 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.
Black-box test case generation from TFM module interface specifications and usage statistics

by

Katsuya Iwata

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:
David M. Weiss, Major Professor
Andrew S. Miner
Srikanta Tirthapura

Iowa State University
Ames, Iowa
2012

Copyright © Katsuya Iwata, 2012. All rights reserved.
DEDICATION

I dedicate this thesis to my wife Harumi without whose support I would not have been able to complete this work. I also dedicate this thesis to my family, friends, and coworkers in Canon Inc. whose loving attention, thought, and guidance encouraged me before and during my graduate education.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>vi</td>
</tr>
<tr>
<td><strong>CHAPTER 1. INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Goal of this thesis</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Structure of this thesis</td>
<td>3</td>
</tr>
<tr>
<td><strong>CHAPTER 2. BACKGROUND</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Software Testing Technique</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Formal Specification Technique</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Statistical Software Testing Technique</td>
<td>17</td>
</tr>
<tr>
<td><strong>CHAPTER 3. APPROACH</strong></td>
<td>21</td>
</tr>
<tr>
<td>3.1 TFM Module Interface Document for Component Testing and Integration Testing</td>
<td>23</td>
</tr>
<tr>
<td>3.1.1 Assumptions - Module Interface Specification</td>
<td>23</td>
</tr>
<tr>
<td>3.1.2 Module Interaction Trace Specification</td>
<td>24</td>
</tr>
<tr>
<td>3.1.3 Input and Output Specification</td>
<td>26</td>
</tr>
<tr>
<td>3.2 Statistics Document Development</td>
<td>29</td>
</tr>
<tr>
<td>3.3 Testing Trace Generation Method</td>
<td>31</td>
</tr>
<tr>
<td>3.3.1 Trace Postconditions</td>
<td>31</td>
</tr>
<tr>
<td>3.3.2 Monitored Variable Constraints</td>
<td>32</td>
</tr>
<tr>
<td>3.3.3 Test Case Generation Algorithm</td>
<td>33</td>
</tr>
<tr>
<td><strong>CHAPTER 4. EVALUATION</strong></td>
<td>37</td>
</tr>
<tr>
<td>4.1 Feasibility of Test Case Generation - Example Application: Floating Weather Station</td>
<td>37</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Test Procedure</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Functions and Notations for Predicates (See Additional Definitions in Appendix A)</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Closed-form Solution for value - Conditional Expressions Format</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>TFM Module Interface Document for a Stack Module</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Closed-form Solution for pushes - Conditional Expressions Format</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of Test Case Generation Process</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Overview of Testing Process</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Primitive Functions on Event Descriptors</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Upper-Face Only Module Interface</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Module Interface Specification</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Output Delegation</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Output Behaviors Definition of Module B with Statistics Annotations</td>
<td>30</td>
</tr>
<tr>
<td>3.8</td>
<td>Monitored Variable Constraints</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>FWS simulator GUI</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Generated Testing Trace (Test Case)</td>
<td>39</td>
</tr>
<tr>
<td>B.1</td>
<td>FWS Module Hierarchical Structure</td>
<td>47</td>
</tr>
<tr>
<td>B.2</td>
<td>FWS Uses Relation</td>
<td>47</td>
</tr>
<tr>
<td>C.1</td>
<td>SensorDriver module Before Transpose</td>
<td>49</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. David M. Weiss for his warm advice and guidance throughout this research and the writing of this thesis. His precise insights and extensive knowledge significantly enhanced my work. I also thank Ms. Laurel Tweed for her advice and suggestions on my writing. Her advice profoundly improved the structure and language of this thesis. I also thank Canon Inc. for giving the opportunity of studying computer science and the financial assistance during the writing of this work. I also thank my committee members for their efforts and contributions to this work: Dr. Andrew S. Miner and Dr. Srikanta Tirthapura. I additionally thank Dr. Hridesh Rajan for his guidance during my initial stage of my computer science study and Dr. Andrew S. Miner for his inspiring lecture.
CHAPTER 1. INTRODUCTION

This chapter explains the centrality of verification and testing to the software industry, the contribution of this thesis, and an overview of related work.

1.1 Goal of this thesis

Society is coming to use and rely more and more on computer systems, such as safety critical systems, business systems, home appliance systems, entertainment systems, and mobile systems. Our dependence on such computer systems raises people’s awareness of the social importance of the software quality in these systems. The quality of software is eventually determined when a product is released. A product manager usually has to make the decision on when to release a product using collected software quality data; however, because software is substantially complex, methods for measuring software quality are sometimes inadequate. Consequently, a product manager may be pressured to make a release decision to meet the product release schedule with uncertain software quality. This thesis proposes a statistical software testing technique [23, 42, 24, 25] to provide a rationale for software quality to help make this management decision.

Software quality is measured with verification and validation. Verification is a method used to ensure that a product is built correctly. Validation is a method to ensure that the correct product is built [6]. While both methods are important, this thesis will focus on verification. Well-known software verification techniques are testing, review, program proving, and model checking. This thesis will focus on testing. The number and importance of test cases that are performed in testing and the absence of errors during executing those test cases gives a measure of the confidence of software quality.
The problem of testing is that the combinatorial number of input values is exponential on the sum of the number of input variable bits, which is large; thus, testing all combinations of input values is usually impractical. Hence, many testing approaches to determine important testing input value combinations have been proposed. The black-box testing techniques [5, 26] in particular deliver benefits for testing large and complex software, since they only use specifications that abstract away details of source code.

The first issue of black-box testing is how to identify use cases to test. A use case is a detailed description of a sequence of transactions between system and actors that exist outside the system. Black-box testing lists testing input patterns that can be derived from specifications, such as requirement specifications or module interface specifications. In industry, testers usually identify use cases with the prediction of possible usages of the product; however, since testers perform the usage prediction individually and ad-hoc, the prediction depends on the testers’ individual skills and preferences, which can cause uncertain software quality. Several researchers have suggested methods for improving this use cases identification. Jacobson [16] introduced a systematic process to identify use cases. His process incorporated UML use case diagrams and produced the use cases. However, since the description of use cases is written in a natural language, which has ambiguity, the exhaustive listing of all use cases is difficult. Some researchers proposed identification methods using formal specifications such as Z[2], VDM [1], and state machine [8]. By utilizing its formalism, they could generate an exhaustive list of use cases that are specified. However, since most specification languages are difficult to write because of their complicated model construction, they are rarely adopted in industry. Among them, Trace Function Method (TFM) is reportedly beneficial [33, 3, 37, 29]. It is formal but relatively intuitive, conforms to the information hiding principle [31], and has reader-friendly tabular format. In this thesis, we identify use cases with TFM specifications.

The second issue of black-box testing is how to prioritize test cases. The exhaustive listing of use cases of a large software component yields an intractable number of use cases. Development organizations have a limited time for testing cases; thus, we have to prioritize them to reduce the number of test cases to perform. Whittaker [42, 9] modeled the system as a discrete-time Markov chain where the transitions are user operations, and generated prioritized test cases.
with a random choice of transitions in the Markov chain; however, they left the discussion on how to identify states and transitions of the Markov chain as an open question. Woit [43] generated prioritized test cases from a list of pairs (event, class of system’s executed events) with an occurrence probability. The event is an invocation of an access program of a component. Although she introduced classes of executed events, she left room to discuss how to identify those classes. Musa [24, 25] proposed the operational profile, which profiles each user operation with its occurrence probability and importance and prioritizes user operations to test. Although this approach is reportedly beneficial [25], the identification of the set of user operations relies on an informal refinement process, which makes completing an exhaustive list of user operations a complex task.

In this thesis, we propose a black-box testing method that derives important test cases with usage statistics, and enables a product manager to make a release decision with a rationale, “the important use cases specified in the usage statistics are tested and have no error.” First, we propose a method to specify components with TFM module interface specifications. Then, we propose a way to associate module usage statistics with the TFM module interface specification. Finally, we propose a method to generate a prioritized list of black-box test cases for component testing and integration testing [5] from the TFM module interface specification with usage statistics.

1.2 Structure of this thesis

This thesis consists of seven sections. Chapter 2 describes the background of this research. Chapter 3 explains the proposed technique. Chapter 4 evaluates the proposed technique with an example application and chapter 5.1 discusses the limitations and improvements. Chapter 5.2 describes related work and comparisons with the proposed technique. Finally, chapter 5.3 summarizes this research, and suggests areas for further research.
CHAPTER 2. BACKGROUND

2.1 Software Testing Technique

Testing approaches are usually categorized as follows: unit testing, component testing, integration testing, system testing, acceptance testing, and regression testing [5, 26, 11]. **Unit testing** checks the correctness of behaviors of the smallest piece of software that can be independently tested. **Component testing** checks the correctness of behaviors of an integrated assembly of one or more units/components and associated data objects. A **software unit** is usually a module\(^1\). A **component** is a set of modules. In this thesis, we consider that unit testing is a particular case of component testing; thus any method applicable to component testing is applicable to unit testing. **Integration testing** checks the correctness of interactions among the components such as inter-component calls, returns, and data handling. **System testing** checks the existence or absence of required properties including non-functional properties such as performance, usability, and security. A built product or a major component is tested in system testing since nonfunctional properties usually can only be tested after the product or major component is built. **Acceptance testing** checks the conformance of the system to the latest user needs. When requirements changes are included in software, **regression testing** shows, by repetition of tests, that the software behaviors are unchanged except for the required changes. Although a testing procedure is usually ad hoc and individualized by the software application and organization, most organizations conduct their testing approach in the order described in Figure. 2.1 when they test a new product or changes. In this thesis, we focus and improve component testing and integration testing.

In component testing and integration testing, all errors\(^2\) can be detected if all the possible

\(^1\)Software projects are organized as a set of work assignments. A **module** is a work assignment [31].

\(^2\)An **error** is a design flaw or deviation from a desired or intended state [19]. A **bug (fault)** is an incorrect
Figure 2.1  Test Procedure

combination of input values are tested; however, testing all is usually impossible because the combinatorial number of input values is too high. For example, an input value in an integer variable domain has $2^b$ values where $b$ is the number of bits for an integer value. If we have $n$ integer variables, then the combinatorial number of input values is $2^b \times 2^b \times ... = 2^{nb}$. Thus, the number of the combinations of input values is exponential on the number of the bits of input variables and the number of input variables. Hence, we have to select only a partial set of input value combinations in practice.

A test case is a pair of input history and output values under a program execution environment. Test cases are selected using two approaches: black-box testing (functional testing) and white-box testing (structural testing) [5, 11]. Black-box testing is an approach to develop test cases for each equivalence class of input variables. The equivalence classes are documented in software specifications. White-box testing is an approach to develop test cases for each interesting code structure, such as conditional branches in source code. The limitation of white-box testing is that one cannot determine whether all equivalence classes are tested, since the source of test cases is the source code, which says nothing about the equivalence class specified in a computer program [15]. A failure is an inability of a system or component to perform its required (intended) functions within specified performance requirements [19]. Errors lead to bugs, and bugs lead to failures.

3 Each input variable is a time function and has a history.

4 An equivalence relation is a relation that is reflexive, transitive, and symmetric. An equivalence class is a class for objects that satisfy an equivalence relation. If we partition the input value domain into equivalence classes (equivalence classes of input values), the testing result on one object in an equivalence class is the same as the testing result on any object in the equivalence class.
in the specification. The limitation of black-box testing is that even if all equivalence classes are tested, one cannot detect errors that are caused by incorrect coding for some input value combinations in an equivalence class. Accordingly, both test case development approaches should be used as long as possible; however, as the number of input and output variables, i.e. the size of component, grows, the total number of white-box and black-box test cases becomes, again, intractable. Therefore, many researchers have proposed approaches to selecting an important partial set of test cases. Among them, black-box testing especially delivers benefits to this problem solving, since it does not use the knowledge of source code, which is too detailed and large, but uses software specifications that abstract away details of source code.

As explained, black-box testing identifies equivalence classes and tests them. As in section 1.1, the first issue was identifying use cases, i.e. equivalence classes of input variables. Many researchers have proposed formal specification languages to specify equivalence classes of input variables exhaustively; thus, section 2.2 explains the formal specification technique. The second issue was prioritizing and selecting test cases. Statistical software testing is a testing method that performs the test case prioritization with usage profiles. Profiling is a program analysis technique that measures characteristics such as function call frequencies of usage and memory usage and produces a usage profile that shows the statistics. Section 2.3 summarizes this statistical software testing technique.

