2012

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

Katsuya Iwata
Iowa State University

Recommended Citation

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

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

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

LIST OF FIGURES

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

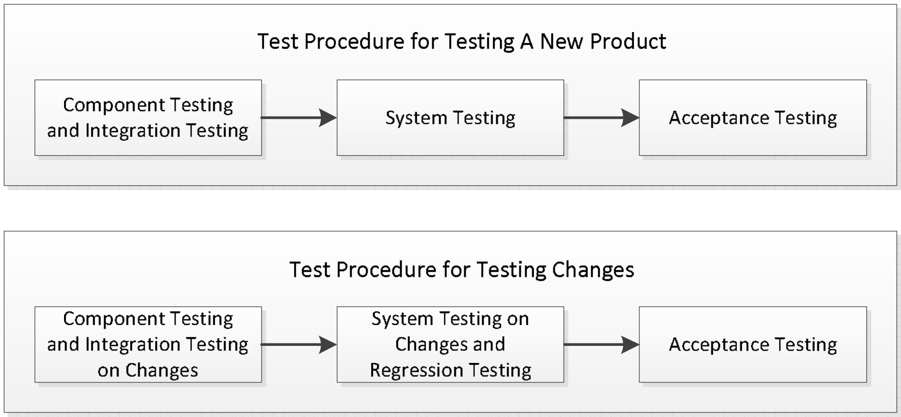

Test Procedure for Testing A New Product
Component Testing and Integration Testing
System Testing
Acceptance Testing
Test Procedure for Testing Changes
Component Testing and Integration Testing on Changes
System Testing on Changes and Regression Testing
Acceptance Testing

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[3] 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[4] 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

step, process, or data definition 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. [5] 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.

[5] 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[6] 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[7] is a

[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.

$$
\begin{array}{lcl}
event\_descriptor & ::= & (\mathrm{PGM}: func, parameter\_list) \\
func & ::= & \text{Name of an access program} \\
parameter\_list & ::= & parameter, parameter\_list | parameter | \_ \\
parameter & ::= & 'input\_variable < type >: value | output\_variable' < type >: value \\
input\_variable & ::= & \text{Name of an input variable} \\
output\_variable & ::= & \text{Name of an output variable} \\
value & ::= & constant
\end{array}
$$

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**[12] `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[13] of inputs and their data type (section 1 in Figure. 2.4);

[12] 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

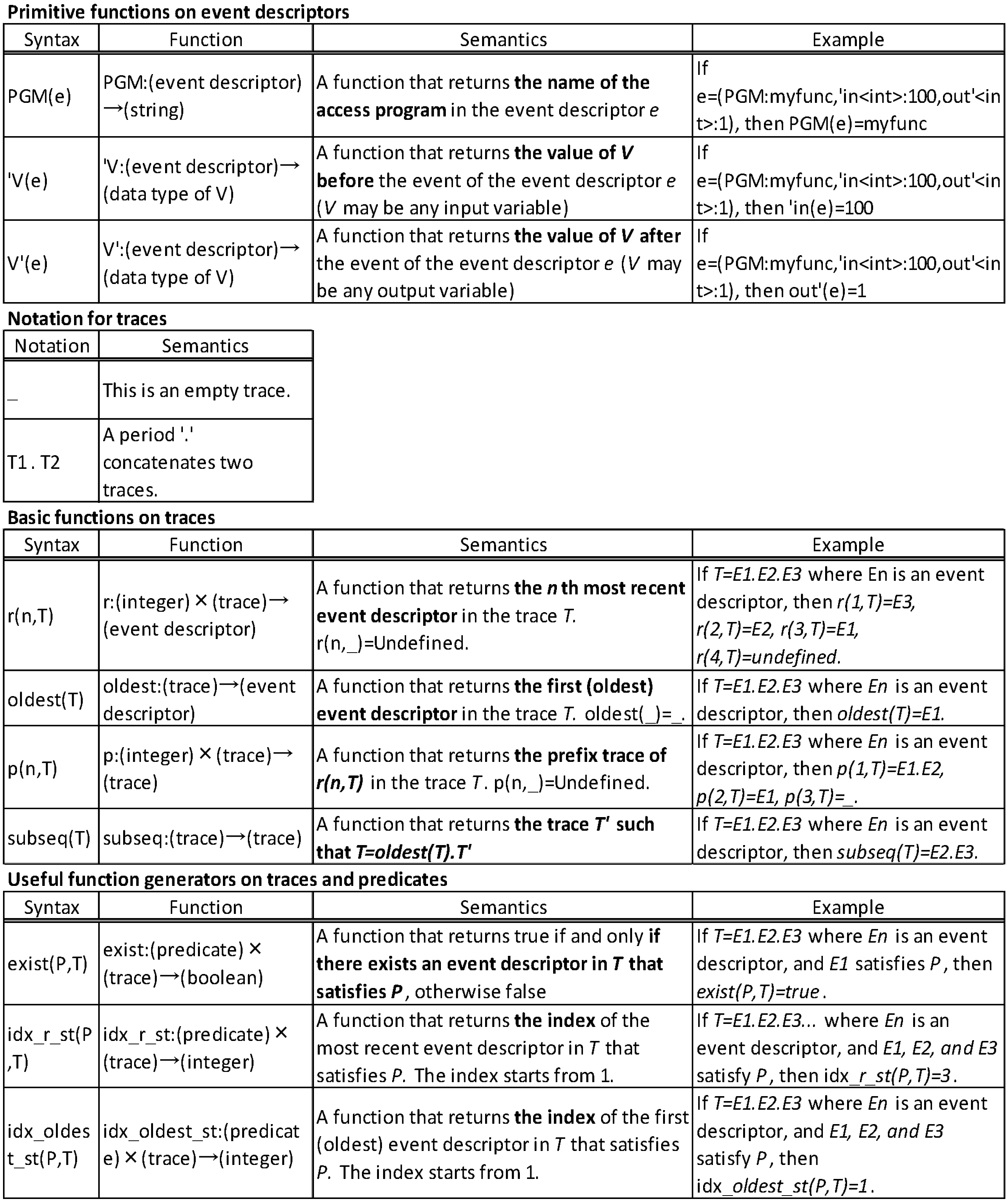
[13] A description is complete when all required description for the document are described[33].

- 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 `inIntegerRange(x)` ≡ `(Lower bound value of int)` ≤ `x` ≤ `(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

Syntax	Function	Semantics	Example
PGM(e)	PGM:(event descriptor)→(string)	A function that returns **the name of the access program** in the event descriptor *e*	If e=(PGM:myfunc,'in<int>:100,out'<int>:1), then PGM(e)=myfunc
'V(e)	'V:(event descriptor)→(data type of V)	A function that returns **the value of *V* before** the event of the event descriptor *e* (*V* may be any input variable)	If e=(PGM:myfunc,'in<int>:100,out'<int>:1), then 'in(e)=100
V'(e)	V':(event descriptor)→(data type of V)	A function that returns **the value of *V* after** the event of the event descriptor *e* (*V* may be any output variable)	If e=(PGM:myfunc,'in<int>:100,out'<int>:1), then out'(e)=1

Notation for traces

Notation	Semantics
_	This is an empty trace.
T1 . T2	A period '.' concatenates two traces.

Basic functions on traces

Syntax	Function	Semantics	Example
r(n,T)	r:(integer) × (trace)→(event descriptor)	A function that returns **the *n* th most recent event descriptor** in the trace *T*. r(n,_)=Undefined.	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.*
oldest(T)	oldest:(trace)→(event descriptor)	A function that returns **the first (oldest) event descriptor** in the trace *T*. oldest(_)=_.	If *T=E1.E2.E3* where *En* is an event descriptor, then *oldest(T)=E1.*
p(n,T)	p:(integer) × (trace)→(trace)	A function that returns **the prefix trace of *r(n,T)*** in the trace *T*. p(n,_)=Undefined.	If *T=E1.E2.E3* where *En* is an event descriptor, then *p(1,T)=E1.E2, p(2,T)=E1, p(3,T)=_.*
subseq(T)	subseq:(trace)→(trace)	A function that returns **the trace *T'* such that *T=oldest(T).T'***	If *T=E1.E2.E3* where *En* is an event descriptor, then *subseq(T)=E2.E3.*

Useful function generators on traces and predicates

Syntax	Function	Semantics	Example
exist(P,T)	exist:(predicate) × (trace)→(boolean)	A function that returns true if and only **if there exists an event descriptor in *T* that satisfies *P***, otherwise false	If *T=E1.E2.E3* where *En* is an event descriptor, and *E1* satisfies *P*, then *exist(P,T)=true*.
idx_r_st(P,T)	idx_r_st:(predicate) × (trace)→(integer)	A function that returns **the index** of the most recent event descriptor in *T* that satisfies *P*. The index starts from 1.	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*.
idx_oldest_st(P,T)	idx_oldest_st:(predicate) × (trace)→(integer)	A function that returns **the index** of the first (oldest) event descriptor in *T* that satisfies *P*. The index starts from 1.	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*.

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.

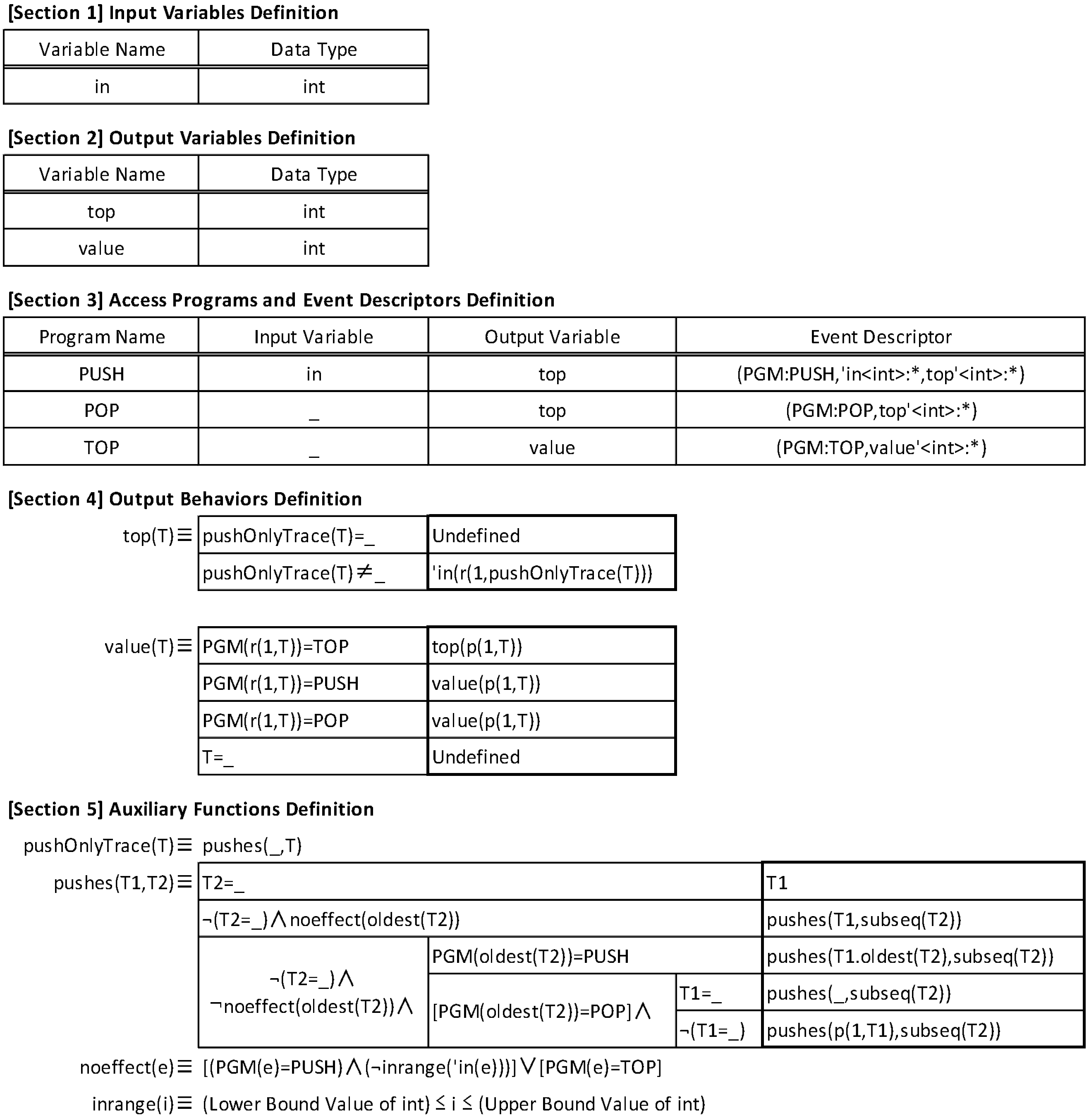
  TFM may use a **tabular format**. The **tabular format** to define **output behavior definition** and **auxiliary function definition** has two parts, the **trace predicates columns** that is immediately right of the equivalence symbol '≡', and the **value column** that is enclosed by a heavy line. The 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 `pushes(T1,T2)` ≡ `pushes(T1,subseq(T2))`, 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, `(PGM: TOP,value'<int>:*)`. This '`*`' means that the value may be any value in its variable data type.