\footnote{Note the relation between a use case and an equivalence class of input variables is one-to-one mapping.}
2.2 Formal Specification Technique

A **description** of a system is written or spoken accurate information about the system [32]. A **document** of a system is a written description, and should always have official status. A **specification** of a system is a description that specifies the required and intended (designed) properties of the system. As software and development organizations become larger, documenting specifications takes on a new significance: specification documents help a team of engineers 1) design; 2) review the design; 3) maintain the software with low cost; 4) make responsibilities clear; 5) make software easy to inspect, hence, easy to change and reuse; 6) reduce design flaws from incomplete design; 7) derive black-box test cases. Although the set of documents differs by development processes or development team organizations, development projects produce the following specification documents: 1) system requirements; 2) module structures; 3) module interfaces; 4) module internal design. As for the component testing and integration testing use, module structure and module interface specification documents give testers the correct (required and intended) behaviors of modules.

Although several definitions of the term, **module** exist, we use Parnas’ definition. Software projects are organized as a set of work assignments, and a **module** is a collection of programs to be implemented as a work assignment [31, 28]. Similar but different terms are **component** and **class**. In this thesis, we make a clear distinction between **component**, **class** and **module**. A **component** is a collection of programs distributed as a unit and used in several systems without modification [33]. A **class** in Object-oriented programming is a software element describing an abstract data type [22] and its partial or total implementation [22]. In other words, a module is a unit for work decomposition, a component is a unit for program deployments, and a class in Object-oriented programming is a unit for design or implementation. A module may include several components or classes. A component may include several modules or classes. A class may be used in several classes or components.

Modules are designed using the information hiding principle [31]. Information hiding [22] is a mechanism to bundle data with functions operating on that data. On the other hand, **information hiding** is a design effort to enable independent development and independent changes.

---

6. "An abstract data type is a set of objects defined by the list of operations, or features, applicable to these objects, and the properties of these operations." [22]

7. Encapsulation is a mechanism to bundle data with functions operating on that data. On the other hand, **information hiding** is a design effort to enable independent development and independent changes.
design principle in which each module is designed to hide its design decisions that are difficult or likely to change in future, so that one can change those decisions later, separately from the rest of modules. The design decisions may be concerned with data structures, procedures, hard-coded parameters, algorithms, etc.

A module is specified with a module interface specification, which may use a formal specification language. The module interface specifies input-output relations of the module. Many approaches for writing module interface specification have been proposed [40]. These approaches can be generally categorized by the orientation of their approach.

- **State-oriented:**
  Z [2] or VDM [1] is a language that describes system with input values, output values, state variables, and actions that changes the state variable’s value.

- **Process-oriented:**
  LOTOS, Estelle, or SDL [17] is a language that describes system as a set of processes.

- **Operation-oriented:**
  Algebraic representations, such as Type Algebra Theory, Larch/CLU, and Initial Algebra Theory, are languages that describe the semantics of operations with a set of axioms [40].

- **Behavior-oriented:**
  Trace representations, such as Trace Assertion Method (TAM) and Trace Function Method (TFM) [33, 3, 37, 29], are languages that describe outputs. TAM describes a set of axioms about traces, which is a history of inputs and outputs, and the final output. TFM describes outputs as functions on traces.

Whereas all representations except trace representations require the identification of inner-module data, trace representations require the specification of only externally observable module inputs and outputs. Thus, trace representations conform to the information hiding principle, which hides the inner design of modules, and are suitable for the module interface specification. In this thesis, we use trace representations.
**Trace Assertion Method (TAM)** identifies canonical traces and equivalence of traces [4]. TAM is not intuitive for engineers in practice because it is axiomatic. Parnas and Dragomiroiu proposed the **Trace Function Method (TFM)** [33], which specifies only output values with predicates on traces. Since TFM does not have canonical traces and equivalences of traces, it is a direct representation to the module input-output relations, and TFM has proven its usefulness in limited industrial cases [29, 30, 3, 37]. In this thesis, we utilize this TFM.

Then, how can we write and read TFM module interface specification? In trace representations, a software module is viewed as a state machine where state transitions occur at discrete points of time. TFM introduces the following terms [33]:

- **An event** is a transition trigger of the state machine. At an event, a software module performs the combination of the following:
  - Reading all of the input variable values;\(^8\)
  - Changing the inner state of the state machine;
  - Updating some of the output variable values.\(^9\)

- **An abbreviated event descriptor** \(^10\) is a set of tuples:
  - Input variable name and value before the event;\(^11\).
  - Output variables name and value after the event.

In this thesis, we call an abbreviated event descriptor an **event descriptor** for simplicity. A possible syntax of event descriptors is described in BNF as follows:

---

\(^8\)These variables include global variables and input variables via access program arguments. Each variable is considered as a time function and has a value at an event.

\(^9\)These variables include global variables and output variables via return values of access programs.

\(^10\)A full event descriptor is values of all variables before and after the event. Variable values unchanged by events are also included in full event descriptors.

\(^11\)Access program names are considered as input variables. The value of the variable is the name of the access program.
**event_descriptor** ::= (PGM : func, parameter_list)

*func* ::= Name of an access program

*parameter_list* ::= parameter, parameter_list | parameter

*parameter* ::= ′input_variable < type >: value | output_variable′ < type >: value

*input_variable* ::= Name of an input variable

*output_variable* ::= Name of an output variable

*value* ::= constant

The letter “.” means an empty string. “type” includes all data types and abstract data types that typed programming languages such as C, C++, and Java accept. With this syntax, an example event descriptor that has a function name *myfunction*, an input variable *myin*, a global variable<sup>12</sup> *myglobal*, and an output variable *myout* is as follows:

(PGM:myfunction, ′myin<int>:100, ′myglobal<int>:10, myout'<int>:30, myglobal'<int>:10).

- **A trace** is a finite sequence of event descriptors. Possible representation of the trace with abbreviated event descriptors is as follows. The period concatenates two event descriptors.

(PGM:myfunction1, ′myin<int>:100, ′myglobal<int>:10, myout'<int>:30, myglobal'<int>:10).(PGM:myfunction2, ′myglobal<int>:10, myglobal'<int>:100)

This trace is equivalent to the following execution steps:

**Step1.** *myfunction1* is called with *myin*=100, *myglobal*=10 and returned *myout*=30, *myglobal*=10.

**Step2.** *myfunction2* is called with *myglobal*=10 and returned *myglobal*=100.

- **A TFM module interface specification** consists of the following:

  - A complete description<sup>13</sup> of inputs and their data type (section 1 in Figure. 2.4);

<sup>12</sup>In this thesis, a **global variable** is a variable that is shared among modules. Note that a global variable can be an input variable or an output variable

<sup>13</sup>A description is complete when all required description for the document are described[33].
– A complete description of outputs and their data type (section 2 in Figure. 2.4);
– A complete description of access programs and event descriptors (section 3 in Figure. 2.4);
– A set of relations of the value of each output variable and the history of events. These relations are described with predicates in a tabular form (section 4 in Figure. 2.4);
– Auxiliary functions\(^{14}\) that are any functions to simplify the predicates, if desired (section 5 in Figure. 2.4);
– We may also associate the document with supplemental descriptions, such as design issues, document review questions, test cases, and others, for software development processes.

The output behaviors definition (section 4 in Figure. 2.4) specifies a “set of relations of the value of each output variable and the history of events.” In this definition, the output values are specified by conditioning the event history with predicates on traces. In this thesis, we refer to the predicates on traces as trace predicates. Trace predicates and output value definitions may use the following denotations, which are summarized in Figure 2.2 in detail.\(^{15}\)

1) **Primitive functions on event descriptors** that refer to data in an event descriptor;
2) Notations for an empty trace and the trace concatenation;
3) **Basic functions on traces** that traverse traces and return some data;
4) **Useful function generators on predicates and traces** that traverse traces and return some data that satisfies predicates.

The purpose of the basic functions and useful function generators are to make the predicate description compact. In addition to these functions or function generators, specification writers may add more auxiliary functions (section 5 in Figure. 2.4) to make descriptions as compact and readable as possible.

\(^{14}\)An example auxiliary function is \(\text{inIntegerRange}(x) \equiv (\text{Lower bound value of int}) \leq x \leq (\text{Upper bound value of int})\). This function helps simplifying predicates that uses the range.

\(^{15}\)More denotations are summarized in Appendix A
### Primitive functions on event descriptors

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM(e)</td>
<td>PGM:(event descriptor) → (string)</td>
<td>A function that returns the name of the access program in the event descriptor e</td>
<td>If e = (PGM:myfunc,&quot;in&lt;int&gt;:100, out&lt;int&gt;:11&quot;, then PGM(e) = myfunc</td>
</tr>
<tr>
<td>e'(e)</td>
<td>e':(event descriptor) → (data type of e)</td>
<td>A function that returns the value of e before the event of the event descriptor e</td>
<td>If e = (PGM:myfunc,&quot;in&lt;int&gt;:100, out&lt;int&gt;:11&quot;, then e' = 100</td>
</tr>
<tr>
<td>e'(e)</td>
<td>e':(event descriptor) → (data type of e)</td>
<td>A function that returns the value of e after the event of the event descriptor e</td>
<td>If e = (PGM:myfunc,&quot;in&lt;int&gt;:100, out&lt;int&gt;:11&quot;, then out'(e) = 1</td>
</tr>
</tbody>
</table>

### Notation for traces

<table>
<thead>
<tr>
<th>Notation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 . T2</td>
<td>A period ';' concatenates two traces.</td>
</tr>
</tbody>
</table>

### Basic functions on traces

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(n,T)</td>
<td>r:(integer) × (trace) → (event descriptor)</td>
<td>A function that returns the n-th most recent event descriptor in the trace T.</td>
<td>If T = E1 . E2 . E3 where En is an event descriptor, then r(1,T) = E3, r(2,T) = E2, r(3,T) = E1, r(4,T) = undefined.</td>
</tr>
<tr>
<td>oldest(T)</td>
<td>oldest:(trace) → (event descriptor)</td>
<td>A function that returns the first (oldest) event descriptor in the trace T.</td>
<td>If T = E1 . E2 . E3 where En is an event descriptor, then oldest(T) = E1.</td>
</tr>
<tr>
<td>p(n,T)</td>
<td>p:(integer) × (trace) → (trace)</td>
<td>A function that returns the prefix trace of r(n,T) in the trace T. p(n_) = Undefined.</td>
<td>If T = E1 . E2 . E3 where En is an event descriptor, then p(1,T) = E1 . E2, p(2,T) = E1, p(3,T) = E3.</td>
</tr>
<tr>
<td>subseq(T)</td>
<td>subseq:(trace) → (trace)</td>
<td>A function that returns the trace T' such that T = oldest(T). T'</td>
<td>If T = E1 . E2 . E3 where En is an event descriptor, then subseq(T) = E2 . E3.</td>
</tr>
</tbody>
</table>

### Useful function generators on traces and predicates

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>exist(P,T)</td>
<td>exist:(predicate) × (trace) → (boolean)</td>
<td>A function that returns true if and only if there exists an event descriptor in T that satisfies P.</td>
<td>If T = E1 . E2 . E3 where En is an event descriptor, and E1 satisfies P, then exist(P,T) = true.</td>
</tr>
<tr>
<td>idx_r_st(P,T)</td>
<td>idx_r_st:(predicate) × (trace) → (integer)</td>
<td>A function that returns the index of the most recent event descriptor in T that satisfies P. The index starts from 1.</td>
<td>If T = E1 . E2 . E3 where En is an event descriptor, and E1, E2, and E3 satisfy P, then idx_r_st(P,T) = 3.</td>
</tr>
<tr>
<td>idx_oldest_st(P,T)</td>
<td>idx_oldest_st:(predicate) × (trace) → (integer)</td>
<td>A function that returns the index of the first (oldest) event descriptor in T that satisfies P. The index starts from 1.</td>
<td>If T = E1 . E2 . E3 where En is an event descriptor, and E1, E2, and E3 satisfy P, then idx_oldest_st(P,T) = 1.</td>
</tr>
</tbody>
</table>

Figure 2.2 Functions and Notations for Predicates (See Additional Definitions in Appendix A)
Example:

Figure 2.4 describes a possible TFM module interface document for a stack module. The stack module is simplified for an explanation, so it does not consider the case when the stack is full.

- TFM specifies the value of each output variable and the return of auxiliary functions in closed-form solutions (equations) at the output behaviors definition (section 4 in Figure 2.4) and auxiliary functions definition (section 5 in Figure 2.4).

The output behavior definition is given as a set of functions whose domain is traces $T$ and whose range is output values. The auxiliary function definition is given as a function whose domain is variable values or traces and whose range is some values.

TFM may use a tabular format. The tabular format to define output behavior definition and auxiliary function definition has two parts, the trace predicates columns that is immediately right of the equivalence symbol ‘$\equiv$’, and the value column that is enclosed by a heavy line. The tabular format is interpreted as meaning that, an output variable value or an auxiliary function return value is equal to a value in the value column if the trace predicates in the corresponding row of the trace predicates columns are true. For example, the definition of value in Figure 2.4 is equivalent to the conditional expressions shown in Figure 2.3. The definition of pushes in Figure 2.4 is equivalent to the ones shown in Figure 2.5. Note that the output value column may have recursive definitions, such as $\text{pushes}(T_1, T_2) \equiv \text{pushes}(T_1, \text{subseq}(T_2))$, to define the output value by traversing traces.