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

Figure 2.3 Closed-form Solution for value - Conditional Expressions Format



[Section 1] Input Variables Definition

Variable Name	Data Type
in	int

[Section 2] Output Variables Definition

Variable Name	Data Type
top	int
value	int

[Section 3] Access Programs and Event Descriptors Definition

Program Name	Input Variable	Output Variable	Event Descriptor
PUSH	in	top	(PGM:PUSH,'in<int>:*,top'<int>:*)
POP	_	top	(PGM:POP,top'<int>:*)
TOP	_	value	(PGM:TOP,value'<int>:*)

[Section 4] Output Behaviors Definition

top(T)≡

pushOnlyTrace(T)=_	Undefined
pushOnlyTrace(T)≠_	'in(r(1,pushOnlyTrace(T)))

value(T)≡

PGM(r(1,T))=TOP	top(p(1,T))
PGM(r(1,T))=PUSH	value(p(1,T))
PGM(r(1,T))=POP	value(p(1,T))
T=_	Undefined

[Section 5] Auxiliary Functions Definition

pushOnlyTrace(T)≡ pushes(_,T)

pushes(T1,T2)≡

T2=_			T1
¬(T2=_)∧noeffect(oldest(T2))			pushes(T1,subseq(T2))
¬(T2=_)∧ ¬noeffect(oldest(T2))∧	PGM(oldest(T2))=PUSH		pushes(T1.oldest(T2),subseq(T2))
	[PGM(oldest(T2))=POP]∧	T1=_	pushes(_,subseq(T2))
		¬(T1=_)	pushes(p(1,T1),subseq(T2))

noeffect(e)≡ [(PGM(e)=PUSH)∧(¬inrange('in(e)))]∨[PGM(e)=TOP]

inrange(i)≡ (Lower Bound Value of int) ≤ i ≤ (Upper Bound Value of int)

Figure 2.4 TFM Module Interface Document for a Stack Module

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

Figure 2.5 Closed-form Solution for pushes - Conditional Expressions Format

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[16] 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[17] of the use case. Specifically, statistical software testing develops usage[18] 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] This **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[19]. 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 2.1 Example of Usage Statistics

User Group	**System Mode**	**Usage**	**Occurrence Probability (%)**	**Criticality (1..100)**
Buyer	Customer Mode	Search products	80	20
		Purchase products	20	80
		Register selling products	0	0
		Website appearance configuration	0	0
		Database operations	0	0
Seller	Customer Mode	Search products	20	10
		Purchase products	0	0
		Register selling products	80	90
		Website appearance configuration	0	0
		Database operations	0	0
Administrator	Maintenance Mode	Search products	10	5
		Purchase products	10	5
		Register selling products	20	10
		Website appearance configuration	30	30
		Database operations	30	50
	Survey Mode	Search products	10	10
		Purchase products	0	0
		Register selling products	0	0
		Website appearance configuration	0	0
		Database operations	90	90

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.[20] An example output for `(user group, system mode)`=`(Administrator, Maintenance Mode)` in Table 2.1 is shown in Table 2.2. A larger priority has a higher priority.

Algorithm 1 Derive a prioritized list of usages (test cases) with usage statistics

```
Input: Usage statistics:
        $Pr$:(User group)×(System mode)×(Usage)→(Occurrence probability)
        $Cr$:(User group)×(System mode)×(Usage)→(Criticality)
Output: A list of pairs (usage, priority) in the priority descending order:
        $L$:(Usage)×(Priority)
for each pair (user group $g_i$, system mode $m_j$) do
    for each usage $u_k$ of ($g_i$, $m_j$) do
        Calculate the priority $p_k$ of $u_k$. $p_k = 100 \times \frac{Pr(g_i,m_j,u_k)}{\sum_{\forall u_k \text{of}(g_i,m_j)} Pr} \times \frac{Cr(g_i,m_j,u_k)}{\sum_{\forall u_k \text{of}(g_i,m_j)} Cr}$
        Insert $(u_k, p_k)$ to $L$ between the elements $l_1 = (u_1, p_1)$ and $l_2 = (u_2, p_2)$
        where $p_1 \geq p_k \geq p_2$.
    end for
end for
```

Table 2.2 Example list of pairs `(usage, priority)`

Usage	**Priority (1..100)**
Database operations	15
Website appearance configuration	9
Register selling products	2
Search products	5
Purchase products	5

[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.[1] 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.

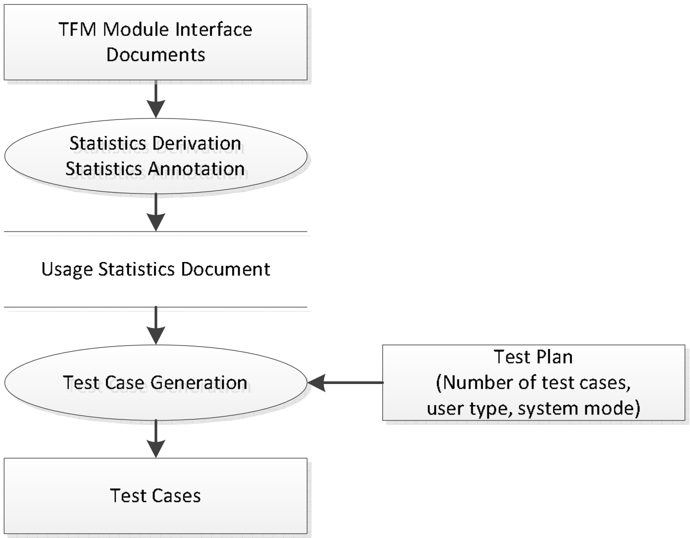

TFM Module Interface Documents
Statistics Derivation Statistics Annotation
Usage Statistics Document
Test Case Generation
Test Plan (Number of test cases, user type, system mode)
Test Cases

Figure 3.1 Overview of Test Case Generation Process

[1] Note that Figure 3.2 omits some software design phases to give a rough view.

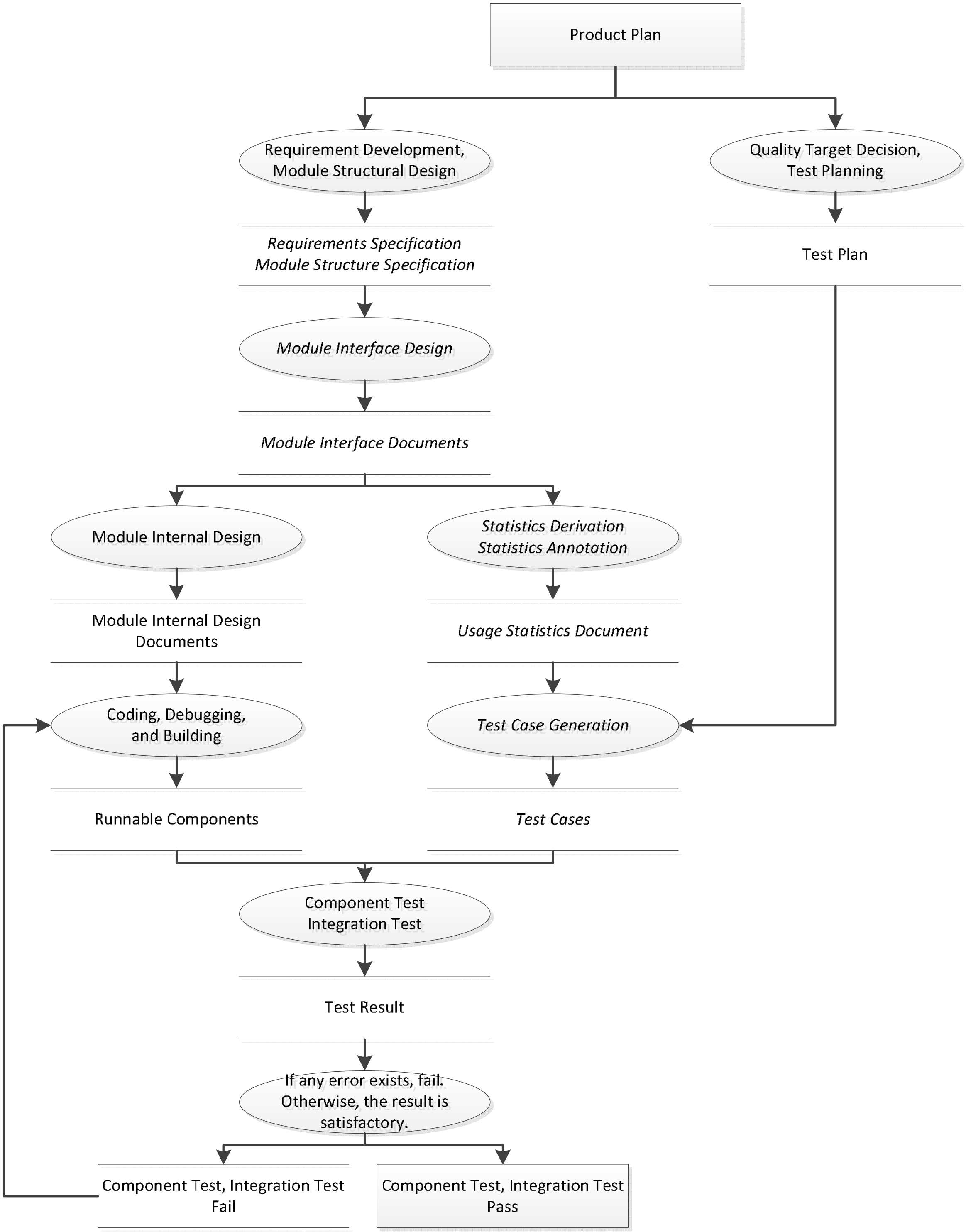

Product Plan
Requirement Development, Module Structural Design
Quality Target Decision, Test Planning
Requirements Specification
Module Structure Specification
Test Plan
Module Interface Design
Module Interface Documents
Module Internal Design
Statistics Derivation
Statistics Annotation
Module Internal Design Documents
Usage Statistics Document
Coding, Debugging, and Building
Test Case Generation
Runnable Components
Test Cases
Component Test
Integration Test
Test Result
If any error exists, fail. Otherwise, the result is satisfactory.
Component Test, Integration Test Fail
Component Test, Integration Test Pass

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 [20].[3]

In TFM, a conditional $\{$`(trace predicates)` $\rightarrow$ `(output value definition)`$\}$ 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 Γ is **consistent** if there is no formula θ such that $\Gamma \vdash \theta$ and $\Gamma \vdash \neg\theta$ [20]. [4]

> **Definition:** A trace specification with its assertions Γ 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 $(V(T) \equiv Undefined)$.[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 [15].

[4] An argument $< \Gamma, \phi >$ where Γ is premises, and ϕ is a conclusion is **derivable** if there is a deduction from some of its premises to its conclusion, which we denote as $\Gamma \vdash \phi$. [38]

[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.

$$\begin{aligned}
\mathtt{T} &= (\mathtt{PGM} : \mathtt{ModuleA\_constructor}, \mathtt{object}' < \mathtt{ModuleA} > : \mathtt{objectA1}).\\
&\quad (\mathtt{PGM} : \mathtt{ModuleA\_constructor}, \mathtt{object}' < \mathtt{ModuleA} > : \mathtt{objectA2}).\\
&\quad (\mathtt{PGM} : \mathtt{ModuleA\_printHelloWorld},' \mathtt{object} < \mathtt{ModuleA} > : \mathtt{objectA1}).\\
&\quad (\mathtt{PGM} : \mathtt{ModuleA\_printHelloWorld},' \mathtt{object} < \mathtt{ModuleA} > : \mathtt{objectA2}).\\
&\quad (\mathtt{PGM} : \mathtt{ModuleA\_destructor},' \mathtt{object} < \mathtt{ModuleA} > : \mathtt{objectA1}).\\
&\quad (\mathtt{PGM} : \mathtt{ModuleA\_destructor},' \mathtt{object} < \mathtt{ModuleA} > : \mathtt{objectA2})
\end{aligned}$$

[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 $T._{<objectname>}$, where $< objectname >$ is the name of the object and $T._{<self>}$ is the trace of the primary object[33].

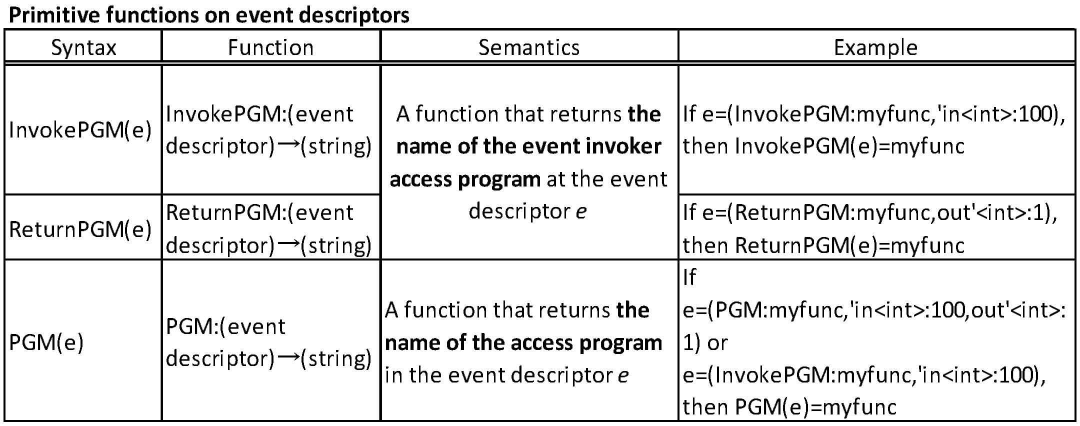
3.1.2.2 Nesting Event Trace

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'`."

$$\begin{aligned} \mathtt{T} \;=\; & (\mathtt{InvokePGM}:\mathtt{ModuleA\_main},'\mathtt{in} < \mathtt{Boolean} > : \mathtt{True}). \\ & (\mathtt{PGM}:\mathtt{ModuleB\_func1},\mathtt{out}' < \mathtt{Boolean} > : \mathtt{True}). \\ & (\mathtt{PGM}:\mathtt{ModuleB\_func2},\mathtt{out}' < \mathtt{int} > : \mathtt{100}). \\ & (\mathtt{ReturnPGM}:\mathtt{ModuleA\_main},\mathtt{result}' < \mathtt{Boolean} > : \mathtt{True}) \end{aligned}$$



Primitive functions on event descriptors

Syntax	Function	Semantics	Example
InvokePGM(e)	InvokePGM:(event descriptor)→(string)	A function that returns **the name of the event invoker access program** at the event descriptor *e*	If e=(InvokePGM:myfunc,'in<int>:100), then InvokePGM(e)=myfunc
ReturnPGM(e)	ReturnPGM:(event descriptor)→(string)		If e=(ReturnPGM:myfunc,out'<int>:1), then ReturnPGM(e)=myfunc
PGM(e)	PGM:(event descriptor)→(string)	A function that returns **the name of the access program** in the event descriptor *e*	If e=(PGM:myfunc,'in<int>:100,out'<int>:1) or e=(InvokePGM:myfunc,'in<int>:100), then PGM(e)=myfunc

Figure 3.3 Primitive Functions on Event Descriptors

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.

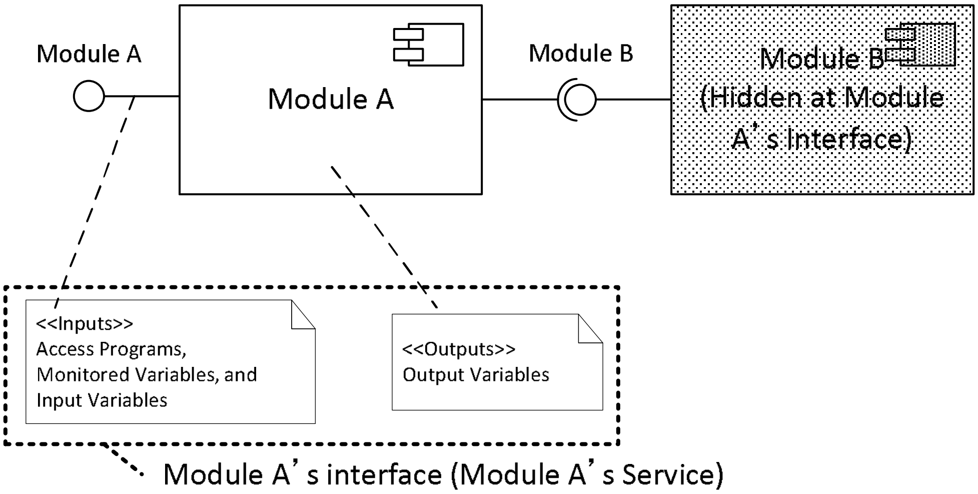


Figure 3.4 Upper-Face Only Module Interface

The upper-face only module interface specification delivers many benefits to software design, coding, and testing[8][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.[9] When we generate black-box test cases for integration testing, the lower-face specifi-

[8] Users can see what a module does without referring to its design details. Architects do not need to make premature decisions on design details when they design architecture. Implementers can choose the most effective implementation after careful considerations of external behaviors. Designers can distinguish what they can change, e.g. replaceable algorithm, from what they cannot, i.e. required services. Testers can design black box test cases.

[9] 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`."

operating system that will be a part of the design decision of whether they use Module A or not. However, the upper-face does not specify them. Designers and implementers must use modules as designed by architects, although the upper-face tells nothing about the used modules, and they must analyze the usage of shared resources such as memory and timing, none of which is indicated by the upper-face interface.

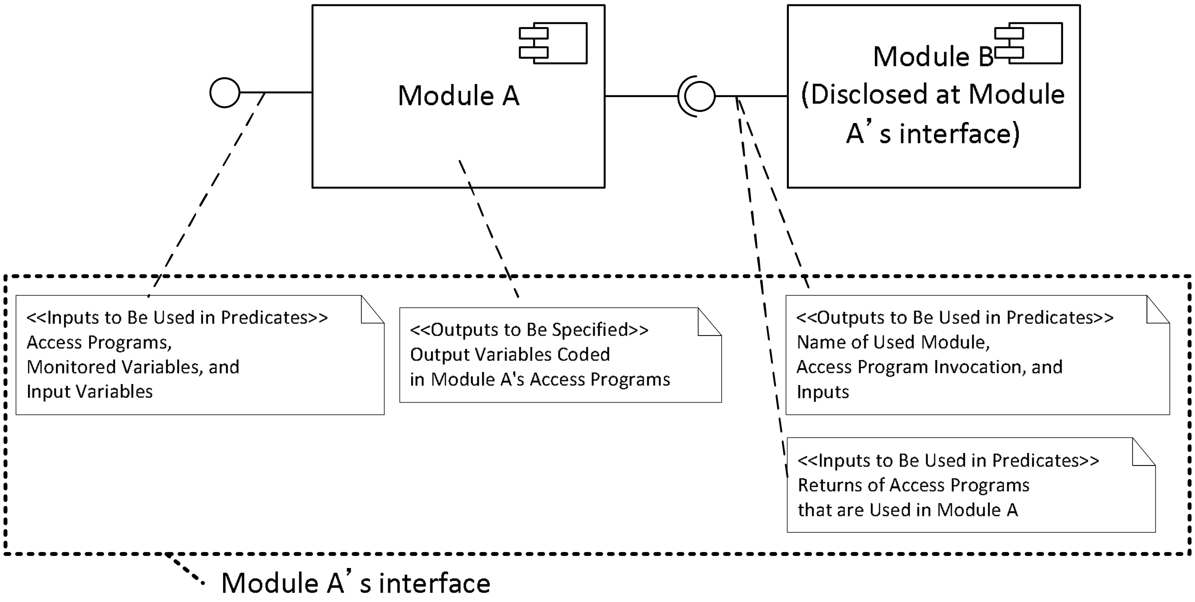
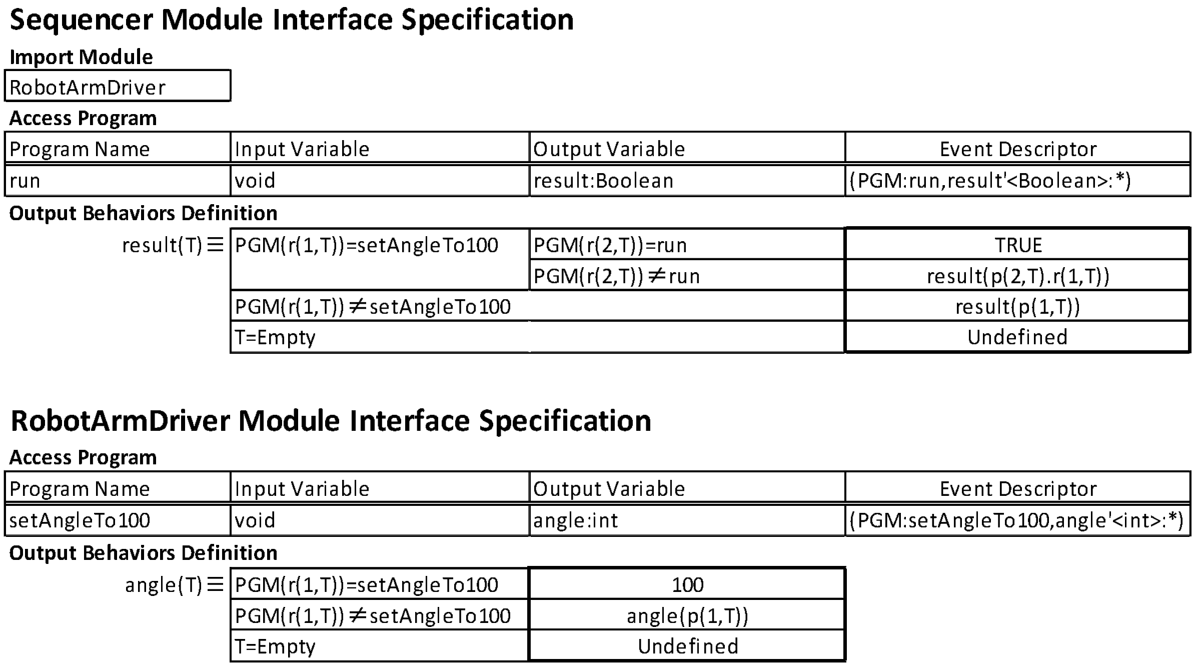

Module A
Module B
(Disclosed at Module A's interface)
<<Inputs to Be Used in Predicates>>
Access Programs,
Monitored Variables, and
Input Variables
<<Outputs to Be Specified>>
Output Variables Coded
in Module A's Access Programs
<<Outputs to Be Used in Predicates>>
Name of Used Module,
Access Program Invocation, and
Inputs
<<Inputs to Be Used in Predicates>>
Returns of Access Programs
that are Used in Module A
Module A's interface

Figure 3.5 Module Interface Specification



Sequencer Module Interface Specification

Import Module

RobotArmDriver

Access Program

Program Name	Input Variable	Output Variable	Event Descriptor
run	void	result:Boolean	(PGM:run,result'<Boolean>:*)

Output Behaviors Definition

result(T)≡	PGM(r(1,T))=setAngleTo100	PGM(r(2,T))=run	TRUE
		PGM(r(2,T))≠run	result(p(2,T).r(1,T))
	PGM(r(1,T))≠setAngleTo100		result(p(1,T))
	T=Empty		Undefined

RobotArmDriver Module Interface Specification

Access Program

Program Name	Input Variable	Output Variable	Event Descriptor
setAngleTo100	void	angle:int	(PGM:setAngleTo100,angle'<int>:*)

Output Behaviors Definition

angle(T)≡	PGM(r(1,T))=setAngleTo100	100
	PGM(r(1,T))≠setAngleTo100	angle(p(1,T))
	T=Empty	Undefined

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.[10] 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:

$$angle(T) \equiv \quad 100 \quad \text{if } PGM(r(1,T)) = setAngleTo100 \tag{3.1}$$

$$angle(T) \equiv \quad angle(p(1,T)) \quad \text{if } PGM(r(1,T)) \neq setAngleTo100 \tag{3.2}$$

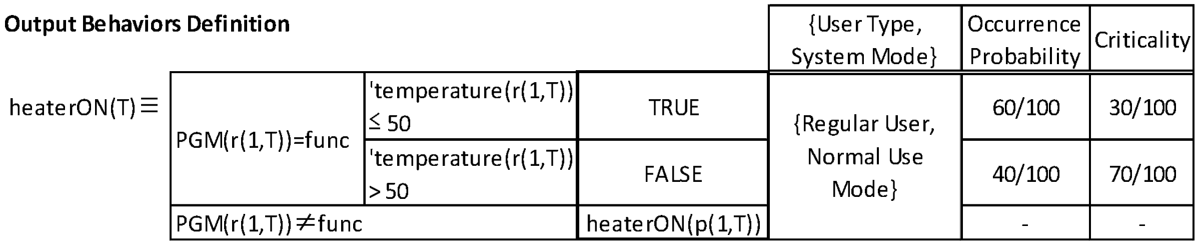
$$angle(T) \equiv \quad Undefined \quad \text{if } T = Empty \tag{3.3}$$

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 `'temperature` is less than or equal to 50F, then the access program `func` turns on the heater switch, which is `heaterON` ≡ `true`. This example says that the occurrence probability of the class that has "`heaterON'<boolean>:true`" is "`60/100`" and the one that has "`heaterON'<boolean>:false`" is "`40/100`." The criticality of the class that has "`heaterON'<boolean>:true`" is "`30/100`" and the one that has "`heaterON'<boolean>: false`" is "`70/100`". These values can be used to calculate a priority of each equivalence class of traces.



Output Behaviors Definition				{User Type, System Mode}	Occurrence Probability	Criticality
heaterON(T) ≡	PGM(r(1,T))=func	'temperature(r(1,T)) ≤ 50	TRUE	{Regular User, Normal Use Mode}	60/100	30/100
		'temperature(r(1,T)) > 50	FALSE		40/100	70/100
	PGM(r(1,T))≠func		heaterON(p(1,T))		-	-

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:

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

$$\mathtt{T.(PGM : func,'in < int > : a, out' < int > : a)}$$

"`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:

$$\mathtt{T.(PGM : funcA, result' < Boolean > : UNEVAL)} \rightarrow$$

$$\mathtt{T.(InvokePGM : funcA).}$$

$$\mathtt{(PGM : funcB, UNEVAL).}$$

$$\mathtt{(ReturnPGM : funcA, result' < Boolean > : True)}$$

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:

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

3.3.2 Monitored Variable Constraints

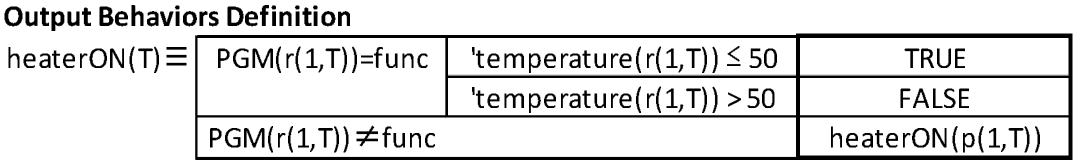
Inputs can be categorized into the following two variables[11]:

- 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` $\leq$ `50`" or "`'temperature` $>$ `50`."



Output Behaviors Definition

heaterON(T) ≡

PGM(r(1,T))=func	'temperature(r(1,T)) ≤ 50	TRUE
	'temperature(r(1,T)) > 50	FALSE
PGM(r(1,T)) ≠ func		heaterON(p(1,T))

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)

Step4. 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)

Step5. 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.

Step6. Generate monitored variable values using monitored variable constraints and a certain probability distribution. (Stage i+3 in appendix D) [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

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

```
**Input** A trace: $T$ and an event index of $T$ to evaluate: $srcIndex$
**Input** Usage statistics specifications: $S$ and Trace max lenth: $l$
**Input** A list of monitored variable constraint expressions: $CONS$
**Input** A key-value map of meta variable and value: $VARS$
/*Part1. Append events produced by the event at $srcIndex$ in $T$*/
**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
Randomly select a equivalence class $ec$ from the all candidate equivalence classes using each occurrence probability and criticality
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.
Add the monitored variable constraints of $ec$ to $CONS$
**if** the index exceeds the upper bound length $srcIndex > l$ **then**
    **Output** $T, CONS, VARS$ **and return**
**end if**
/*Part2. Append events produced by all nested events*/
Increment $srcNextIndex = srcIndex + 1$
Initialize $lastIndex = srcIndex$
**for** all of the inserted nested events **do**
    **CALL** EventEvaluation **with** $T_{test}, lastIndex, S, l, CONS, VARS$ **and**
    **UPDATE** $T, lastIndex, CONS, VARS$
    **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**
    **OBTAIN** all candidate equivalence classes that the sub-list of $T$ between 0 and $lastIndex$ will be classified with any future events
    Randomly select an equivalence class $ec$ from the all candidate equivalence classes using each occurrence probability and criticality
    Increment $srcNextIndex = srcNextIndex + 1$
    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 $lastIndex$.
**end for**
**if** ReturnPGM(event at $lastIndex$ in $T \neq null$ **then**
    Increment $lastIndex = lastIndex + 1$
**end if**
Put $ec$'s output value definition to $VARS$
**Output** $T, lastIndex, CONS, VARS$ **and return**
```