- In this stack module, the most recent PUSH event is canceled by a POP event. In order to make the output behavior definition compact, we defined the auxiliary function pushOnlyTrace that strips all canceled PUSH events and POP events and returns a trace that has only PUSH events that have not been removed by POP event. pushOnlyTrace also strips failed PUSH events (a PUSH event fails if its input value is out of the range).
In this thesis, values in the event descriptor definitions are described as '⋆'. For example, \((\text{PGM: TOP}, \text{value} <\text{int} : \star)\). This '⋆' means that the value may be any value in its variable data type.

\[
\text{value}(T) \equiv \begin{cases} 
\text{top}(p(1,T)) & \text{if } \text{PGM}(r(1,T)) = \text{TOP} \\
\text{value}(p(1,T)) & \text{if } \text{PGM}(r(1,T)) = \text{PUSH} \\
\text{value}(p(1,T)) & \text{if } \text{PGM}(r(1,T)) = \text{POP} \\
\text{Undefined} & \text{if } T = \_ 
\end{cases}
\]

Figure 2.3  Closed-form Solution for value - Conditional Expressions Format
[Section 1] Input Variables Definition

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>int</td>
</tr>
</tbody>
</table>

[Section 2] Output Variables Definition

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>int</td>
</tr>
<tr>
<td>value</td>
<td>int</td>
</tr>
</tbody>
</table>

[Section 3] Access Programs and Event Descriptors Definition

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>in</td>
<td>top</td>
<td>(PGM:PUSH, 'in&lt;int&gt;:<em>,top&lt;int&gt;:</em>)</td>
</tr>
<tr>
<td>POP</td>
<td>_</td>
<td>top</td>
<td>(PGM:POP, top'&lt;int&gt;:*)</td>
</tr>
<tr>
<td>TOP</td>
<td>_</td>
<td>value</td>
<td>(PGM:TOP, value'&lt;int&gt;:*)</td>
</tr>
</tbody>
</table>

[Section 4] Output Behaviors Definition

\[
top(T) \equiv \begin{cases} 
\text{pushOnlyTrace}(T) = _ & \text{Undefined} \\
\text{pushOnlyTrace}(T) \neq _ & 'in(r(1, \text{pushOnlyTrace}(T))) 
\end{cases} \\
\]

\[
value(T) \equiv \begin{cases} 
\text{PGM}(r(1,T)) = \text{TOP} & \text{top}(p(1,T)) \\
\text{PGM}(r(1,T)) = \text{PUSH} & \text{value}(p(1,T)) \\
\text{PGM}(r(1,T)) = \text{POP} & \text{value}(p(1,T)) \\
T = _ & \text{Undefined} 
\end{cases} \\
\]

[Section 5] Auxiliary Functions Definition

\[
\text{pushes}(T1, T2) \equiv \begin{cases} 
T2 = _ & T1 \\
\neg(T2 = _) & \text{pushes}(T1, T2) \\
\neg(\text{noeffect}(\text{oldest}(T2))) & \text{pushes}(T1, \text{subseq}(T2)) \\
\neg(\text{PGM}(\text{oldest}(T2)) = \text{PUSH}) & \text{pushes}(T1, \text{oldest}(T2), \text{subseq}(T2)) \\
\text{PGM}(\text{oldest}(T2)) = \text{POP} & \text{pushes}(T1, \text{subseq}(T2)) \\
\neg(T1 = _) & \text{pushes}(p(1,T1), \text{subseq}(T2)) \\
\end{cases} \\
\]

\[
\text{noeffect}(e) \equiv \begin{cases} 
\text{PGM}(e) = \text{PUSH} & \land \neg\text{inrange('in(e))} \\
\text{PGM}(e) = \text{TOP} & \land \neg\text{inrange('in(e))} \\
\end{cases} \\
\]

\[
\text{inrange}(i) \equiv (\text{Lower Bound Value of int}) \leq i \leq (\text{Upper Bound Value of int}) \\
\]

Figure 2.4 TFM Module Interface Document for a Stack Module
\[
\text{pushes}(T_1, T_2) \equiv \begin{cases} 
T_1 & \text{if } T_2 = _- \\
\text{pushes}(T_1, \text{subseq}(T_2)) & \text{if } \neg (T_2 = _) \land \neg \text{noeffect(oldest}(T_2)) \\
\text{pushes}(T_1, \text{oldest}(T_2), \text{subseq}(T_2)) & \text{if } \neg (T_2 = _) \land \neg \text{noeffect(oldest}(T_2)) \land [\text{PGM(oldest}(T_2)) = \text{PUSH}] \\
\text{pushes}(\_, \text{subseq}(T_2)) & \text{if } \neg (T_2 = _) \land \neg \text{noeffect(oldest}(T_2)) \land \neg \text{noeffect(oldest}(T_2)) \land [\text{PGM(oldest}(T_2)) = \text{POP}] \\
\text{pushes}(\text{p}(1, T_1), \text{subseq}(T_2)) & \text{if } \neg (T_2 = _) \land \neg \text{noeffect(oldest}(T_2)) \land \neg \text{noeffect(oldest}(T_2)) \land \neg \text{noeffect(oldest}(T_2)) \land \neg (T_1 = _) \land [\text{PGM(oldest}(T_2)) = \text{POP}] \\
\text{pushes}(T_1, T_2) & \equiv \\
\end{cases}
\]
2.3 Statistical Software Testing Technique

Until the late 1970s, the goal of testing was to show that software worked. Myers[26] pointed out that one can corrupt the testing process with this goal as follows: the probability of showing that software does not work will increase as the software is tested, and hence, the software should be tested less. Another possible goal of testing was to find software errors (bugs). Since we do not know when in the future errors may occur, we do not know when to stop testing, and hence, cannot release products. In the 1980s, Beizer[5] claimed that the goal of software testing was to provide enough testing to ensure that the probability of failure can be accepted. He claimed that ”enough” comes from a judgment of software reliability with a statistical measure. This is accomplished by statistical software testing.

**Statistical software testing** or **statistical quality control** is a testing method that constructs a predetermined number of test cases that are relatively more important than others[23, 42, 24, 25]. The importance is usually decided by considerations of the occurrence frequency and criticality of the use case. Specifically, statistical software testing develops usage statistics as usage profiles, prioritizes each usage with the statistics, and lists prioritized usages to test. Consequently, the absence of errors in the list of usages provides a statistical rationale for the certification of software quality.

Musa proposed to create the usage statistics for each pair (user group, system mode) to reduce the effect of population size to the statistics [24, 25]. A **user group** is an independent group of users, such as administrators or customers. A **system mode** is an independent coarse-grained state of the system, such as an error recovery mode, a daily use mode, or the nuclear power plant shutdown mode [17].

How are those statistics derived? They can be developed in two ways: measurements and estimations. First, the criticality can be derived only from estimations. If the product is a safety critical system, the estimation may come from a hazard analysis [19] of the system. A

---

16 **Reliability** is the probability that a component will perform its intended function satisfactorily for a prescribed time and under predetermined environmental conditions [19].

17 **Criticality** is the parameter that shows the severity of effect when the software part fails.

18 A **usage** is a use case. The relation between a use case and an equivalence class of input variables is one-to-one mapping; thus, the relation between a usage and an equivalence class of input variables is also one-to-one.
usage may be assigned a high criticality factor if found to cause hazards during hazard analysis. Alternatively, we may consider that usages with frequent bug fixes or modifications usually have bugs\[27\] \[39\], and hence, their criticality is larger than others.

Secondly, we have two options for deriving the occurrence probability. The first option is counting the occurrences of each usage during program executions and calculating their occurrence probabilities. The occurrence counting may be done with execution logs that are saved during a usage monitoring test to test the user interface or user satisfaction of products, or family of products\[41\] in the market, if possible. The second option for deriving it is estimations. The estimation should be done by a group of people who know the product usage well and can estimate the adequate occurrence probability of each equivalence class.

Example:

Consider an online sales system as an example. Assume that we want to develop usage statistics for the system.

- The system usages were identified as follows: \{Search products, Purchase products, Register selling products, Website appearance configuration, Database operations\}.
- The system had three groups of users: \{buyers, sellers, system administrators\}, and the respective groups had different ways of using the system.
- Three system modes were identified as follows: \{customer mode, maintenance mode, survey mode\}.

The number of people in the system administrators group was smaller than others, and although the survey mode was rarely used, it was important; thus, to avoid false prioritizations, we develop statistics for each user group to remove the factor of population size from the statistics.

\[19\]Technically, the log embedding may be done with Aspect-Oriented Programming library such as AspectJ and AspectC++. If we want to insert logging code at compile-time, we may develop a compile-time logging code insertion program. If we want to embed logging code at run-time, run-time instrumentation mechanisms such as dynamic Aspect-Oriented Programming may be options.
Then, we measure/estimate and calculate each occurrence probability and criticality adequately, which is summarized as Table 2.1. This table is the usage statistics of the system.

<table>
<thead>
<tr>
<th>User Group</th>
<th>System Mode</th>
<th>Usage</th>
<th>Occurrence Probability (%)</th>
<th>Criticality (1..100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buyer</td>
<td>Customer Mode</td>
<td>Search products</td>
<td>80</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Purchase products</td>
<td>20</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register selling products</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Website appearance configuation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Database operations</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Seller</td>
<td>Customer Mode</td>
<td>Search products</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Purchase products</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register selling products</td>
<td>80</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Website appearance configuation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Database operations</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Administrator</td>
<td>Maintenance Mode</td>
<td>Search products</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Purchase products</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register selling products</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Website appearance configuation</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Database operations</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>Survey</td>
<td>Mode</td>
<td>Search products</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Purchase products</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register selling products</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Website appearance configuation</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Database operations</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>
Then, the final issue is how to develop test cases using the usage statistics. In statistical software testing, a testing priority of each usage is calculated with the statistics, and is used to prioritize the usage. Algorithm 1 is an example algorithm for developing a prioritized list of usages, and hence, test cases.\textsuperscript{20} An example output for \((\text{user group, system mode}) = (\text{Administrator, Maintenance Mode})\) in Table 2.1 is shown in Table 2.2. A larger priority has a higher priority.

\begin{algorithm}
\caption{Derive a prioritized list of usages (test cases) with usage statistics}
\textbf{Input:} Usage statistics:
\begin{align*}
Pr: & (\text{User group}) \times (\text{System mode}) \times (\text{Usage}) \rightarrow (\text{Occurrence probability}) \\
Cr: & (\text{User group}) \times (\text{System mode}) \times (\text{Usage}) \rightarrow (\text{Criticality})
\end{align*}
\textbf{Output:} A list of pairs \((\text{usage, priority})\) in the priority descending order:
\begin{align*}
L: & (\text{Usage}) \times (\text{Priority})
\end{align*}
\begin{algorithmic}[1]
\Procedure{Algorithm 1}{Derive a prioritized list of usages (test cases) with usage statistics}
\For{each pair \((g_i, m_j)\)}
\For{each usage \(u_k\) of \((g_i, m_j)\)}
\State Calculate the priority \(p_k\) of \(u_k\).
\State \(p_k = 100 \times \frac{Pr(g, m, u) \times Pr(g, m, u)}{\sum_{u_k \in (g_i, m_j)} Pr(g, m, u)} \times \frac{Cr(g, m, u) \times Cr(g, m, u)}{\sum_{u_k \in (g_i, m_j)} Cr(g, m, u)}\)
\State Insert \((u_k, p_k)\) to \(L\) between the elements \(l_1 = (u_1, p_1)\) and \(l_2 = (u_2, p_2)\) where \(p_1 \geq p_k \geq p_2\).
\EndFor
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

\begin{table}[h]
\centering
\caption{Example list of pairs (usage, priority)}
\begin{tabular}{|l|c|}
\hline
Usage & Priority (1..100) \\
\hline
Database operations & 15 \\
Website appearance configuration & 9 \\
Register selling products & 2 \\
Search products & 5 \\
Purchase products & 5 \\
\hline
\end{tabular}
\end{table}

\textsuperscript{20}This example algorithm multiplies the occurrence probability and criticality of a usage to calculate its priority; however, the calculation method is not limited to this in the statistical software testing. We may choose an appropriate calculation method for each application.
CHAPTER 3. APPROACH

This thesis presents a new black-box test case generation method for component testing and integration testing. Figure 3.1 outlines the work presented in this thesis, and Figure 3.2 describes the overall data flow of component testing and integration testing. As shown in Figure 3.1, first, we get TFM module interface documents, measure or estimate usage statistics of modules, and associate them with the TFM module interface documents. These documents work as usage statistics documents. Finally, using the usage statistics document and a predetermined test plan that is a tuple (number of test cases, user type, system mode), we generate a prioritized list of black-box test cases for component testing and integration testing.

Figure 3.1 Overview of Test Case Generation Process

1Note that Figure 3.2 omits some software design phases to give a rough view.
Figure 3.2 Overview of Testing Process
3.1 TFM Module Interface Document for Component Testing and Integration Testing

3.1.1 Assumptions - Module Interface Specification

**Definition:** A trace specification is an ordered pair (syntax specification, semantic specification), where a syntax specification is a decidable set\(^2\) of syntax sentences, and a semantic specification is a decidable set of assertions \(^3\).

In TFM, a conditional \{\langle \text{trace predicates} \rangle \rightarrow \langle \text{output value definition} \rangle \} in the output behaviors definition, i.e. an equation and its condition in the conditional expressions (Figure 2.3), is an assertion. Thus, the output behaviors definition has a decidable set of assertions and is the semantic specification.

**Definition:** A trace specification with its assertions \(\Gamma\) is **consistent** if there is no formula \(\theta\) such that \(\Gamma \vdash \theta\) and \(\Gamma \vdash \neg \theta\) \(^4\).

**Definition:** A trace specification with its assertions \(\Gamma\) is **total** if for any given trace constant \(T\), every output variables \(V\) has a constant \(a\), such that \(\Gamma \vdash (V(T) \equiv a)\), or \(V\) is defined as \((V(T) \equiv \text{Undefined})\). \(^5\)

The rest of this thesis assumes that every module has consistent and total TFM specifications, which is written at the module interface design phase. \(^6\)

---

\(^2\) A **decidable set** is a set that has an algorithm that halts in a finite time to decide whether or not a given object is in the set. The decidable set is also called computable or recursive.

\(^3\) An **assertion** is a set of conditions that program states or variables must satisfy \(^1\5\).

\(^4\) An argument \(<\Gamma, \phi>\) where \(\Gamma\) is premises, and \(\phi\) is a conclusion is **derivable** if there is a deduction from some of its premises to its conclusion, which we denote as \(\Gamma \vdash \phi\). \(^3\8\)

\(^5\) Note a specification allows specifying partial functions.

\(^6\) The methodology to write or check TFM specifications is outside the scope of this project.
3.1.2 Module Interaction Trace Specification

We introduce an approach to write TFM module interface specification with module interactions. Conventional TFM module interface specifications have some syntax and semantics that complicate the specification of module interactions\(^7\); thus, we relax them in section 3.1.2.1. Additionally, traces in TFM do not consider nesting events that appear in most test cases in integration testing. Therefore, we introduce new primitive functions in section 3.1.2.2.

3.1.2.1 Relaxation of Multiple Object Trace Representation

For component testing and integration testing, we want a single testing trace that has the information of the occurrence order of events on multiple objects; hence, we integrate traces of all objects and write it as \( T \), and give an object name to any operation on the object via an input variable of the corresponding event.

**Example:**

An example is shown in the following trace. Two objects of Module A are instantiated in the first two events, and the rest of the events get their object value via each input variable.

\[
T = (\text{PGM: ModuleA\_constructor, object' < ModuleA > : objectA1}).
\]
\[
(PGM: ModuleA\_constructor, object' < ModuleA > : objectA2)).
\]
\[
(PGM: ModuleA\_printHelloWorld, object < ModuleA > : objectA1).
\]
\[
(PGM: ModuleA\_printHelloWorld, object < ModuleA > : objectA2).
\]
\[
(PGM: ModuleA\_destructor, object < ModuleA > : objectA1).
\]
\[
(PGM: ModuleA\_destructor, object < ModuleA > : objectA2)
\]

\(^7\)When we say a trace constant \( T \) in conventional TFM module interface specifications, \( T \) is the trace of a primary object that we specify in the document. If the module interface contains interactions of two or more objects, the conventional TFM document separates each object’s trace with a notation \( T\cdot<\text{objectname}> \), where \(<\text{objectname}>\) is the name of the object and \( T\cdot<\text{self}> \) is the trace of the primary object[33].
3.1.2.2 Nesting Event Trace

A nesting event is a sub event that is invoked by other events. We cannot represent nesting events with the current set of the primitive trace functions (which we reviewed in Chapter 2), because an event descriptor always requires the values of output variables, which we cannot derive until after all nested events are evaluated. Therefore, we separate an event that invokes other events into two. We adopt “InvokePGM” and “ReturnPGM” variable in addition to “PGM” variable and primitive functions, InvokePGM(e), ReturnPGM(e), and modified PGM(e) defined in Figure 3.3. Then, we specify all input variables in the event descriptor of InvokePGM and all output variables in the event descriptor of ReturnPGM.

Example:

The example is shown in the following trace. This example describes the procedure that ModuleA_main program calls the access programs of Module B, ModuleB_func1 and ModuleB_func2, and returns “True” via an output variable, “result’.”

\[
T = (\text{InvokePGM: ModuleA_main,} \text{'in} < \text{Boolean} > : \text{True})
\]

\[
(PGM: ModuleB_func1, \text{out}’ < \text{Boolean} > : \text{True}).
\]

\[
(PGM: ModuleB_func2, \text{out}’ < \text{int} > : \text{100}).
\]

\[
(\text{ReturnPGM: ModuleA_main, result}’ < \text{Boolean} > : \text{True})
\]

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvokePGM(e)</td>
<td>InvokePGM: (event</td>
<td>A function that returns the name of the event invoker access program at</td>
<td>If e=(InvokePGM:myfunc,'in'&lt;int&gt;:100), then InvokePGM(e)=myfunc</td>
</tr>
<tr>
<td></td>
<td>descriptor)→(string)</td>
<td>the event descriptor e</td>
<td></td>
</tr>
<tr>
<td>ReturnPGM(e)</td>
<td>ReturnPGM: (event</td>
<td>If e=(ReturnPGM:myfunc, out'&lt;int&gt;:1), then ReturnPGM(e)=myfunc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>descriptor)→(string)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PGM(e)</td>
<td>PGM: (event descriptor)→(string)</td>
<td>A function that returns the name of the access program in the event</td>
<td>If e=(PGM:myfunc,'n'&lt;int&gt;:\text{100}, out'&lt;int&gt;:1) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>descriptor e</td>
<td>e=(InvokePGM:myfunc,'n'&lt;int&gt;:\text{100}), then PGM(e)=myfunc</td>
</tr>
</tbody>
</table>
3.1.3 Input and Output Specification

A module interface has an upper-face and a lower-face[33]. The **upper-face** shows a service that the module provides. The **lower-face** shows interactions with other modules. Figure 3.4 describes a module interface that has only an upper-face. Module A receives the value of inputs and returns the value of output variables. Although Module A is designed to use Module B, the use of Module B is not the service of Module A; thus, the use of Module B is hidden at Module A’s interface.

![Figure 3.4 Upper-Face Only Module Interface](image)

The upper-face only module interface specification delivers many benefits to software design, coding, and testing[10, 20]; however, testers need the lower-face specification to answer the following questions that arise during integration testing[5]:

- **Does any improper call or return sequence exist?**

- **Does any inconsistent handling of data objects exist?**

The lower-face might be specified in other documents, such as module internal design documents. Users of Module A need information about the required resources such as libraries, back-end servers, and...
cation must be formally defined. Therefore, we specify module interactions formally in TFM module interface documents.

We specify the output variables of a module that are actually returned from the module, and specify the used program events in trace predicates. Figure 3.5 shows what to specify with specific variables. Note that a module interface no longer specifies all service’s output variables. Some of them are specified only in used modules. We call it output delegation. The benefits of the output delegation are as follows:

- It enables the separation of concerns at module interfaces since some output value specifications are delegated to the used modules that exclusively offer those output services.

- We still can derive all services of the module from its interface specification if we aggregate all outputs in its used modules and itself.

Note that trace predicates on input values can be mapped to Meyer’s precondition[21], and trace predicates on output values with invoked access programs and returned values from the invoked access programs can be mapped to postcondition.

Example:

An example TFM interface document is shown in Figure 3.6. In this example, Sequencer’s access program “run” provides a service to set a robot arm angle to a constant “100” degrees and return “True”. RobotArmDriver has “setAngleTo100” that controls the arm physically and sets its angle to “100”. Since “run” calls “setAngleTo100”, “result” has the trace predicate that includes “setAngleTo100.”
Figure 3.5 Module Interface Specification

Sequencer Module Interface Specification

<table>
<thead>
<tr>
<th>Access Program</th>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>void</td>
<td>result:Boolean</td>
<td>{PGM:run, result&lt;Boolean&gt;:*}</td>
<td></td>
</tr>
</tbody>
</table>

Output Behaviors Definition

\[
\text{result}(T) \equiv \begin{cases} 
\text{PGM}(r(1,T))=\text{setAngleTo100} & \text{PGM}(r(2,T))=\text{run} \quad \text{TRUE} \\
\text{PGM}(r(2,T)) \neq \text{run} & \text{result}(p(2,T,r(1,T))) \\
\text{PGM}(r(1,T)) \neq \text{setAngleTo100} & \text{result}(p(1,T)) \\
T=\text{Empty} & \text{Undefined} 
\end{cases}
\]

RobotArmDriver Module Interface Specification

<table>
<thead>
<tr>
<th>Access Program</th>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>setAngleTo100</td>
<td>void</td>
<td>angle:int</td>
<td>{PGM:setAngleTo100, angle&lt;in&gt;:*}</td>
<td></td>
</tr>
</tbody>
</table>

Output Behaviors Definition

\[
\text{angle}(T) \equiv \begin{cases} 
\text{PGM}(r(1,T))=\text{setAngleTo100} & 100 \\
\text{PGM}(r(1,T)) \neq \text{setAngleTo100} & \text{angle}(p(1,T)) \\
T=\text{Empty} & \text{Undefined} 
\end{cases}
\]

Figure 3.6 Output Delegation
3.2 Statistics Document Development

As we saw in Chapter 2, a black-box test case is developed for each equivalence class of input values. In trace representations, a trace includes all input values; thus, an equivalence class of input values has an one-to-one correspondence to an equivalence class of traces. How can we find equivalence classes of traces in a TFM document? TFM documents specify the value of each output variable in a closed-form solution. This closed-form solution has two types of right hand side: 1) an output value definition or 2) a recursive definition with trace transformation functions. For example, RobotArmDriver in Figure 3.6 has the following equations:

\[
\begin{align*}
\text{angle}(T) & \equiv 100 \quad \text{if } PGM(r(1, T)) = setAngleTo100 \\
\text{angle}(T) & \equiv \text{angle}(p(1, T)) \quad \text{if } PGM(r(1, T)) \neq setAngleTo100 \\
\text{angle}(T) & \equiv \text{Undefined} \quad \text{if } T = \text{Empty}
\end{align*}
\]

On the right hand side, the equation (3.1) and (3.3) has each output value definition, and the equation (3.2) has a recursive definition with a trace transformation function \( p(1, T) \). If the TFM document is total, the trace transformation function eventually transforms the trace \( T \) into the one that satisfies the predicates of an equation that has an output value definition on its right hand side, (3.1) or (3.3) in this example. Since an equivalence class of traces delivers the same output values, the equivalence class has an one-to-one correspondence to the equation with an output value definition.

Therefore, we associate usage statistics with each equivalence class of traces, which is an equation with the output value definition in TFM documents, to prioritize the test case for the equivalence class. Then, we develop the occurrence probability and criticality of each equation for each user type and system mode with the same approach as the one explained in Chapter 2, and use them to calculate a priority of each equivalence class of traces and generate test cases. For rest of this thesis, we make an assumption that reasonable occurrence probabilities and criticalities of each equivalence class of traces can be developed for each module.

---

10 An equivalence relation is a relation that is reflexive, transitive, and symmetric. An equivalence class is a set of objects that satisfies an equivalence relation. Any event history in an equivalence class derives the same output values.
Example: An example of the statistics annotated document is shown in Figure 3.7.

This is an example module that controls a heater switch. If the room temperature \( \text{temperature} \) is less than or equal to 50F, then the access program \( \text{func} \) turns on the heater switch, which is \( \text{heaterON} \equiv \text{true} \). This example says that the occurrence probability of the class that has \( \text{heaterON} \langle \text{boolean} \rangle : \text{true} \) is “60/100” and the one that has \( \text{heaterON} \langle \text{boolean} \rangle : \text{false} \) is “40/100.” The criticality of the class that has \( \text{heaterON} \langle \text{boolean} \rangle : \text{true} \) is “30/100” and the one that has \( \text{heaterON} \langle \text{boolean} \rangle : \text{false} \) is “70/100”. These values can be used to calculate a priority of each equivalence class of traces.