Algorithm 4 `QueryPossiblySatisfiableEquivalenceClasses`

```
Input A trace: $T$
Input Name of a program to invoke: $f$ and Name of its module: $m$
Input Usage statistics specifications: $S$
Input A list of monitored variable constraint expressions: $CONS$
Input A key-value map of meta variable and value: $VARS$
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 $f$
for each equation $eq$ do
  Initialize $T_{temp} = T$
  REEXAMINE_TRANSFORMED:
  Extract the preconditions of the last event in $T_{temp}$ from $eq$'s predicate
  Check whether $T$ with $CONS$ and $VARS$ satisfies the preconditions with a predicate
    solver
  if satisfied then
    if the output value of $eq$ is a trace transformation function then
      Transform $T_{temp}$ with the trace transformation function in the output value definition
        of $eq$
      GO TO REEXAMINE_TRANSFORMED to reexamine the transformed trace
        $T_{temp}$.
    else
      Add $eq$ to the set of equations $SATEQ$
    end if
  end if
end for
Output the equivalence classes that correspond to $SATEQ$ and return
```

CHAPTER 4. EVALUATION

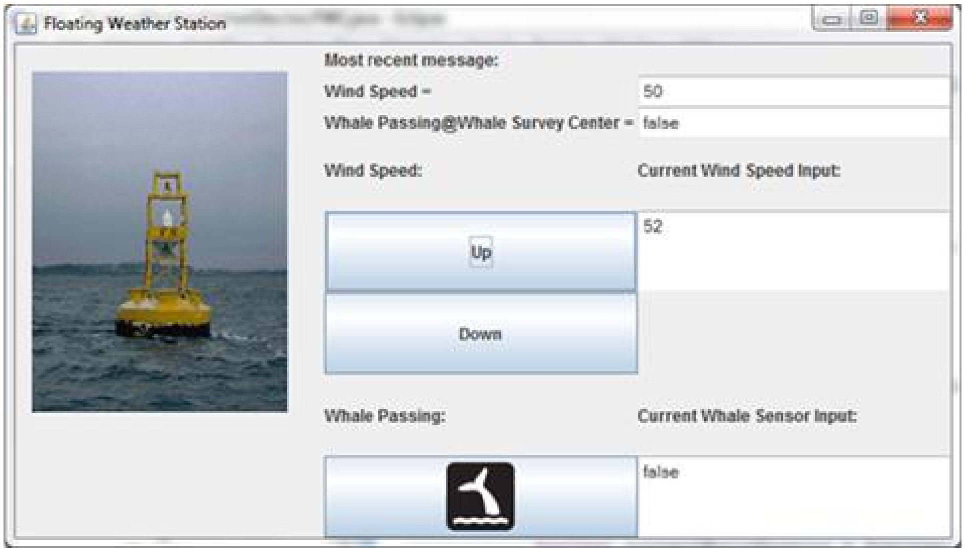
4.1 Feasibility of Test Case Generation - Example Application: Floating Weather Station

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



Most recent message:

Wind Speed = 50

Whale Passing@Whale Survey Center = false

Wind Speed:

Current Wind Speed Input: 52

Whale Passing:

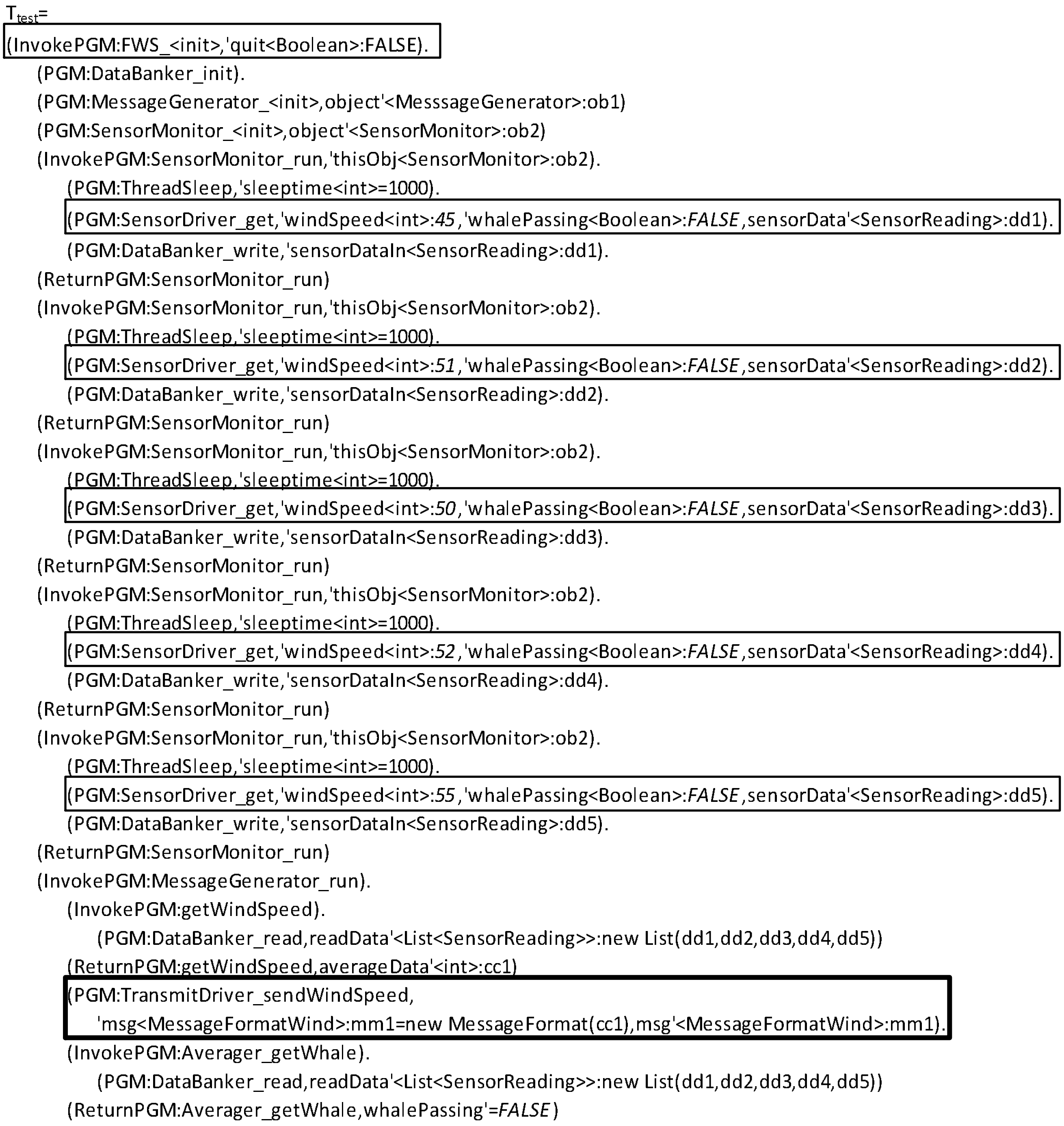
Current Whale Sensor Input: false

Figure 4.1 FWS simulator GUI

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_init` with a maximum testing trace length `37`. The service of `FWS_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 `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.



```
Ttest=
(InvokePGM:FWS_<init>,'quit<Boolean>:FALSE).
    (PGM:DataBanker_init).
    (PGM:MessageGenerator_<init>,object'<MesssageGenerator>:ob1)
    (PGM:SensorMonitor_<init>,object'<SensorMonitor>:ob2)
    (InvokePGM:SensorMonitor_run,'thisObj<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).
    (ReturnPGM:SensorMonitor_run)
    (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
        (PGM:ThreadSleep,'sleeptime<int>=1000).
        (PGM:SensorDriver_get,'windSpeed<int>:51,'whalePassing<Boolean>:FALSE,sensorData'<SensorReading>:dd2).
        (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd2).
    (ReturnPGM:SensorMonitor_run)
    (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
        (PGM:ThreadSleep,'sleeptime<int>=1000).
        (PGM:SensorDriver_get,'windSpeed<int>:50,'whalePassing<Boolean>:FALSE,sensorData'<SensorReading>:dd3).
        (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd3).
    (ReturnPGM:SensorMonitor_run)
    (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
        (PGM:ThreadSleep,'sleeptime<int>=1000).
        (PGM:SensorDriver_get,'windSpeed<int>:52,'whalePassing<Boolean>:FALSE,sensorData'<SensorReading>:dd4).
        (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd4).
    (ReturnPGM:SensorMonitor_run)
    (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
        (PGM:ThreadSleep,'sleeptime<int>=1000).
        (PGM:SensorDriver_get,'windSpeed<int>:55,'whalePassing<Boolean>:FALSE,sensorData'<SensorReading>:dd5).
        (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd5).
    (ReturnPGM:SensorMonitor_run)
    (InvokePGM:MessageGenerator_run).
        (InvokePGM:getWindSpeed).
            (PGM:DataBanker_read,readData'<List<SensorReading>>:new List(dd1,dd2,dd3,dd4,dd5))
        (ReturnPGM:getWindSpeed,averageData'<int>:cc1)
        (PGM:TransmitDriver_sendWindSpeed,
            'msg<MessageFormatWind>:mm1=new MessageFormat(cc1),msg'<MessageFormatWind>:mm1).
        (InvokePGM:Averager_getWhale).
            (PGM:DataBanker_read,readData'<List<SensorReading>>:new List(dd1,dd2,dd3,dd4,dd5))
        (ReturnPGM:Averager_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 $E = O(l_{max}^2 s_{max}b_{max})$, and the running time of algorithm 2 is $O(E+|V|) = O(l_{max}^2 s_{max}b_{max} + |V|)$, 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] **Oracle** is any means that provides information about the (correct) expected behavior of a component [13].

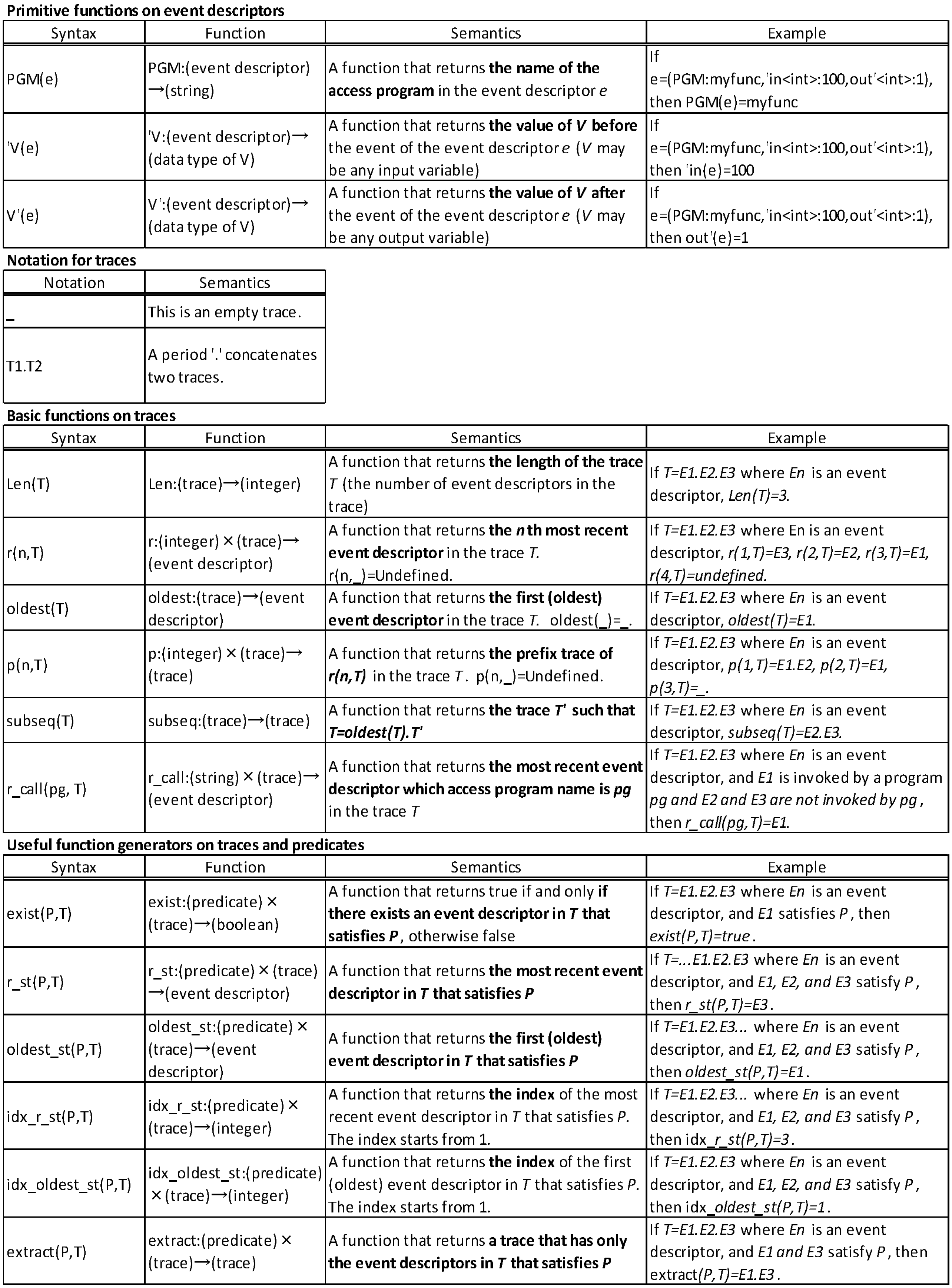
none of them generated the oracle or test cases for multiple modules, i.e. component testing and integration testing.

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

Syntax	Function	Semantics	Example
PGM(e)	PGM:(event descriptor)→(string)	A function that returns **the name of the access program** in the event descriptor *e*	If e=(PGM:myfunc,'in<int>:100,out'<int>:1), then PGM(e)=myfunc
'V(e)	'V:(event descriptor)→(data type of V)	A function that returns **the value of *V* before** the event of the event descriptor *e* (*V* may be any input variable)	If e=(PGM:myfunc,'in<int>:100,out'<int>:1), then 'in(e)=100
V'(e)	V':(event descriptor)→(data type of V)	A function that returns **the value of *V* after** the event of the event descriptor *e* (*V* may be any output variable)	If e=(PGM:myfunc,'in<int>:100,out'<int>:1), then out'(e)=1

Notation for traces

Notation	Semantics
_	This is an empty trace.
T1.T2	A period '.' concatenates two traces.

Basic functions on traces

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

Useful function generators on traces and predicates

Syntax	Function	Semantics	Example
exist(P,T)	exist:(predicate) × (trace)→(boolean)	A function that returns true if and only **if there exists an event descriptor in *T* that satisfies *P*** , otherwise false	If *T=E1.E2.E3* where *En* is an event descriptor, and *E1* satisfies *P* , then *exist(P,T)=true* .
r_st(P,T)	r_st:(predicate) × (trace)→(event descriptor)	A function that returns **the most recent event descriptor in *T* that satisfies *P***	If *T=...E1.E2.E3* where *En* is an event descriptor, and *E1, E2, and E3* satisfy *P* , then *r_st(P,T)=E3* .
oldest_st(P,T)	oldest_st:(predicate) × (trace)→(event descriptor)	A function that returns **the first (oldest) event descriptor in *T* that satisfies *P***	If *T=E1.E2.E3...* where *En* is an event descriptor, and *E1, E2, and E3* satisfy *P* , then *oldest_st(P,T)=E1* .
idx_r_st(P,T)	idx_r_st:(predicate) × (trace)→(integer)	A function that returns **the index** of the most recent event descriptor in *T* that satisfies *P.* The index starts from 1.	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* .
idx_oldest_st(P,T)	idx_oldest_st:(predicate) × (trace)→(integer)	A function that returns **the index** of the first (oldest) event descriptor in *T* that satisfies *P.* The index starts from 1.	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* .
extract(P,T)	extract:(predicate) × (trace)→(trace)	A function that returns **a trace that has only the event descriptors in *T* that satisfies *P***	If *T=E1.E2.E3* where *En* is an event descriptor, and *E1 and E3* satisfy *P* , then extract*(P,T)=E1.E3* .

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

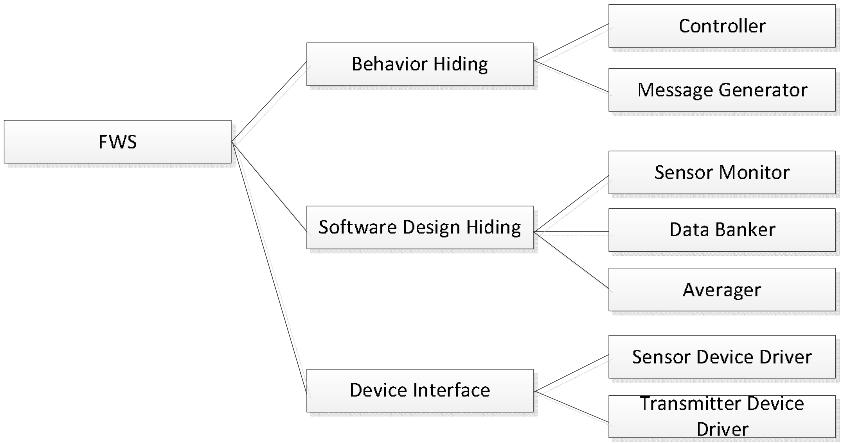


Figure B.1 FWS Module Hierarchical Structure

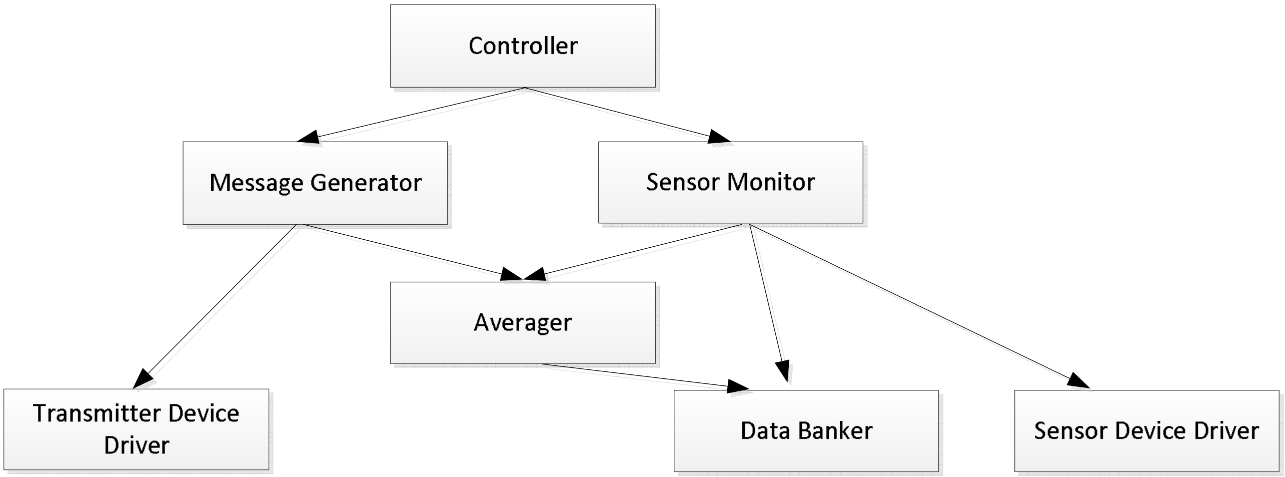


Figure B.2 FWS Uses Relation

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 Variable Constraints*	*Executed Count*	*Total Sample*	*Occurrence Probability*
sensorData(T)≡	Pre(1,T,SensorDriver_get)={True,i}	new SensorReading('windSpeed(r(i,T)), 'whalePassing(r(i,T)))	T.(PGM:SensorDriver_get,'windSpeed<int>:UNEVAL,'whalePassing<Boolean>:UNEVAL, sensorData'<SensorReading>:UNEVAL)→ T.(PGM:SensorDriver_get,'windSpeed<int>:aa,'whalePassing<Boolean>:bb, sensorData'<SensorReading>:new SensorReading(aa,bb))	INTMIN ≤ aa ≤ INTMAX, bb = TRUE v FALSE	*114*	*114*	*100*
	⋁j { ¬[⋀i (Predicate (i,j))] }	Undefined	Undefined	-	*0*	*0*	0

Figure C.1 SensorDriver module Before Transpose

Module Index

Behavior Hiding	Controller
	MessageGenerator
Software Design Hiding	SensorMonitor
	Averager
	DataBanker
Device Interface	SensorDriver
	TransmitDriver

Auxiliary Functions

Trace Predicates	r(i,T)=evt	r(i,T)≠evt	r(i,T)=Empty
Pre(i,T,evt)≡	{TRUE,i}	Pre(i+1,T,evt)	{FALSE,i}

Trace Predicates	r(i,T)=evt	r(i,T)≠evt	r(i,T)=Empty
Post(i,T,evt)≡	{TRUE,i}	Post(i-1, T,evt)	{FALSE,i}

suffix(i,T)≡	subseq^i(T)

Trace Predicates	*'quit(r([idx_r_st(PGM(r(1,T))=(MessageGenerator_run)], T))=FALSE*	*'quit(r([idx_r_st(PGM(r(1,T))=(MessageGenerator_run)], T))=TRUE*
T.loopEvents()≡	T. (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,*). (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,*). (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,*). (PGM:MessageGenerator_run,'obj<MessageGenerator>:ob1,*). loopEvents()	T

Data Type

Type Name	*Attributes*
MessageFormatWind	windSpeed:int
MessageFormatWhale	whalePassing:Boolean
SensorReading	windSpeed:int
	whalePassing:Boolean

Controller Module Interface Specification

Trace Event Syntax

Program Name	*Input Variable*	*Output Variable*	*Event Descriptor*
FWS_<init>	quit:Boolean	NONE	(PGM:FWS_<init>, 'quit<Boolean>:*)

Trace Semantics

Trace Predicates	*idx_oldest_st(PGM(r(1,T))=FWS_<init>, T)=i0*	∨j { ¬[∧i (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.
	[idx_oldest_st(PGM(r(1,T))=DataBanker_init,T)=i1] ^ [i0<i1]	
	[idx_oldest_st(PGM(r(1,T))=MessageGenerator_<init>,T)=i2] ^ [i1<i2]	
	[idx_oldest_st(PGM(r(1,T))=SensorMonitor_<init>,T)=i3] ^ [i2<i3]	
	suffix(i3,T)=loopEvents()	
	Len(extract ([(PGM=SensorMonitor_run)^('quit=FALSE)] , T)) = n	
Output Behavior Definition	NONE(T)≡NONE	NONE(T)≡ Undefined
Trace Postcondition	T.(PGM:FWS_<init>,'quit<Boolean>:qq)→ T.(InvokePGM:FWS_<init>,'quit<Boolean>:qq). (PGM:DataBanker_init). (PGM:MessageGenerator_<init>,object'<MessageGenerator>:ob1,UNEVAL). (PGM:SensorMonitor_<init>,object'<SensorMonitor>:ob2,UNEVAL). [(PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL). (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL). (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL). (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL). (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL). (PGM:MessageGenerator_run,'obj<MessageGenerator>:ob1,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

Program Name	*Input Event Descriptor*	*Output Event Descriptor*	*Event Descriptor*
<init>	NONE	object:MessageGenerator	(PGM:MessageGenerator_<init>, object'<MessageGenerator>:*)
run	thisObj:MessageGenerator	NONE	(PGM:MessageGenerator_run, 'thisObj<SensorMonitor>:*)

Trace Semantics

Trace Predicates	Pre(1,T,MessageGenerator_<init>)={TRUE,i0}		Pre(1,T,MessageGenerator_<init>)={TRUE,i1}	⋁j { ¬[⋀i (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.
	Post(i0,T,MessageGenerator_run)={TRUE,i1} ^			
	Post(i1,T,Averager_getWindSpeed)={TRUE,i2}			
	Post(i2,T,TransmitDriver_send)={TRUE,i3} ^			
	Post(i3,T,Averager_getWhale)={TRUE,i4}			