<table>
<thead>
<tr>
<th>Output Behaviors Definition</th>
<th>{User Type, System Mode}</th>
<th>Occurrence Probability</th>
<th>Criticality</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{heaterON}(T) \equiv )</td>
<td>{Regular User, Normal Use Mode}</td>
<td>60/100</td>
<td>30/100</td>
</tr>
<tr>
<td>( \text{PGM}[r(1,T)] = \text{func} )</td>
<td>( \text{temperature}[r(1,T)] \leq 50 )</td>
<td>TRUE</td>
<td>( 40/100 )</td>
</tr>
<tr>
<td>( \text{PGM}[r(1,T)] = \text{func} )</td>
<td>( \text{temperature}[r(1,T)] &gt; 50 )</td>
<td>FALSE</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 3.7 Output Behaviors Definition of Module B with Statistics Annotations
3.3 Testing Trace Generation Method

3.3.1 Trace Postconditions

To generate testing traces, we “symbolically” execute [18] TFM module interface specification using usage statistics, we construct testing traces, and we aggregate constraints of input values. Before we propose a test case generation algorithm, we introduce new concepts, trace postconditions and monitored variable constraints. A trace postcondition is a conditional whose antecedent is a trace constant with a last event that has unevaluated output values, and whose consequent is a trace constant all of whose output values are evaluated. This evaluated means that some value or meta-variable is assigned to all output variables of the event. A meta-variable is a variable that is replaced by a constant in the test case generation algorithm that will presented in section 3.3.3. An example trace postcondition is as follows:

\[ T.(\text{PGM:func,} \text{'}in < int > : a, \text{out'} < int > : \text{UNEVAL}) \rightarrow \]

\[ T.(\text{PGM:func,} \text{'}in < int > : a, \text{out'} < int > : a) \]

“a” is a meta-variable. This conditional means that if a trace constant ends with an event descriptor (PGM:func, ’in<int>:a, out'<int>:*)) whose out’ is “UNEVAL” (unevaluated), then, the trace constant after the evaluation of the last event will be the consequent of the conditional whose out’ has the meta-variable “a”.

Another case of the trace postcondition is when an event invokes other modules’ access programs. As section 3.1.2.2 explained, we evaluate such an event to a separated pair of InvokePGM and ReturnPGM with its invoked events in the middle. For example, if Module A’s access program funcA invokes funcB that is an access program of Module B, its trace postcondition is as follows:

\[ T.(\text{PGM:func, result'} < \text{Boolean} > : \text{UNEVAL}) \rightarrow \]

\[ T.(\text{InvokePGM:funcA}). \]

(PGM:funcB, UNEVAL).

(ReturnPGM:funcA, result' < Boolean > : True)
Here, “funcB” has the output “UNEVAL” that means any outputs can occur. This is because we delegate output tasks to “funcB” and the evaluation of Module A’s event does not clarify the outputs of “funcB” unless they are handled as the return values of funcB and inputs to Module A in the interface specification of Module A. The formal syntax of trace postconditions is as follows in BNF:

\[
\begin{align*}
\text{postcondition} & ::= T.trUneval \rightarrow T.trEval \\
\text{trUneval} & ::= eUneval \\
\text{trEval} & ::= eEval|eEval.trEval \\
\text{eUneval} & ::= (PGM : func,in,outUneval) \\
\text{eEval} & ::= (programVar : func,in,outEval) \\
\text{programVar} & ::= PGM|InvokePGM|ReturnPGM \\
\text{func} & ::= \text{Name of an access program} \\
\text{in} & ::= i,in|i. \\
\text{i} & ::= 'inVar < type >: val \\
\text{inVar} & ::= \text{Name of an input variable} \\
\text{outUneval} & ::= oUneval,outUneval|oUneval|_ \\
\text{outEval} & ::= oEval,outEval|oEval|_ \\
\text{oUneval} & ::= outVar' < type >: UNEVAL|UNEVAL \\
\text{oEval} & ::= outVar' < type >: val \\
\text{outVar} & ::= \text{Name of an output variable} \\
\text{val} & ::= \text{constant|meta variable}
\end{align*}
\]

3.3.2 Monitored Variable Constraints

Inputs can be categorized into the following two variables:\(^{11}\):

- A **monitored variable** that is a variable whose value is determined externally. Physical values, such as sensor data, are the values of the monitored variables.

\(^{11}\)These names correspond to the variable names defined in the four-variable model[34, 36].
• An input variable that is a variable whose value is derived from access programs of other modules.

Input variable values are determined by software itself, so they are derivable from its specification; however, monitored variable values are derivable as domains, not values. The domains are specified as constraints in TFM documents. We call the collection of these constraints as monitored variable constraints. An example is shown in Figure 3.8. Consider the input variable 'temperature is the monitored variable. The value of 'temperature is not specified for each equivalence class of traces, but its monitored variable constraints are specified as “'temperature ≤ 50” or “'temperature > 50.”

Output Behaviors Definition

\[
\text{heaterON}(T) \equiv \begin{cases} 
\text{PGM}(r(1,T))=\text{func} & \text{'temperature}(r(1,T)) \leq 50 \quad \text{TRUE} \\
& \text{'temperature}(r(1,T)) > 50 \quad \text{FALSE} \\
\text{PGM}(r(1,T)) \neq \text{func} & \text{heaterON}(p(1,T))
\end{cases}
\]

Figure 3.8 Monitored Variable Constraints

3.3.3 Test Case Generation Algorithm

Finally, we generate test cases with the following steps:

Step1. Derive trace postconditions for each equivalence class of traces from the TFM module interface specifications.

Derive monitored variable constraints for each equivalence class of traces from the TFM module interface specifications.

Step2. Initialize an output trace \(T_{test}\) with a given start event.

Step3. For the oldest unevaluated event in \(T_{test}\), query the TFM module interface specification and get equivalence classes of traces that have possibly-satisfiable trace predicates on a given trace constant. A possibly-satisfiable trace predicate is a predicate that can be satisfied in future event evaluations, i.e. value or meta-variable assignments to variables. (Stage 0, i or i+2 in appendix D)
Step 4. Choose one equivalence class of traces randomly using the usage statistics. Get the consequent of the trace postcondition that is associated with the chosen equivalence class of traces, and insert the consequence of the trace postcondition into $T_{test}$. Get the monitored variable constraints that are associated with the chosen equivalence class of traces, and add the constraints to the monitored variable constraints collection. (Stage 0, i+1 or i+2 in appendix D)

Step 5. Repeat Step 3-4 until all events in $T_{test}$ are evaluated or the length of $T_{test}$ reaches the predetermined length in a test plan.

Step 6. Generate monitored variable values using monitored variable constraints and a certain probability distribution. (Stage i+3 in appendix D)\(^\text{12}\)

Note that the query in step 3 may return several possibly-satisfiable predicates, since an event is nondeterministic until all its nesting events are evaluated and all monitored variables are determined. The detailed pseudo-code is described in algorithm 2, 3, 4. An example testing trace generation will be described in section 4.1.

---

### Algorithm 2 Testing Trace Generation

1. **Input** Name of a program to invoke: $f$ and Name of its module: $m$
2. **Input** Usage statistics specifications for a particular user type and system mode: $S$ and Trace max length: $l$
3. Initialize a trace $T_{test}$ with $\{(PGM : f, *)\}$
4. Initialize a list of monitored variable constraint expressions $CONS$ with $Empty$
5. Initialize a key-value map of meta variable and value $VARS$ with $Empty$
6. CALL EventEvaluation (See Algorithm 3) with $T_{test}$, srcIndex = 0, $S$, $l$, $CONS$, $VARS$ and UPDATE $T_{test}$, $CONS$, $VARS$
7. Generate the value of monitored variables to satisfy $CONS$ and set them in $VARS$
8. Replace meta variables in $T_{test}$ with the corresponding values in $VARS$
9. **Output** $T_{test}$ and return

---

\(^{12}\)The probability distribution may be any distribution that has some rational. It may be a uniform distribution if we want to uniformly test input values in the domain.
Algorithm 3 EventEvaluation

1: **Input** A trace: $T$ and an event index of $T$ to evaluate: $srcIndex$
2: **Input** Usage statistics specifications: $S$ and Trace max lenth: $l$
3: **Input** A list of monitored variable constraint expressions: $CONS$
4: **Input** A key-value map of meta variable and value: $VARS$

/*Part1. Append events produced by the event at $srcIndex$ in $T$/
5: **CALL** QueryPossiblySatisfiableEquivalenceClasses with the sub-list of $T$ between 0 and $srcIndex$, function name at $srcIndex$ in $T$, module name at $srcIndex$ in $T$, $CONS$, and $VARS$ and

**OBTAIN** all candidate equivalence classes (i.e. equations with an output value definition) that the sub-list of $T$ between 0 and $srcIndex$ will be classified with any future events
6: Randomly select an equivalence class $ec$ from the all candidate equivalence classes using each occurrence probability and criticality
7: Overwrite or insert the events in the trace postcondition consequence of $ec$ to $T$. The events whose index exceeds the trace length upper bound $l$ are truncated.
8: Add the monitored variable constraints of $ec$ to $CONS$
9: if the index exceeds the upper bound length $srcIndex > l$ then
10:  **Output** $T, CONS, VARS$ and return
11: end if

/*Part2. Append events produced by all nested events*/
12: Increment $srcNextIndex = srcIndex + 1$
13: Initialize $lastIndex = srcIndex$
14: for all of the inserted nested events do
15:  **CALL** EventEvaluation with $T_{test}, lastIndex, S, l, CONS, VARS$ and
16:  **UPDATE** $T, lastIndex, CONS, VARS$
17:  **CALL** QueryPossiblySatisfiableEquivalenceClasses (See Algorithm 4) with the sub-list of $T$ between 0 and $lastIndex$, function name of the event at $srcIndex$ in $T$, module name of the event at $srcIndex$ in $T$, $CONS$, and $VARS$ and
18:  **OBTAIN** all candidate equivalence classes that the sub-list of $T$ between 0 and $lastIndex$ will be classified with any future events
19:  Randomly select an equivalence class $ec$ from the all candidate equivalence classes using each occurrence probability and criticality
20:  Increment $srcNextIndex = srcNextIndex + 1$
21:  Overwrite the sub-list of the produced trace in $ec$'s trace postcondition consequence between $srcNextIndex$ and the last element index onto $T$ from the position of $lastIndex$.
22: end for
23: if ReturnPGM(event at $lastIndex$ in $T \neq null$ then
24:  Increment $lastIndex = lastIndex + 1$
25: end if
26: **Output** $T, lastIndex, CONS, VARS$ and return
Algorithm 4 QueryPossiblySatisfiableEquivalenceClasses

1: **Input** A trace: $T$
2: **Input** Name of a program to invoke: $f$ and Name of its module: $m$
3: **Input** Usage statistics specifications: $S$
4: **Input** A list of monitored variable constraint expressions: $CONS$
5: **Input** A key-value map of meta variable and value: $VARS$
6: In $m$’s module interface specification in $S$, find and get equations (pairs of an output value definition and its trace predicates), $eq$, that have the trace postcondition for $f$
7: for each equation $eq$ do
8: Initialize $T_{temp} = T$
9: **REEXAMINE** TRANSFORMED:
10: Extract the preconditions of the last event in $T_{temp}$ from $eq$’s predicate
11: Check whether $T$ with $CONS$ and $VARS$ satisfies the preconditions with a predicate solver
12: if satisfied then
13: if the output value of $eq$ is a trace transformation function then
14: Transform $T_{temp}$ with the trace transformation function in the output value definition of $eq$
15: **GO TO** REEXAMINE.TRANSFORMED to reexamine the transformed trace $T_{temp}$.
16: else
17: Add $eq$ to the set of equations $SATEQ$
18: end if
19: end if
20: end for
21: **Output** the equivalence classes that correspond to $SATEQ$ and return
CHAPTER 4. EVALUATION

4.1 Feasibility of Test Case Generation - Example Application: Floating Weather Station

This section describes an example application to show that the proposed test case generation is feasible. Floating weather stations (FWS) are buoys that are deployed at sea and monitor the wind speed and periodically report the data via radio. The family of FWS is developed with the FAST (Family-Oriented Abstraction, Specification, and Translation) process [41]. As example software, we derived a member of FWS family that has a whale sensor. The whale sensor monitors whales with sonars, and its sensor driver program returns true if it detects whales.

The software system consists of 7 modules. The size of the program coded in Java is 10 classes, 38 methods, and 400 LOC. We ran the program with a FWS simulator program (Figure 4.1) written in Java (this work used the simulator code in [41]).

The module structure and representative process structure is presented in the module guide, the module hierarchical structure, and the uses relation in Appendix B. Although the original FWS family member has multiple threads, this example system is single-threaded for simplicity.

The usage statistics, trace postconditions, and monitored variable constraints are presented in Appendix C. The pair of expected user type and system mode is limited to one pair for simplicity. We measured occurrence probabilities with a simulator execution recording that prints out and records event descriptors as a log using Java reflection, JavaVM stack tracing, and hard-coding to get variable names and values. We set all criticality values at the same value for simplicity; thus, we used only occurrence probabilities for prioritizing. Trace postconditions and monitored variable constraints are derived as presented in section 3.3.3. In this example, we
derived trace postconditions and monitored variable constraints from trace predicates manually, but we could automate this derivation process since all trace postconditions are defined formally in trace predicates.

Consider that we test an access program $FWS_{\text{init}}$ with a maximum testing trace length 37. The service of $FWS_{\text{init}}$ is periodically receiving the sensor value five times and transmitting the averaged sensor value. We performed the test case generation algorithm and derived one test case shown in Figure 4.2. Detailed intermediate states in the test case generation algorithm are described step-by-step in appendix D. The nesting events are indented. The monitored variable values that testers have to create are enclosed by a thin line, and the service output is enclosed by a heavy line.

Testers can 1) determine how they should input monitored variable values; 2) check whether the program returned the service output values as indicated in the testing trace (in Figure 4.2, the output of $\text{TransmitDriver.sendWindSpeed}$ is the service output.); 3) check whether their testing program produced the sequence of events as indicated in the testing trace. Checking the service output values corresponds to component testing, and checking the sequence of events corresponds to integration testing; thus, testers can perform component testing and integration testing with the generated testing trace.
Test =
(Invoke PGM: FWS < init >, 'quit< Boolean >: FALSE).

(PGM: DataBanker_ init).
(PGM: MessageGenerator_ < init >, object' < MessageGenerator> : ob1)
(PGM: SensorMonitor_ < init >, object' < SensorMonitor> : ob2)

(PGM: ThreadSleep, 'sleepTime< int> : 1000).

(PGM: SensorDriver_ get, 'windSpeed< int> : 45, 'whalePassing< Boolean >: FALSE, sensorData'< SensorReading> : dd1).
(PGM: DataBanker_ write, 'sensorDataIn'< SensorReading> : dd1).
(Invoke PGM: SensorMonitor_ run)

(PGM: ThreadSleep, 'sleepTime< int> : 1000).

(PGM: DataBanker_ write, 'sensorDataIn'< SensorReading> : dd2).
(Invoke PGM: SensorMonitor_ run)

(PGM: ThreadSleep, 'sleepTime< int> : 1000).

(PGM: DataBanker_ write, 'sensorDataIn'< SensorReading> : dd3).
(Invoke PGM: SensorMonitor_ run)

(PGM: ThreadSleep, 'sleepTime< int> : 1000).

(PGM: DataBanker_ write, 'sensorDataIn'< SensorReading> : dd4).
(Invoke PGM: SensorMonitor_ run)

(PGM: ThreadSleep, 'sleepTime< int> : 1000).

(PGM: DataBanker_ write, 'sensorDataIn'< SensorReading> : dd5).
(Invoke PGM: SensorMonitor_ run)

(PGM: ThreadSleep, 'sleepTime< int> : 1000).

(PGM: DataBanker_ read, readData'< List< SensorReading> >: new List(dd1, dd2, dd3, dd4, dd5))
(Return PGM: getWindSpeed, averageData'< int>: cc1)

(PGM: TransmitDriver_ sendWindSpeed.

' msg< MessageFormatWind>: mm1 = new MessageFormat(cc1), msg'< MessageFormatWind>: mm1).
(Invoke PGM: Average_getWhale).

(PGM: DataBanker_ read, readData'< List< SensorReading> >: new List(dd1, dd2, dd3, dd4, dd5))
(Return PGM: Average_getWhale, whalePassing' = FALSE)

Figure 4.2 Generated Testing Trace (Test Case)
4.2 Computation Termination and Efficiency of Test Case Generation

Algorithm

Algorithm 2 does not halt if there exists a recursive output value definition that does not have a solution. In this case, algorithm 4 (line 14) will evaluate recursion infinitely many times and does not terminate. However, such a recursive output value definition does not exist if the TFM specification is total (as explained in section 3.1.1). Except for such a case, algorithm 2 halts either when the number of event descriptors in a generated trace reaches a threshold, or when an initial access program that we input to the algorithm returns its outputs.

Algorithm 2 (line 7) is NP-complete if a set of monitored variable constraints CONS makes a satisfying assignment as hard as 3-SAT or other NP-complete problems. If monitored variable value assignments are not as hard as NP-complete problems, the efficiency of algorithm 2 is calculated as follows and is polynomial time:

Consider $l_{max}$ is the maximum length of testing trace that is predetermined in a test plan, $b_{max}$ is the maximum number of equations in the simultaneous equations of the output behavior definition, $s_{max}$ is the maximum time for solving a trace predicate on a trace constant, $|V|$ is the number of all monitored variables.

Since the length of the output testing trace has an upper-bound $l_{max}$, the running time of algorithm 4 is $Q = O(l_{max}s_{max}b_{max})$, the running time of algorithm 3 is $E = O(l_{max}^2s_{max}b_{max})$, and the running time of algorithm 2 is $O(E+|V|) = O(l_{max}^2s_{max}b_{max} + |V|)$, which is polynomial time.
CHAPTER 5. CONCLUSION AND SUMMARY

5.1 Limitations and Improvement Idea

To adopt the presented approach, we needed to 1) identify modules and 2) prepare module interface specification documents in TFM. Identifying modules is difficult if performed on some ongoing software development projects with unidentified modules, but a large amount of code. In this case, we might have to refactor the software to identify them. Preparing TFM module interface documents is also difficult without strong tool support such as a trace predicate consistency checker, totalness checker, and editor tool with auto completions. We should develop these tools in future.

Second, we needed to measure or estimate usage statistics. The estimation is difficult if the module interface is complicated. If we visualize the behavior in other representations that are better at the graphical visualization, such as sequence diagrams and behavior animations, the prediction will be easier. The measurement is difficult if we do not want to sacrifice the program execution speed for recording the execution log. We may activate the logging only at the recording phase and deactivate it when we release the product; however, some behaviors may change if the execution timing changes because of logging. Thus, we need a practical solution for those behavior recordings in future.

Third, we generated test cases in traces. Testers need to follow the event sequence in traces and execute the program. In order to facilitate the testing process, we should develop a harness program that drives the target programs along a given testing trace, much as Hoffman proposed a harness program for a single module testing trace [12].

Fourth, the proposed test case generation algorithm aggregates monitored variable constraints and finds a monitored variable value assignment that satisfies the set of constraints;
thus, the test case generation algorithm can be NP-complete (as discussed in section 4.2). If possible, such a set of monitored variable constraints should be avoided when we write specifications. TFM module interface specification editor tools should have a constraints difficulty check capability and warn users of the difficulties of satisfying assignments.

Finally, we have considered only single-threaded programs, not multi-threaded programs. If we accommodate multi-threaded programs, we might need to introduce additional functions to be used in predicates such as a function that returns thread ID. Furthermore, we might need to simulate the thread dispatching mechanism in the test case generator. These are all considerations for future work.

### 5.2 Related Work

We proposed a method to specify module interactions with TFM. Module interactions are usually specified with UML sequence diagrams. Since a UML sequence diagram is a graphical representation, especially if the interaction has conditional branches, the diagram becomes large and its layout poor. As a practical alternative, our proposed method not only expresses interactions with text of trace predicates in tabular format, which is compact, but is relatively easy to exhaustively specify.

Many CASE tools incorporate test case generation into a state machine based specification framework [14, 8, 7]. In particular, MaTeLo has a feature to input usage profiles and generate test cases. Its approach is similar to Whittaker’s approach [42, 9]. However, since they are based on Markov chains, we need to identify the states of modules that are not externally visible and do not conform to the information hiding principle. In our approach, we identify only externally visible events to the module, which conforms to the information hiding and is beneficial, as explained in Section 3.1.3.

Our oracle\(^1\) is based on the satisfaction of the trace predicate. Hoffman’s approach [12] generated test cases for single modules with module interface specifications as oracle. Peters’ approach [35] generated oracles for single modules with LD-relation specification. However,\(^1\)

\(^1\)Oracle is any means that provides information about the (correct) expected behavior of a component [13].
none of them generated the oracle or test cases for multiple modules, i.e. component testing and integration testing.

5.3 Summary

We proposed a black-box testing method that derives important test cases from usage statistics. The method specified large components with TFM module interface specifications and associated module usage statistics with the TFM module interface specification. Finally, we showed that the method could generate a prioritized list of testing traces for component testing and integration testing from the TFM module interface specification with usage statistics.

Consequently, the proposed method enables a product manager to make a release decision with a rationale, “the important use cases specified in the usage statistics are tested and have no error.”
APPENDIX A. FUNCTIONS AND NOTATIONS FOR PREDICATES
Primitive functions on event descriptors

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM(e)</td>
<td>PGM(event descriptor) →(string)</td>
<td>A function that returns the name of the access program in the event descriptor e</td>
<td>If e=(PGM:myfunc,in&lt;int&gt;:=100,out&lt;int&gt;:=1), then PGM(e)=myfunc</td>
</tr>
<tr>
<td>V(e)</td>
<td>V(event descriptor) → (data type of V)</td>
<td>A function that returns the value of V before the event of the event descriptor e (V may be any input variable)</td>
<td>If e=(PGM:myfunc,in&lt;int&gt;:=100,out&lt;int&gt;:=1), then in(e)=100</td>
</tr>
<tr>
<td>V'(e)</td>
<td>V'(event descriptor) → (data type of V)</td>
<td>A function that returns the value of V after the event of the event descriptor e (V may be any output variable)</td>
<td>If e=(PGM:myfunc,in&lt;int&gt;:=100,out&lt;int&gt;:=1), then out'(e)=1</td>
</tr>
</tbody>
</table>

Notation for traces

<table>
<thead>
<tr>
<th>Notation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>This is an empty trace.</td>
</tr>
<tr>
<td>T1 T2</td>
<td>A period ‘ ’ concatenates two traces.</td>
</tr>
</tbody>
</table>

Basic functions on traces

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Len(T)</td>
<td>Len(trace) →(integer)</td>
<td>A function that returns the length of the trace T (the number of event descriptors in the trace)</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, Len(T)=3</td>
</tr>
<tr>
<td>r(n,T)</td>
<td>r(integer) × (trace) → (event descriptor)</td>
<td>A function that returns the n-th most recent event descriptor in the trace T. ( r(n,_{-})=\text{Undefined} )</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, r(1,T)=E3, r(2,T)=E2, r(3,T)=E1, r(4,T)=undefined.</td>
</tr>
<tr>
<td>oldest(T)</td>
<td>oldest(trace) →(event descriptor)</td>
<td>A function that returns the first (oldest) event descriptor in the trace T. ( \text{oldest}(_{-})=_ )</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, oldest(T)=E1</td>
</tr>
<tr>
<td>p(n,T)</td>
<td>p(integer) × (trace) → (trace)</td>
<td>A function that returns the prefix trace of ( r(n,T) ) in the trace T. ( p(n,_{-})=\text{Undefined} )</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, p(1,T)=E1.E2, p(2,T)=E1, p(3,T)=._</td>
</tr>
<tr>
<td>subseq(T)</td>
<td>subseq(trace) → (trace)</td>
<td>A function that returns the trace ( T' ) such that ( \text{subseq}(T) = T' )</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, subseq(T)=E2.E3</td>
</tr>
<tr>
<td>r_call(pg,T)</td>
<td>r_call(string) × (trace) → (event descriptor)</td>
<td>A function that returns the most recent event descriptor which access program name is pg in the trace T</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, and E3 is invoked by a program pg and E2 and E3 are not invoked by pg, then r_call(pg,T)=E3</td>
</tr>
</tbody>
</table>

Useful function generators on traces and predicates

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Function</th>
<th>Semantics</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>exist(P,T)</td>
<td>exist.(predicate) × (trace) →(boolean)</td>
<td>A function that returns true if and only if there exists an event descriptor in T that satisfies P, otherwise false</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, and E1 satisfies P, then exist(P,T)=true.</td>
</tr>
<tr>
<td>oldst(P,T)</td>
<td>oldst.(predicate) × (trace) →(event descriptor)</td>
<td>A function that returns the most recent event descriptor in T that satisfies P</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, and E1.E2 and E3 satisfy P, then oldst(P,T)=E3.</td>
</tr>
<tr>
<td>oldest_st(P,T)</td>
<td>oldest_st.(predicate) × (trace) →(event descriptor)</td>
<td>A function that returns the first (oldest) event descriptor in T that satisfies P</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, and E1.E2 and E3 satisfy P, then oldest_st(P,T)=E1.</td>
</tr>
<tr>
<td>idx_call(P,T)</td>
<td>idx_call(string) × (trace) →(integer)</td>
<td>A function that returns the index of the most recent event descriptor in T that satisfies P. The index starts from 1.</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, and E1.E2 and E3 satisfy P, then idx_call(P,T)=1.</td>
</tr>
<tr>
<td>idx_oldest_st(P,T)</td>
<td>idx_oldest_st.(predicate) × (trace) →(integer)</td>
<td>A function that returns the index of the first (oldest) event descriptor in T that satisfies P. The index starts from 1.</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, and E1.E2 and E3 satisfy P, then idx_oldest_st(P,T)=1.</td>
</tr>
<tr>
<td>extract(P,T)</td>
<td>extract.(predicate) × (trace) →(trace)</td>
<td>A function that returns a trace that has only the event descriptors in T that satisfies P</td>
<td>If T=E1.E2.E3 where E1 is an event descriptor, and E1 and E3 satisfy P, then extract(P,T)=E1.E3.</td>
</tr>
</tbody>
</table>
APPENDIX B.  FWS MODULE GUIDE