	[Pre(i0,T,Averager_getWhale)={TRUE,i5}] ^ [whalePassing'(r(i4,T))=whalePassing'(r(i5,T))]	{ [
		[Pre(i0,T,Averager_getWhale)={TRUE,i5}] ^
		[whalePassing'(r(i4,T))≠ whalePassing'(r(i5,T))]
		} v {
		[Pre(i0,T,Averager_getWhale)={FALSE,i5}]
		] } ^
		{
		[Post(i4,T,TransmitDriver_sendToWhaleSurveyCenter)={TRUE,i5}] ^ ['msg(r(i5,T))=new MessageFormatWhale(TRUE)]
		}		

Output Behavior Definition	NONE(T)=NONE	NONE(T)=NONE	object(T)=new MessageGenerator	Undefined
Trace Postcondition	T.(PGM:MessageGenerator_run)→ T.(InvokePGM:MessageGenerator_run).(PGM:Averager_getWindSpeed,averageData'<int>:ag,UNEVAL).(PGM:TransmitDriver_sendWindSpeed.'msg<MessageFormatWind>:new MessageFormat(ag),UNEVAL).(PGM:Averager_getWhale,whalePassing'<boolean>:FALSE,UNEVAL).(ReturnPGM:MessageGenerator_run)	T.(PGM:MessageGenerator_run)→ T.(InvokePGM:MessageGenerator_run).(PGM:Averager_getWindSpeed,averageData'<int>:ag,UNEVAL).(PGM:TransmitDriver_send.'msg<MessageFormatWind>:new MessageFormat(ag),UNEVAL).(PGM:Averager_getWhale,whalePassing'<boolean>:TRUE,UNEVAL).(PGM:TransmitDriver_sendWhale,'msg<MessageFormatWhale>:new MessageFormat(TRUE),UNEVAL).(ReturnPGM:MessageGenerator_run)	T.(PGM:MessageGenerator_<init>,object'<MessageGenerator>:UNEVAL)→ T.(PGM:MessageGenerator_<init>,object'<MesssageGenerator>:new MessageGenerator)	Undefined
Monitored Variable Constraints	-	-	-	-
Executed Count	22	2	1	0
Total Sample	24		1	0
Occurrence Probability	91.667	8.333	100	0

SensorMonitor Module Interface Specification

Trace Event Syntax

Program Name	*Input Variable*	*Output Variable*	*Event Descriptor*
<init>	NONE	object:SensorMonitor	(PGM:SensorMonitor_<init>,object'<SensorMonitor>:*)
run	thisObj:SensorMonitor	NONE	(PGM:SensorMonitor_run,'thisObj<SensorMonitor>:*)

Trace Semantics

Trace Predicates	Pre(1,T,SensorMonitor_<init>)={True,i0} Pre(i0,T,run)={True,i1} ^ ['thisObj(r(i1,T))=object'(r(i0,T))] Post(i1,T,Thread_Sleep)={TRUE,i2} ^ ['sleeptime(r(i2,T))=1000] Post(i2,T,SensorDriver_get)={TRUE,i3} Post(i3,T,DataBanker_write)={TRUE,i4} ^ ['sensorDataIn(r(i4,T))=sensorData'(r(i3,T))]	Pre(1,T,SensorMonitor_<init>)={True,i}	⋁j { ¬[⋀i (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.
Output Behavior Definition	NONE(T)=NONE	object(T)=new SensorMonitor	Undefined
Trace Postcondition	T.(PGM:SensorMonitor_run,'thisObj<SensorMonitor>:oo)→ T. (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:oo). (PGM:ThreadSleep,'sleeptime<int>=1000). (PGM:SensorDriver_get,'windSpeed<int>:UNEVAL,'whalePassing<Boolean>:UNEVAL,sensorData'<SensorReading>:dd). (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd). (ReturnPGM:SensorMonitor_run)	T.(PGM:SensorMonitor_<init>,object'<SensorMonitor>:UNEVAL) → T.(PGM:SensorMonitor_<init>,object'<SensorMonitor>:new SensorMonitor)	Undefined
Monitored Variable Constraints	-	-	-
Executed Count	114	1	0
Total Sample	114	1	0
Occurrence Probability	100	100	0

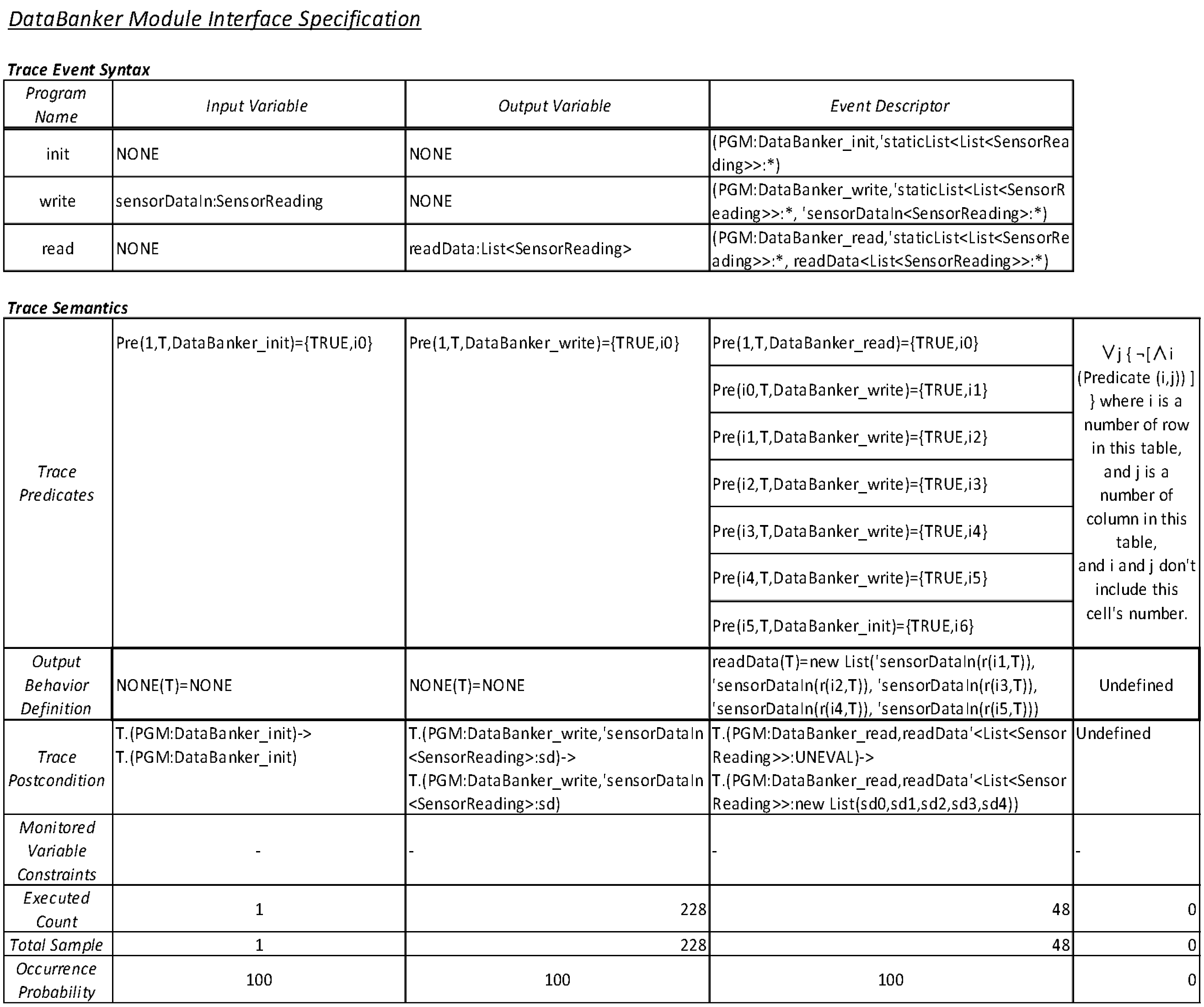
Averager Module Interface Specification

Trace Event Syntax

Program Name	*Input Variable*	*Output Variable*	*Event Descriptor*
getWindSpeed	NONE	averageData:int	(PGM:Averager_getWindSpeed,averageData'<int>:*)
getWhale	NONE	whalePassing:boolean	(PGM:Averager_getWhale,whalePassing'<Boolean>:*)

Trace Semantics

Trace Predicates	Pre(1,T,Averager_getWindSpeed)={TRUE,i0}	Pre(1,T,Averager_getWhale)={TRUE,i0}		⋁j { ¬[⋀i (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.
	Post(i0,T,DataBanker_read)={TRUE,i1}	exist({[Pre(i0,T,DataBanker_write)={TRUE,i1}] ^ ['sensorDataIn(r(i1,T)) = {*,TRUE}] } v {[Pre(i1,T,DataBanker_write)={TRUE,i2}] ^ ['sensorDataIn(r(i2,T)) = {*,TRUE}] } v {[Pre(i2,T,DataBanker_write)={TRUE,i3}] ^ ['sensorDataIn(r(i3,T)) = {*,TRUE}] } v {[Pre(i3,T,DataBanker_write)={TRUE,i4}] ^ ['sensorDataIn(r(i4,T)) = {*,TRUE}] } v {[Pre(i4,T,DataBanker_write)={TRUE,i5}] ^ ['sensorDataIn(r(i5,T)) = {*,TRUE}] } ,T) = TRUE	exist({[Pre(i0,T,DataBanker_write)={TRUE,i1}] ^ ['sensorDataIn(r(i1,T)) = {*,FALSE}] } ^ {[Pre(i1,T,DataBanker_write)={TRUE,i2}] ^ ['sensorDataIn(r(i2,T)) = {*,FALSE}] } ^ {[Pre(i2,T,DataBanker_write)={TRUE,i3}] ^ ['sensorDataIn(r(i3,T)) = {*,FALSE}] } ^ {[Pre(i3,T,DataBanker_write)={TRUE,i4}] ^ ['sensorDataIn(r(i4,T)) = {*,FALSE}] } ^ {[Pre(i4,T,DataBanker_write)={TRUE,i5}] ^ ['sensorDataIn(r(i5,T)) = {*,FALSE}] } ,T) = TRUE	
		Post(i0,T,DataBanker_read)={TRUE,i}	Post(i0,T,DataBanker_read)={TRUE,i}	
Output Behavior Definition	averageData(T)= Σj ((readData'(r(i1,T)).listElement(j)).value / readData'(r(i1,T)).length)	whalePassing(T)= True	whalePassing(T)= False	Undefined
Trace Postcondition	T.(PGM:Averager_getWindSpeed,averageData'<int>:UNEVAL)-> T. (InvokePGM:Averager_getWindSpeed). (PGM:DataBanker_read,readData'<List<int>>:rd,UNEVAL). (ReturnPGM:Averager_getWindSpeed,averageData'<int>:Σi (rd.listElement(i)).value / rd.length))	T.(PGM:Averager_getWhale,whalePassing'<Boolean>:UNEVAL)-> T. (InvokePGM:Averager_getWhale). (PGM:DataBanker_read,readData'<List<int>>:rd,UNEVAL). (ReturnPGM:Averager_getWhale,whalePassing'=True)	T.(PGM:Averager_getWhale,whalePassing'<Boolean>:UNEVAL)-> T. (InvokePGM:Averager_getWhale). (PGM:DataBanker_read,readData'<List<int>>:rd,UNEVAL). (ReturnPGM:Averager_getWhale,whalePassing'=False)	Undefined
Monitored Variable Constraints	-	[(rd.listItemAt[0]).whalePassing = True] v [(rd.listItemAt[1]).whalePassing = True] v [(rd.listItemAt[2]).whalePassing = True] v [(rd.listItemAt[3]).whalePassing = True] v [(rd.listItemAt[4]).whalePassing = True]	[(rd.listItemAt[0]).whalePassing = False] ^ [(rd.listItemAt[1]).whalePassing = False] ^ [(rd.listItemAt[2]).whalePassing = False] ^ [(rd.listItemAt[3]).whalePassing = False] ^ [(rd.listItemAt[4]).whalePassing = False]	-
Executed Count	24	3	21	0
Total Sample	24	24		0
Occurrence Probability	100	12.5	87.5	0



DataBanker Module Interface Specification

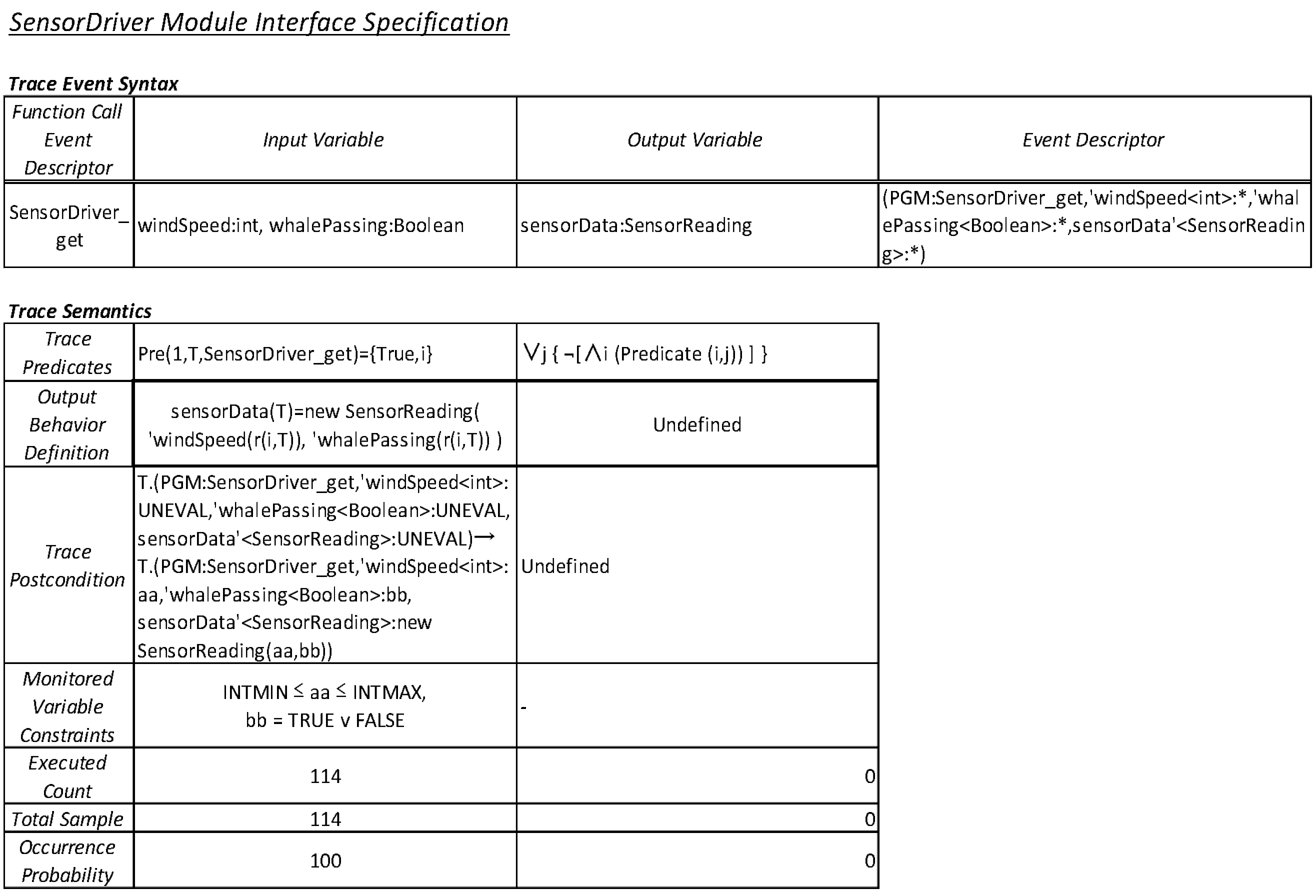
Trace Event Syntax

Program Name	*Input Variable*	*Output Variable*	*Event Descriptor*
init	NONE	NONE	(PGM:DataBanker_init,'staticList<List<SensorReading>>:*)
write	sensorDataIn:SensorReading	NONE	(PGM:DataBanker_write,'staticList<List<SensorReading>>:*, 'sensorDataIn<SensorReading>:*)
read	NONE	readData:List<SensorReading>	(PGM:DataBanker_read,'staticList<List<SensorReading>>:*, readData<List<SensorReading>>:*)

Trace Semantics

Trace Predicates	Pre(1,T,DataBanker_init)={TRUE,i0}	Pre(1,T,DataBanker_write)={TRUE,i0}	Pre(1,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}
				⋁j { ¬[⋀i (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.

Output Behavior Definition	NONE(T)=NONE	NONE(T)=NONE	readData(T)=new List('sensorDataIn(r(i1,T)), 'sensorDataIn(r(i2,T)), 'sensorDataIn(r(i3,T)), 'sensorDataIn(r(i4,T)), 'sensorDataIn(r(i5,T)))	Undefined
Trace Postcondition	T.(PGM:DataBanker_init)-> T.(PGM:DataBanker_init)	T.(PGM:DataBanker_write,'sensorDataIn<SensorReading>:sd)-> T.(PGM:DataBanker_write,'sensorDataIn<SensorReading>:sd)	T.(PGM:DataBanker_read,readData'<List<SensorReading>>:UNEVAL)-> T.(PGM:DataBanker_read,readData'<List<SensorReading>>:new List(sd0,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

Function Call Event Descriptor	*Input Variable*	*Output Variable*	*Event Descriptor*
SensorDriver_get	windSpeed:int, whalePassing:Boolean	sensorData:SensorReading	(PGM:SensorDriver_get,'windSpeed<int>:*,'whalePassing<Boolean>:*,sensorData'<SensorReading>:*)

Trace Semantics

Trace Predicates	Pre(1,T,SensorDriver_get)={True,i}	$\bigvee j \{ \neg [\bigwedge i \text{ (Predicate } (i,j))] \}$
Output Behavior Definition	sensorData(T)=new SensorReading('windSpeed(r(i,T)), 'whalePassing(r(i,T)))	Undefined
Trace Postcondition	T.(PGM:SensorDriver_get,'windSpeed<int>:UNEVAL,'whalePassing<Boolean>:UNEVAL, sensorData'<SensorReading>:UNEVAL)⟶ T.(PGM:SensorDriver_get,'windSpeed<int>:aa,'whalePassing<Boolean>:bb, sensorData'<SensorReading>:new SensorReading(aa,bb))	Undefined
Monitored Variable Constraints	INTMIN ≤ aa ≤ INTMAX, bb = TRUE v FALSE	-
Executed Count	114	0
Total Sample	114	0
Occurrence Probability	100	0

TransmitDriver Module Interface Specification

Trace Event Syntax

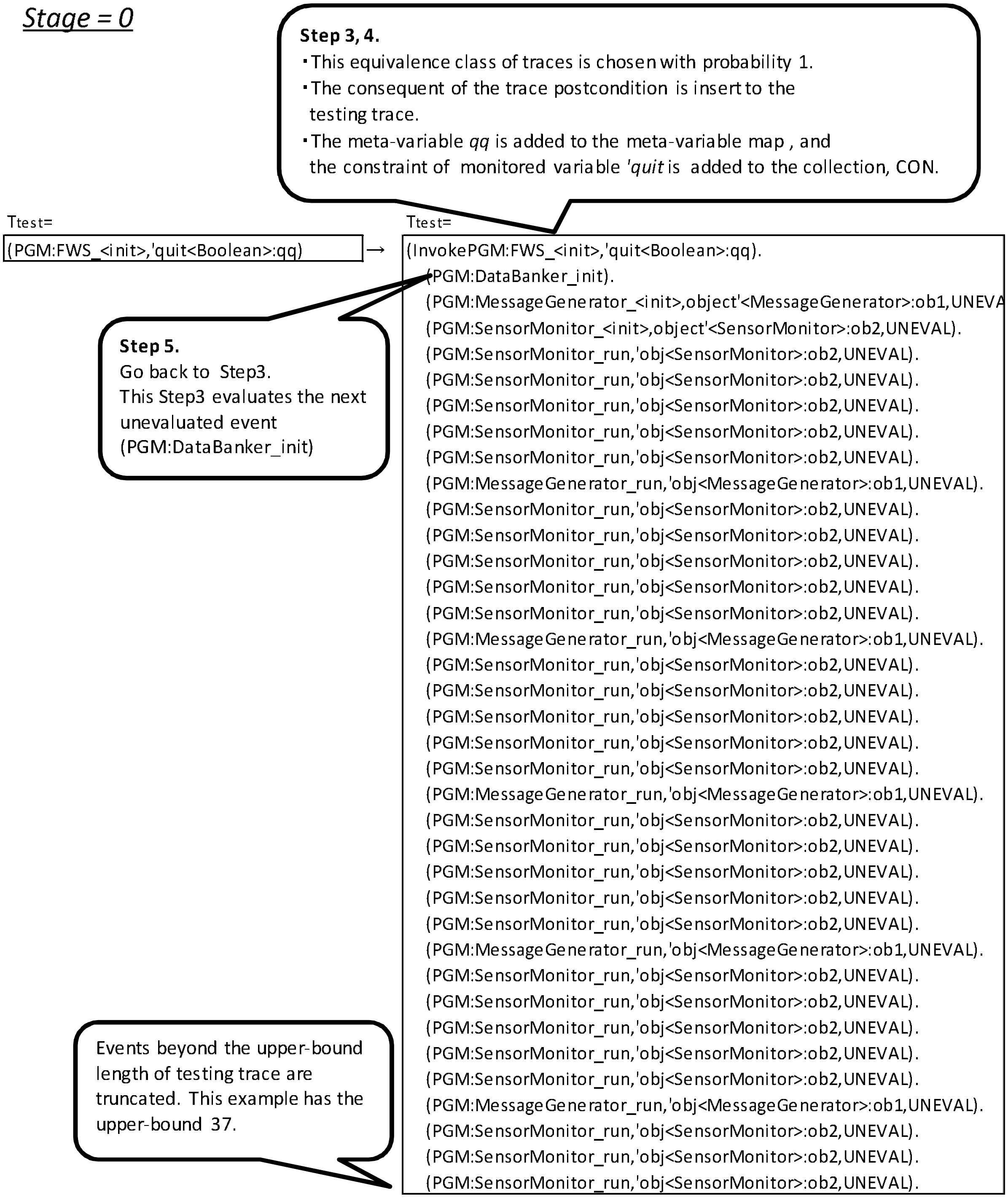
Program Name	*Input Variable*	*Output Variable*	*Event Descriptor*
sendWindSpeed	msg:MessageFormatWind	msg:MessageFormatWind	(PGM:TransmitDriver_sendWindSpeed,'msg<MessageFormatWind>:*,msg'<MessageFormatWind>:*)
sendToWhaleSurveyCenter	msg:MessageFormatWhale	msg:MessageFormatWhale	(PGM:TransmitDriver_sendToWhaleSurveyCenter,'msg<MessageFormatWhale>:*,msg'<MessageFormatWhale>:*)

Trace Semantics

Trace Predicates	Pre(1,T,TransmitDriver_sendWindSpeed)={True,i}	Pre(1,T,TransmitDriver_sendToWhaleSurveyCenter)={True,i}	⋁j { ¬[⋀i (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.
Output Behavior Definition	msg(T)='msg(r(I,T))	msg(T)='msg(r(I,T))	Undefined
Trace Postcondition	T.(PGM:TransmitDriver_sendWindSpeed,'msg<MessageFormatWind>:mm,msg'<MessageFormatWind>:UNEVAL)→ T.(PGM:TransmitDriver_sendWindSpeed,'msg<MessageFormatWind>:mm,msg'<MessageFormatWind>:mm)	T.(PGM:TransmitDriver_sendToWhaleSurveyCenter,'msg<MessageFormatWhale>:mm,msg'<MessageFormatWhae>:UNEVAL)→ T.(PGM:TransmitDriver_sendToWhaleSurveyCenter,'msg<MessageFormatWhale>:mm,msg'<MessageFormatWhae>:mm)	Undefined
Monitored Variable Constraints	-	-	-
Executed Count	114	3	0
Total Sample	114	3	0
Occurrence Probability	100	100	0

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.

Ttest=