• Behavior-Hiding Modules
  – Controller
    Secret: How to control the execution sequence of FWS
  – MessageGenerator:
    Secret: How to obtain monitored data and transmit averaged data in a message

• Device Interface Modules
  – Sensor Device Driver
    Secret: How to monitor and control the wind speed sensors and whale sensors.
  – Transmitter Device Driver
    Secret: How to control the transmitter

• Software-Design-Hiding Modules
  – SensorMonitor
    Secret: How to obtain data from sensors and store it for later retrieval.
  – DataBanker
    Secret: How to store the most recent wind data and whale data
  – Averager
    Secret: How to process the current DataBanker data to produce a current wind data and whale data estimate
Figure B.1  FWS Module Hierarchical Structure

Figure B.2  FWS Uses Relation
APPENDIX C. FWS MODULE INTERFACE SPECIFICATION

Note: output behaviors definition and auxiliary function definition tables are transposed to save paper space. For example, the trace semantics part of SensorDriver module interface is equivalent to Figure C.1.
## Trace Postcondition

### Monitored Variables

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Occurred</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Executed Count

<table>
<thead>
<tr>
<th>Total Sample</th>
<th>Occurrence Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Example

$\text{sensorData}(T)$

$\equiv \text{Pre}(1,T,\text{SensorDriver_get}) = \{ \text{True}, i \}$

$\text{new SensorReading}'\text{windSpeed}(r(i,T))', '\text{whalePassing}(r(i,T))''$

$\text{T.(PGM:SensorDriver_get,'windSpeed<int>:UNEVAL,'whalePassing<Boole}an>:UNEVAL,\text{sensorData'<SensorReading}:UNEVAL) \rightarrow \text{T.(PGM:SensorDriver_get,'windSpeed<int>:aa,'whalePassing<Boole}an>:bb,\text{sensorData'<SensorReading>:new SensorReading(aa,bb))}$

#### Constraints

$97 \leq \text{aa} \leq \text{INTMAX}$

$\text{bb} = \text{TRUE} \lor \text{FALSE}$

#### Figure C.1 SensorDriver module Before Transpose
Module Index

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>Behavior Hiding</td>
</tr>
<tr>
<td>MessageGenerator</td>
<td>Software Design</td>
</tr>
<tr>
<td>SensorMonitor</td>
<td>Hiding</td>
</tr>
<tr>
<td>Averager</td>
<td></td>
</tr>
<tr>
<td>DataBanker</td>
<td></td>
</tr>
<tr>
<td>SensorDriver</td>
<td>Device Interface</td>
</tr>
<tr>
<td>TransmitDriver</td>
<td></td>
</tr>
</tbody>
</table>

Auxiliary Functions

Trace Predicates

\[ \text{r}(i, T) = \text{evt} \quad \text{r}(i, T) \neq \text{evt} \quad \text{r}(i, T) = \text{Empty} \]

\[ \text{Pref}(i, T, \text{evt}) \equiv \{ \text{TRUE}, i \} \quad \text{Pref}(i+1, T, \text{evt}) \equiv \{ \text{FALSE}, i \} \]

Trace Predicates

\[ \text{r}(i, T) = \text{evt} \quad \text{r}(i, T) \neq \text{evt} \quad \text{r}(i, T) = \text{Empty} \]

\[ \text{Post}(i, T, \text{evt}) \equiv \{ \text{TRUE}, i \} \quad \text{Post}(i+1, T, \text{evt}) \equiv \{ \text{FALSE}, i \} \]

suffix(i, T) \equiv \text{subseq}(i, T)

Trace Predicates

\[ '\text{quit}' \]

\[ '\text{quit}' \]

\[ \text{T.loopEvents}() \equiv \{ \text{TRUE} \} \]

Data Type

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageFor</td>
<td>Attributes</td>
</tr>
<tr>
<td>matWind</td>
<td>windSpeed: int</td>
</tr>
<tr>
<td>MessageFor</td>
<td>Attributes</td>
</tr>
<tr>
<td>matWhale</td>
<td>whalePassing: bool</td>
</tr>
<tr>
<td>SensorReading</td>
<td>Attributes</td>
</tr>
<tr>
<td>windSpeed: int</td>
<td>whalePassing: bool</td>
</tr>
<tr>
<td>whalePassing: bool</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{T.loopEvents}() \equiv \{ \text{TRUE} \} \]
Controller Module Interface Specification

Trace Event Syntax

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FWS_&lt;init&gt;</td>
<td>quit:Boolean</td>
<td>NONE</td>
<td>(PGM:FWS_&lt;init&gt;, 'quit&lt;Boolean&gt;:*)</td>
</tr>
</tbody>
</table>

Trace Semantics

<table>
<thead>
<tr>
<th>Trace Predicates</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>idx_oldest_st(PGM(r(1,T))=FWS_&lt;init&gt;, T)=i0</td>
<td>Vj { ¬(Predicate(i,j)) } where i is a number of rows in this table, and j is a number of column in this table, and i and j don't include this cell's number.</td>
</tr>
<tr>
<td>[idx_oldest_st(PGM(r(1,T))=DataBanker_&lt;init&gt;,T)=i1] ^ [1&lt;i1]</td>
<td></td>
</tr>
<tr>
<td>[idx_oldest_st(PGM(r(1,T))=MessageGenerator_&lt;init&gt;,T)=i2] ^ [i1&lt;i2]</td>
<td></td>
</tr>
<tr>
<td>[idx_oldest_st(PGM(r(1,T))=SensorMonitor_&lt;init&gt;,T)=i3] ^ [i2&lt;i3]</td>
<td></td>
</tr>
<tr>
<td>suffix(i3,T)=loopEvents()</td>
<td></td>
</tr>
<tr>
<td>Len( extract ( [(PGM=SensorMonitor_run)^('quit=FALSE] , T ) ) = n</td>
<td></td>
</tr>
</tbody>
</table>

Output Behavior Definition

| None(T) ≡ None |

Trace Postcondition

T. | PGM:FWS_<init>, 'quit<Boolean>:qq)→
(PGM:DataBanker_<init>.
(PGM:MessageGenerator_<init>,object'<MessageGenerator>:obj1,UNEVAl).
(PGM:MessageGenerator_run,'obj<MessageGenerator>:obj1,UNEVAl).
×n (*See a note below)
(ReturnPGM:FWS_<init>) UnDefined

Monitored Variable Constraints

qq=FALSE v TRUE -

Executed Count

1 0

Total Sample

1

Occurrence Probability (%)

100 0

*Note: [T] × n is the abbreviation of n times repeated T sequence. For example, [T] × 3 means T.T.T. This notation is introduced here to save paper space.
MessageGenerator Module Interface Specification

Trace Event Syntax

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Input Event Descriptor</th>
<th>Output Event Descriptor</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;init&gt;</td>
<td>NONE</td>
<td>object:MessageGenerator</td>
<td>(PGM:MessageGenerator_&lt;init&gt;, object:<em>MessageGenerator:</em>')</td>
</tr>
<tr>
<td>run</td>
<td>thisObj:MessageGenerator</td>
<td>NONE</td>
<td>(PGM:MessageGenerator_run, 'thisObj:SensorMonitor:*')</td>
</tr>
</tbody>
</table>

Trace Semantics

<table>
<thead>
<tr>
<th>Trace Predicates</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pre(I,T,MessageGenerator_&lt;init&gt;)={TRUE,1} ^ Pre(I,T,MessageGenerator_&lt;init&gt;)={TRUE,1}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Post(I,T,MessageGenerator_run)={TRUE,1} ^ Post(I,T,Average_getWindSpeed)={TRUE,2}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Post(I,2,T,TransmitDriver_send)={TRUE,3} ^ Post(I,3,T,Average_getWhale)={TRUE,4}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Pre(I,T,Average_getWhale)={TRUE,5}] ^ [Pre(I,T,Average_getWhale)={TRUE,5}]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[whalePassing'(r(4,T))≠whalePassing'(r(5,T))] ^ [whalePassing'(r(4,T))≠whalePassing'(r(5,T))]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Post(I,4,T,TransmitDriver_sendToWhaleSurveyCenter)={TRUE,5}] ^ [msg(r(5,T))=new MessageFormatWhale(TRUE)]</td>
<td></td>
</tr>
</tbody>
</table>

Output Behavior Definition

| NONE(T)=NONE | NONE(T)=NONE | object(T)=new MessageGenerator | Undefined |

Trace Postcondition

| T.(PGM:MessageGenerator_run) → T.(InvokePGM:MessageGenerator_run) |
| T.(InvokePGM:MessageGenerator_run) |
| (PGM:Average_getWindSpeed,averageData<{'ag':ag,UNEVAL}) |
| (PGM:TransmitDriver_sendWindSpeed,msg<MessageFormatWind:msg,UNEVAL>) |
| (PGM:Average_getWhale,whalePassing'boolag:FALSE,UNEVAL) |
| (ReturnPGM:MessageGenerator_run) |

| | T.(PGM:MessageGenerator_run) → T.(InvokePGM:MessageGenerator_run) |
| | (PGM:Average_getWindSpeed,averageData<{'ag':ag,UNEVAL}) |
| | (PGM:TransmitDriver_sendWindSpeed,msg<MessageFormatWind:msg,UNEVAL>) |
| | (PGM:Average_getWhale,whalePassing'boolag:FALSE,UNEVAL) |
| | (ReturnPGM:MessageGenerator_run) |

| T.(PGM:MessageGenerator_<init>) | object:*MessageGenerator:*' | undefined |

Monitored Variable Constraints

| monitored | - | - | - | - |

Executed Count

| Total Sample | 24 | 1 | 0 |

Occurrence Probability

| 91.667 | 8.333 | 100 | 0 |
### SensorMonitor Module Interface Specification

#### Trace Event Syntax

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>thisObj:SensorMonitor</td>
<td>NONE</td>
<td>(PGM:SensorMonitor_run&lt;thisObj '&lt;SensorMonitor&gt;/='' )</td>
</tr>
</tbody>
</table>

#### Trace Semantics

**Pre**:
- Pre(1,T, SensorMonitor_<init>) = {True, i0}
- Pre(1,T, SensorMonitor_run, 'thisObj:SensorMonitor') = {True, i1}
- Pre(T, Thread_Sleep = TRUE) = sleepTime < (T) = 1000
- Pre(T, SensorDriver_get = TRUE) = sensorData.get (T) = d1
- Post(T, DataBanker_write = TRUE) = sensorDataIn (T) = sensorData (T) = d2

**Post**:
- Post(T, Thread_Sleep) = sleepTime < 1000
- Post(T, DataBanker_write) = sensorDataIn = sensorData = d2

**Output Behavior Definition**
- NONE(T) = NONE object(T) = new SensorMonitor
- Undefined

#### Trace Postcondition

- T.(PGM:SensorDriver_get,'windSpeed<int>:UNIVAL',sensorData=sensorDataIn \( \Rightarrow \) undefined
- T.(PGM:DataBanker_write,'sensorDataIn=sensorData',sensorDataIn \( \Rightarrow \) undefined
- T.(PGM:SensorMonitor_run)

**Monitor Variable Constraints**
- 

**Executed Count**
- 114

**Total Sample**
- 114

**Occurrence Probability**
- 100

\( V_{ij} = \bigvee_{i} \bigwedge_{j} \text{Predicate}(i, j) \)

where i is a number of row in this table, and j is a number of column in this table, and i and j don't include this cell's number.
**Averager Module Interface Specification**

**Trace Event Syntax**

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
</table>
| getWindSpeed   | NONE           | averageData     | PGM: Averager_getWindSpeed, averageData
| getWhale       | NONE           | whalePassing    | PGM: Averager_getWhale, whalePassing |