```
(PGM:FWS_<init>,'quit<Boolean>:qq)
```

→

Ttest=

```
(InvokePGM:FWS_<init>,'quit<Boolean>:qq).
 (PGM:DataBanker_init).
 (PGM:MessageGenerator_<init>,object'<MessageGenerator>:ob1,UNEVA
 (PGM:SensorMonitor_<init>,object'<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:MessageGenerator_run,'obj<MessageGenerator>:ob1,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:MessageGenerator_run,'obj<MessageGenerator>:ob1,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:MessageGenerator_run,'obj<MessageGenerator>:ob1,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:MessageGenerator_run,'obj<MessageGenerator>:ob1,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:MessageGenerator_run,'obj<MessageGenerator>:ob1,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
 (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
```

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.

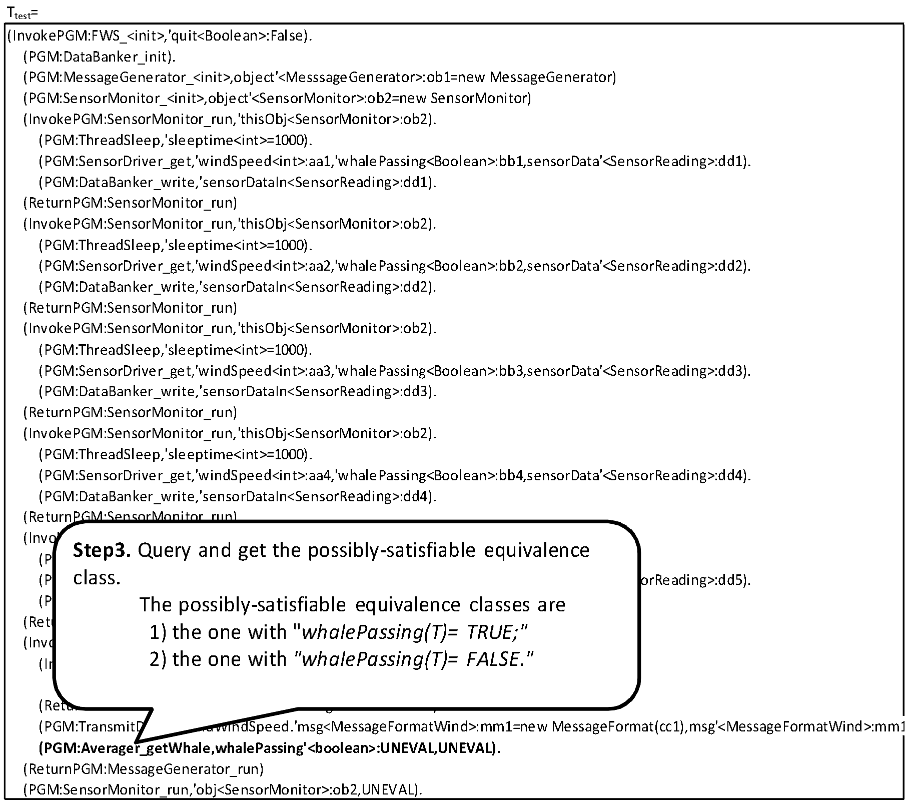
Meta-VariableMap=

Key	*Value*
qq	Monitored Variable
ob1	new MessageGenerator Instance
ob2	new SensorMonitor Instance

CON=

qq=FALSE

Stage = i



T_{test}=

```
(InvokePGM:FWS_<init>,'quit<Boolean>:False).
   (PGM:DataBanker_init).
   (PGM:MessageGenerator_<init>,object'<MesssageGenerator>:ob1=new MessageGenerator)
   (PGM:SensorMonitor_<init>,object'<SensorMonitor>:ob2=new SensorMonitor)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa1,'whalePassing<Boolean>:bb1,sensorData'<SensorReading>:dd1).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd1).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa2,'whalePassing<Boolean>:bb2,sensorData'<SensorReading>:dd2).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd2).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa3,'whalePassing<Boolean>:bb3,sensorData'<SensorReading>:dd3).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd3).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa4,'whalePassing<Boolean>:bb4,sensorData'<SensorReading>:dd4).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd4).
   (ReturnPGM:SensorMonitor_run)
   (Invo
      (P
      (P                                                                        orReading>:dd5).
      (P
   (Ret
   (Invo
      (In

      (Retu
      (PGM:TransmitD         ndWindSpeed.'msg<MessageFormatWind>:mm1=new MessageFormat(cc1),msg'<MessageFormatWind>:mm1
      (PGM:Averager_getWhale,whalePassing'<boolean>:UNEVAL,UNEVAL).
   (ReturnPGM:MessageGenerator_run)
   (PGM:SensorMonitor_run,'obj<SensorMonitor>:ob2,UNEVAL).
```

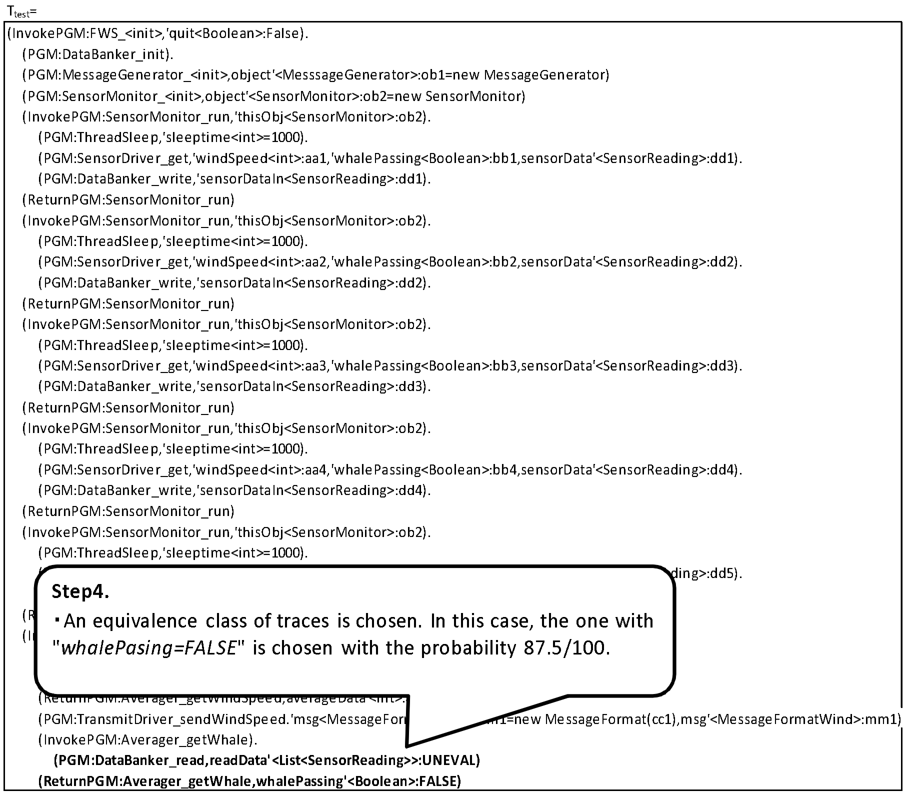
Step3. Query and get the possibly-satisfiable equivalence class.

The possibly-satisfiable equivalence classes are

1) the one with "*whalePassing(T)= TRUE;*"

2) the one with *"whalePassing(T)= FALSE.*"

Stage = i+1



T_{test}=

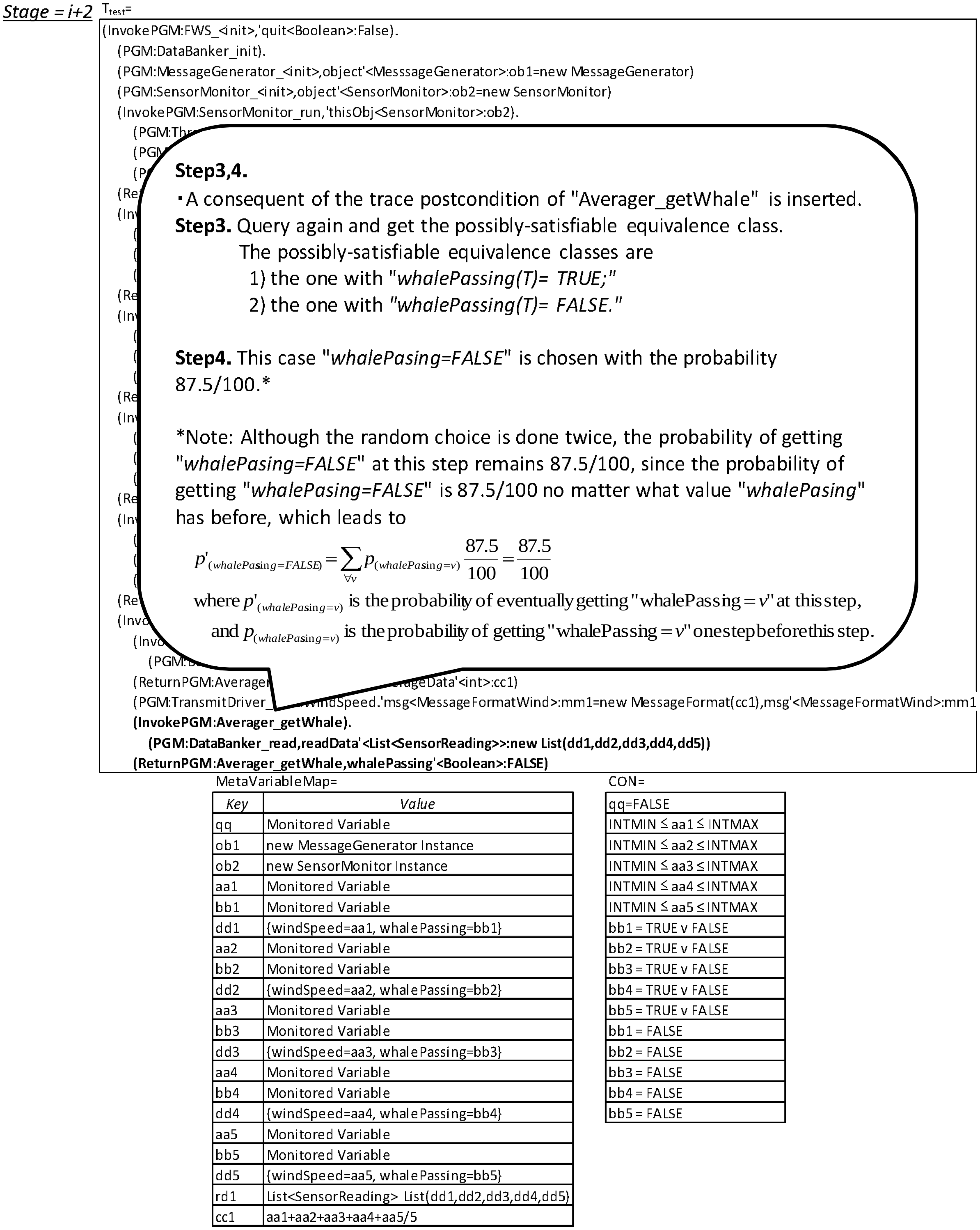
```
(InvokePGM:FWS_<init>,'quit<Boolean>:False).
   (PGM:DataBanker_init).
   (PGM:MessageGenerator_<init>,object'<MesssageGenerator>:ob1=new MessageGenerator)
   (PGM:SensorMonitor_<init>,object'<SensorMonitor>:ob2=new SensorMonitor)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa1,'whalePassing<Boolean>:bb1,sensorData'<SensorReading>:dd1).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd1).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa2,'whalePassing<Boolean>:bb2,sensorData'<SensorReading>:dd2).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd2).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa3,'whalePassing<Boolean>:bb3,sensorData'<SensorReading>:dd3).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd3).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:aa4,'whalePassing<Boolean>:bb4,sensorData'<SensorReading>:dd4).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd4).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (                                                                        ding>:dd5).