**Trace Semantics**

Trace Predicates

```
Pre(1, T, Averager_getWindSpeed) = \{TRUE, 1\}
Post(0, T, DataBanker_read) = \{TRUE, 1\}

ex(\{\[Pre(0, T, DataBanker_write) = TRUE, 2\] ^
[SensorData\{\(r(3, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(4, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(5, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(6, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(7, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(8, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(9, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(10, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(11, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(12, T)\) = \{*, FALSE\}\}] ^
[SensorData\{\(r(13, T) = \{*, FALSE\}\}] ^
[SensorData\{\(r(14, T) = \{*, FALSE\}\}] ^
[SensorData\{\(r(15, T) = \{*, FALSE\}\}] ^
[SensorData\{\(r(16, T) = \{*, FALSE\}\}] ^
[SensorData\{\(r(17, T) = \{*, FALSE\}\}] ^
[SensorData\{\(r(18, T) = \{*, FALSE\]\} \}
\}, T) \rightarrow \{TRUE\}
```

**Output Behavior Definition**

```
averageData(\(T\)) = \(\sum_{j} \text{readData}\{\(r(3, T)\)\}.listElement(j) / \text{readData}\{\(r(3, T)\)\}.length\)
```

```
whilePassing(\(T\)) = True
whilePassing(\(T\)) = False
```

**Trace Postcondition**

```
T.\{PGM: Averager_getWhale, whalePassing = Boolean\} \rightarrow
T.\{invokePGM(Averager_getWhale),
(PGM: DataBanker_read, readData\{\(\text{List<int>}\)\} \rightarrow \text{readData}\{\(r(3, T)\)\}, UNET, UNET),
(PGM: Averager_getWhale, \text{whilePassing} = True) \}
```

```
T.\{PGM: Averager_getWhale, whalePassing = Boolean\} \rightarrow
T.\{invokePGM(Averager_getWhale),
(PGM: DataBanker_read, readData\{\(\text{List<int>}\)\} \rightarrow \text{readData}\{\(r(3, T)\)\}, UNET, UNET),
(PGM: Averager_getWhale, \text{whilePassing} = False) \}
```

**Monitored Variable Constraints**

```
\text{listAt}\{0\} \rightarrow \text{whilePassing} = False \}
```

**Executed count**

```
24 3 21 0
```

**Total Sample**

```
24 24 0 0
```

**Occurrence Probability**

```
100 12.5 87.5 0
```
DataBanker Module Interface Specification

Trace Event Syntax

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>NONE</td>
<td>NONE</td>
<td>(PGM:DataBanker_init;'static list=list&lt;SensorReading&gt;&gt;*)</td>
</tr>
<tr>
<td>write</td>
<td>sensorDataIn:SensorReading</td>
<td>NONE</td>
<td>(PGM:DataBanker_write;'static list=list&lt;SensorReading&gt;&gt;<em>, 'sensorDataIn:SensorReading&gt;&gt;</em>)</td>
</tr>
<tr>
<td>read</td>
<td>NONE</td>
<td>readData:list&lt;SensorReading&gt;</td>
<td>(PGM:DataBanker_read;'static list=list&lt;SensorReading&gt;&gt;<em>, 'readData:list&lt;SensorReading&gt;&gt;</em>)</td>
</tr>
</tbody>
</table>

Trace Semantics

Trace Predicates

Pre(L.T.DataBanker_init)=(TRUE,i0) Pre(L.T.DataBanker_write)=(TRUE,i0) Pre(L.T.DataBanker_read)=(TRUE,i0) Pre(i0,T.DataBanker_write)=(TRUE,i1) Pre(i1,T.DataBanker_write)=(TRUE,i2) Pre(i2,T.DataBanker_write)=(TRUE,i3) Pre(i3,T.DataBanker_write)=(TRUE,i4) Pre(i4,T.DataBanker_write)=(TRUE,i5) Pre(i5,T.DataBanker_init)=(TRUE,i6)

Output Behavior Definition

NONE(T)=NONE NONE(T)=NONE readData(T)=new List('sensorDataIn[0],...,'sensorDataIn[i-1],...,'sensorDataIn[5],...,'sensorDataIn[5]) Undefined

Trace Postcondition

T.(PGM:DataBanker_init) -> T.(PGM:DataBanker_init)
T.(PGM:DataBanker_write,'sensorDataIn<SensorReading>=sd) -> T.(PGM:DataBanker_read,'sensorDataIn<SensorReading>=sd)
T.(PGM:DataBanker_read,'readData'<List<SensorReading>>UNEQUAL) -> T.(PGM:DataBanker_read,'readData'<List<SensorReading>>new list[sd,sd1,sd2,sd3,sd4]) Undefined

Monitored Variable Constraints

- - - - - -

Executed Count

1 228 48 0

Total Sample

1 228 48 0

Occurrence Probability

100 100 100 0
**SensorDriver Module Interface Specification**

### Trace Event Syntax

<table>
<thead>
<tr>
<th>Function Call Event Descriptor</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SensorDriver_get</td>
<td>windSpeed:int, whalePassing:Boolean</td>
<td>sensordata:SensorReading</td>
<td>(PGM:SensorDriver_get,'windSpeed&lt;int&gt;','whalePassing&lt;Boolean&gt;','sensordata'&lt;SensorReading&gt;)</td>
</tr>
</tbody>
</table>

### Trace Semantics

<table>
<thead>
<tr>
<th>Trace Predicates</th>
<th>Output Behavior Definition</th>
<th>Trace Postcondition</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Monitored Variable Constraints</th>
<th>Executed Count</th>
<th>Total Sample</th>
<th>Occurrence Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTMIN ≤ aa ≤ INTMAX, bb = TRUE v FALSE</td>
<td>114</td>
<td>114</td>
<td>0</td>
</tr>
</tbody>
</table>
**TransmitDriver Module Interface Specification**

**Trace Event Syntax**

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Input Variable</th>
<th>Output Variable</th>
<th>Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendWindSpeed</td>
<td>msg:MessageFormatWind</td>
<td>msg:MessageFormatWind</td>
<td>(PGM:TransmitDriver_sendWindSpeed,'msg&lt;MessageFormatWind&gt;:<em>msg&lt;MessageFormatWind&gt;:</em>)</td>
</tr>
<tr>
<td>sendToWhaleSurveyCenter</td>
<td>msg:MessageFormatWhale</td>
<td>msg:MessageFormatWhale</td>
<td>(PGM:TransmitDriver_sendToWhaleSurveyCenter,'msg&lt;MessageFormatWhale&gt;:<em>msg&lt;MessageFormatWhale&gt;:</em>)</td>
</tr>
</tbody>
</table>

**Trace Semantics**

<table>
<thead>
<tr>
<th>Trace Predicates</th>
<th>Output Behavior Definition</th>
<th>Trace Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>msg(T)=msg(r(I,T))</td>
<td>Undefned</td>
</tr>
</tbody>
</table>

**Unmonitored Variable Constraints**

<table>
<thead>
<tr>
<th>Executed Count</th>
<th>Total Sample</th>
<th>Occurrence Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>114</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>100</td>
</tr>
</tbody>
</table>

**Unmonitored Variable Constraints**

<table>
<thead>
<tr>
<th>Executed Count</th>
<th>Total Sample</th>
<th>Occurrence Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>114</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>100</td>
</tr>
</tbody>
</table>
APPENDIX D. TESTING TRACE GENERATION ALGORITHM
INTERMEDIATE STATES

The intermediate states of the testing trace in the test case generation algorithm are described for particular stages.
Stage = 0

Step 3, 4.
• This equivalence class of traces is chosen with probability 1.
• The consequent of the trace postcondition is insert to the testing trace.
• The meta-variable qq is added to the meta-variable map, and the constraint of monitored variable 'quit' is added to the collection, CON.

Step 5.
Go back to Step3. This Step3 evaluates the next unevaluated event (PGM:DataBanker_init)

Events beyond the upper-bound length of testing trace are truncated. This example has the upper-bound 37.

Meta-Variable Map=

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>qq</td>
<td>Monitored Variable</td>
</tr>
<tr>
<td>ob1</td>
<td>new MessageGenerator Instance</td>
</tr>
<tr>
<td>ob2</td>
<td>new SensorMonitor Instance</td>
</tr>
</tbody>
</table>

CON=
qq = FALSE
An equivalence class of traces is chosen. In this case, the one with "whalePassing=FALSE" is chosen with the probability 87.5/100.
Stage = t + 2

*Note: Although the random choice is done twice, the probability of getting "whalePassing = FALSE" at this step remains 87.5/100, since the probability of getting "whalePassing = FALSE" is 87.5/100 no matter what value "whalePassing" has before, which leads to

\[ P(\text{whalePassing} = \text{FALSE}) = \sum_{\text{all } \text{values}} P(\text{whalePassing} = \text{FALSE}) = \frac{87.5}{100} = \frac{87.5}{100} \]

where \( P(\text{whalePassing} = \text{FALSE}) \) is the probability of eventually getting "whalePassing = v" at this step, and \( P(\text{whalePassing} = \text{FALSE}) \) is the probability of getting "whalePassing = v" one step before this step.

MetaVariableTable

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>CON</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc1</td>
<td>Monitor1 Variable</td>
<td>INTMIN ≤ a1 ≤ INTMAX</td>
</tr>
<tr>
<td>bb1</td>
<td>Monitor1 Variable</td>
<td>INTMIN ≤ a1 ≤ INTMAX</td>
</tr>
<tr>
<td>cc2</td>
<td>Monitor1 Variable</td>
<td>INTMIN ≤ a1 ≤ INTMAX</td>
</tr>
<tr>
<td>cc3</td>
<td>Monitor1 Variable</td>
<td>INTMIN ≤ a1 ≤ INTMAX</td>
</tr>
<tr>
<td>dd1</td>
<td>(windSpeed = a1, whalePassing = bb1)</td>
<td>bb1 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd2</td>
<td>(windSpeed = a2, whalePassing = bb2)</td>
<td>bb2 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd3</td>
<td>(windSpeed = a3, whalePassing = bb3)</td>
<td>bb3 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd4</td>
<td>(windSpeed = a4, whalePassing = bb4)</td>
<td>bb4 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd5</td>
<td>(windSpeed = a5, whalePassing = bb5)</td>
<td>bb5 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd6</td>
<td>(windSpeed = a6, whalePassing = bb6)</td>
<td>bb6 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd7</td>
<td>(windSpeed = a7, whalePassing = bb7)</td>
<td>bb7 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd8</td>
<td>(windSpeed = a8, whalePassing = bb8)</td>
<td>bb8 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd9</td>
<td>(windSpeed = a9, whalePassing = bb9)</td>
<td>bb9 = TRUE v FALSE</td>
</tr>
<tr>
<td>dd10</td>
<td>(windSpeed = a10, whalePassing = bb10)</td>
<td>bb10 = TRUE v FALSE</td>
</tr>
</tbody>
</table>

Step 3.

- A consequent of the trace postcondition of "Averager_getWhale" is inserted.

Step 3. Query again and get the possibly-satisfiable equivalence class.

The possibly-satisfiable equivalence classes are
1) the one with "whalePassing(T) = TRUE;"
2) the one with "whalePassing(T) = FALSE;"

Step 4. This case "whalePassing = FALSE" is chosen with the probability 87.5/100.*
Step 6. Generate monitored variable values and populate all values in this testing trace.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>qq</td>
<td>FALSE</td>
</tr>
<tr>
<td>ob1</td>
<td>new MessageGenerator Instance</td>
</tr>
<tr>
<td>ob2</td>
<td>new SensorMonitor Instance</td>
</tr>
<tr>
<td>aa1</td>
<td>45</td>
</tr>
<tr>
<td>bb1</td>
<td>FALSE</td>
</tr>
<tr>
<td>dd1</td>
<td>{windSpeed=aa1, whalePassing=bb1}</td>
</tr>
<tr>
<td>aa2</td>
<td>51</td>
</tr>
<tr>
<td>bb2</td>
<td>FALSE</td>
</tr>
<tr>
<td>dd2</td>
<td>{windSpeed=aa2, whalePassing=bb2}</td>
</tr>
<tr>
<td>aa3</td>
<td>50</td>
</tr>
<tr>
<td>bb3</td>
<td>FALSE</td>
</tr>
<tr>
<td>dd3</td>
<td>{windSpeed=aa3, whalePassing=bb3}</td>
</tr>
<tr>
<td>aa4</td>
<td>52</td>
</tr>
<tr>
<td>bb4</td>
<td>FALSE</td>
</tr>
<tr>
<td>dd4</td>
<td>{windSpeed=aa4, whalePassing=bb4}</td>
</tr>
<tr>
<td>aa5</td>
<td>55</td>
</tr>
<tr>
<td>bb5</td>
<td>FALSE</td>
</tr>
<tr>
<td>dd5</td>
<td>{windSpeed=aa5, whalePassing=bb5}</td>
</tr>
<tr>
<td>rd1</td>
<td>List&lt;SensorReading&gt; List(dd1, dd2, dd3, dd4, dd5)</td>
</tr>
<tr>
<td>cc1</td>
<td>51</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