   (R
   (I


      (ReturnPGM:Averager_getWindSpeed,averageData<int>:
      (PGM:TransmitDriver_sendWindSpeed.'msg<MessageFor              m1=new MessageFormat(cc1),msg'<MessageFormatWind>:mm1)
      (InvokePGM:Averager_getWhale).
         (PGM:DataBanker_read,readData'<List<SensorReading>>:UNEVAL)
      (ReturnPGM:Averager_getWhale,whalePassing'<Boolean>:FALSE)
```

Step4.

・An equivalence class of traces is chosen. In this case, the one with "*whalePasing=FALSE*" is chosen with the probability 87.5/100.



Stage = i+2 T_{test}=

```
(InvokePGM:FWS_<init>,'quit<Boolean>:False).
   (PGM:DataBanker_init).
   (PGM:MessageGenerator_<init>,object'<MesssageGenerator>:ob1=new MessageGenerator)
   (PGM:SensorMonitor_<init>,object'<SensorMonitor>:ob2=new SensorMonitor)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:Thr
      (PGM
      (P
   (Re
   (In
      (
      (
      (
   (Re
   (In
      (
      (
      (
   (Re
   (In
      (
      (
      (
   (Re
   (In
      (
      (
      (
   (Re
   (Invo
      (Invo
         (PGM:D
      (ReturnPGM:Averager                    ageData'<int>:cc1)
      (PGM:TransmitDriver_     WindSpeed.'msg<MessageFormatWind>:mm1=new MessageFormat(cc1),msg'<MessageFormatWind>:mm1
      (InvokePGM:Averager_getWhale).
         (PGM:DataBanker_read,readData'<List<SensorReading>>:new List(dd1,dd2,dd3,dd4,dd5))
      (ReturnPGM:Averager_getWhale,whalePassing'<Boolean>:FALSE)
```

Step3,4.

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

Step3. 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."*

Step4. This case "*whalePasing=FALSE*" is chosen with the probability 87.5/100.*

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

$$p'_{(whalePasing=FALSE)} = \sum_{\forall v} p_{(whalePasing=v)} \frac{87.5}{100} = \frac{87.5}{100}$$

where $p'_{(whalePasing=v)}$ is the probability of eventually getting "whalePassing $= v$" at this step,
and $p_{(whalePasing=v)}$ is the probability of getting "whalePassing $= v$" one step before this step.

MetaVariableMap=

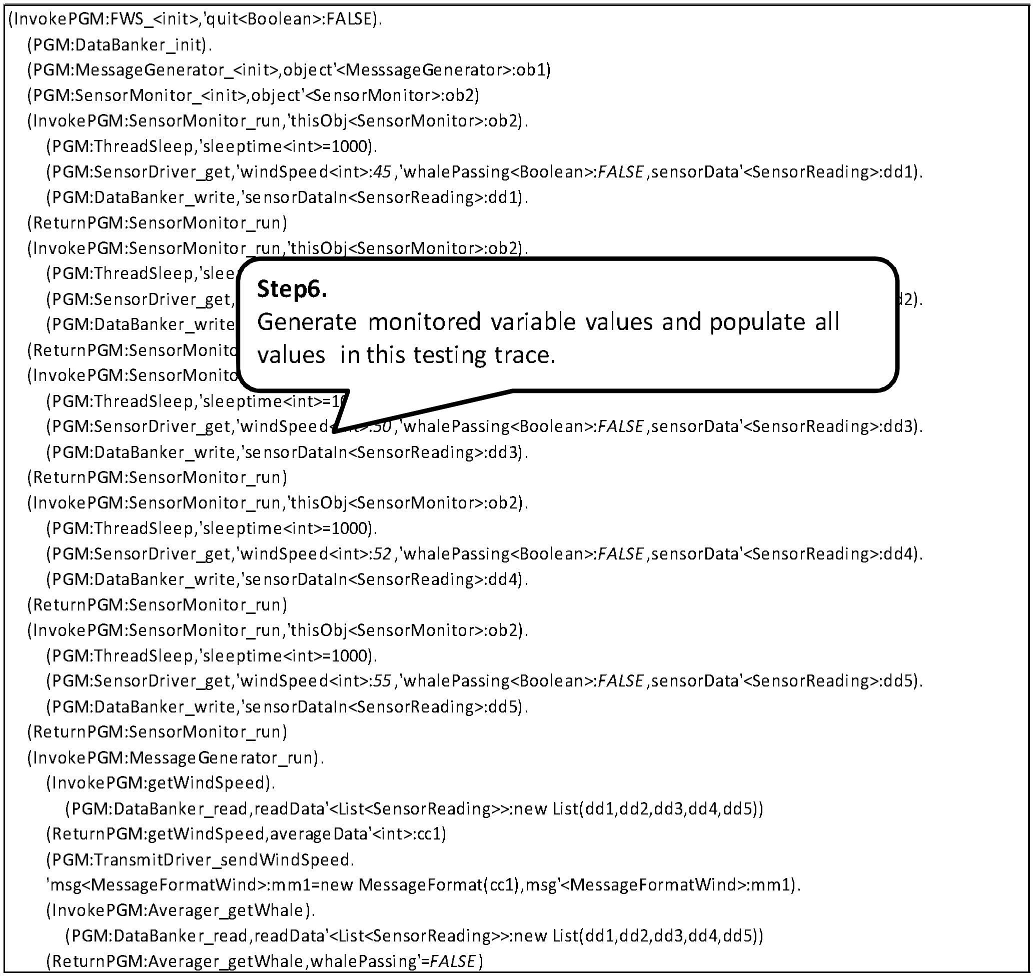
Key	*Value*
qq	Monitored Variable
ob1	new MessageGenerator Instance
ob2	new SensorMonitor Instance
aa1	Monitored Variable
bb1	Monitored Variable
dd1	{windSpeed=aa1, whalePassing=bb1}
aa2	Monitored Variable
bb2	Monitored Variable
dd2	{windSpeed=aa2, whalePassing=bb2}
aa3	Monitored Variable
bb3	Monitored Variable
dd3	{windSpeed=aa3, whalePassing=bb3}
aa4	Monitored Variable
bb4	Monitored Variable
dd4	{windSpeed=aa4, whalePassing=bb4}
aa5	Monitored Variable
bb5	Monitored Variable
dd5	{windSpeed=aa5, whalePassing=bb5}
rd1	List<SensorReading> List(dd1,dd2,dd3,dd4,dd5)
cc1	aa1+aa2+aa3+aa4+aa5/5

CON=

qq=FALSE
INTMIN ≤ aa1 ≤ INTMAX
INTMIN ≤ aa2 ≤ INTMAX
INTMIN ≤ aa3 ≤ INTMAX
INTMIN ≤ aa4 ≤ INTMAX
INTMIN ≤ aa5 ≤ INTMAX
bb1 = TRUE v FALSE
bb2 = TRUE v FALSE
bb3 = TRUE v FALSE
bb4 = TRUE v FALSE
bb5 = TRUE v FALSE
bb1 = FALSE
bb2 = FALSE
bb3 = FALSE
bb4 = FALSE
bb5 = FALSE

Stage = i+3 (Last Stage)

T_{test}=



```
(InvokePGM:FWS_<init>,'quit<Boolean>:FALSE).
   (PGM:DataBanker_init).
   (PGM:MessageGenerator_<init>,object'<MesssageGenerator>:ob1)
   (PGM:SensorMonitor_<init>,object'<SensorMonitor>:ob2)
   (InvokePGM:SensorMonitor_run,'thisObj<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).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'slee
      (PGM:SensorDriver_get,                                                                          d2).
      (PGM:DataBanker_write
   (ReturnPGM:SensorMonito
   (InvokePGM:SensorMonito
      (PGM:ThreadSleep,'sleeptime<int>=1
      (PGM:SensorDriver_get,'windSpeed<   :50,'whalePassing<Boolean>:FALSE,sensorData'<SensorReading>:dd3).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd3).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:52,'whalePassing<Boolean>:FALSE,sensorData'<SensorReading>:dd4).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd4).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:SensorMonitor_run,'thisObj<SensorMonitor>:ob2).
      (PGM:ThreadSleep,'sleeptime<int>=1000).
      (PGM:SensorDriver_get,'windSpeed<int>:55,'whalePassing<Boolean>:FALSE,sensorData'<SensorReading>:dd5).
      (PGM:DataBanker_write,'sensorDataIn<SensorReading>:dd5).
   (ReturnPGM:SensorMonitor_run)
   (InvokePGM:MessageGenerator_run).
      (InvokePGM:getWindSpeed).
         (PGM:DataBanker_read,readData'<List<SensorReading>>:new List(dd1,dd2,dd3,dd4,dd5))
      (ReturnPGM:getWindSpeed,averageData'<int>:cc1)
      (PGM:TransmitDriver_sendWindSpeed.
      'msg<MessageFormatWind>:mm1=new MessageFormat(cc1),msg'<MessageFormatWind>:mm1).
      (InvokePGM:Averager_getWhale).
         (PGM:DataBanker_read,readData'<List<SensorReading>>:new List(dd1,dd2,dd3,dd4,dd5))
      (ReturnPGM:Averager_getWhale,whalePassing'=FALSE)
```

Step6.
Generate monitored variable values and populate all values in this testing trace.

MetaVariableMap=

Key	*Value*
qq	FALSE
ob1	new MessageGenerator Instance
ob2	new SensorMonitor Instance
aa1	45
bb1	FALSE
dd1	{windSpeed=aa1, whalePassing=bb1}
aa2	51
bb2	FALSE
dd2	{windSpeed=aa2, whalePassing=bb2}
aa3	50
bb3	FALSE
dd3	{windSpeed=aa3, whalePassing=bb3}
aa4	52
bb4	FALSE
dd4	{windSpeed=aa4, whalePassing=bb4}
aa5	55
bb5	FALSE
dd5	{windSpeed=aa5, whalePassing=bb5}
rd1	List<SensorReading> List(dd1,dd2,dd3,dd4,dd5)
cc1	51

BIBLIOGRAPHY

[1] The Vienna Development Method: The Meta-Language. In Dines Bjø rner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[2] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification Language. In *On the Construction of Programs*, pages 343–410. 1980.

[3] R.L. Baber, D.L. Parnas, S.A. Vilkomir, Paul Harrison, and T. O'Connor. Disciplined methods of software specification: a case study. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 2, pages 428–437. IEEE, 2005.

[4] Wolfram Bartussek and David Lorge Parnas. Using assertions about traces to write abstract specifications for software modules. pages 211–236, October 1978.

[5] Boris Beizer. *Software testing techniques.* Van Nostrand Reinhold, 1990.

[6] Barry W. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, January 1984.

[7] CATS. CATS CO. LTD - ZIPC, 2012.

[8] Winfried Dulz. MaTeLo-statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. *Quality Software, 2003. Proceedings. Third*, page 336, November 2003.

[9] IK ElFar. Model-based Software Testing. *Encyclopedia of Software*, pages 1–22, 2001.

[10] Constance L Heitmeyer and John D Mclean. Abstract Requirements Specification : A New Approach and Its Application. (5):580–590, 1983.

[11] William Hetzel. *The Complete Guide to Software Testing.* Wiley; 2 edition, 1993.

[12] Daniel Hoffman. Module test case generation. *ACM SIGSOFT Software Engineering Notes*, pages 97–102, 1989.

[13] W E Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, 12(10):997–1005, October 1986.

[14] IBM. IBM - Rational Rhapsody, December 2012.

[15] IEEE. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), 1990.

[16] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Professional, 1992.

[17] Kenneth J. Turner. *Using Formal Description Techniques: An Introduction to Estell, Lotos and Sdl.* John Wiley & Sons Ltd, 1992.

[18] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[19] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley Professional; 1 edition, 1995.

[20] J. McLean. A formal method for the abstract specification of software. *Journal of the ACM (JACM)*, 31(3):600–627, 1984.

[21] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[22] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 2nd edition, 1997.

[23] Harlan D Mills. Zero Defect Software : Cleanroom Engineering Zero Defect Software : Cleanroom Engineering. *Advances in Computers*, 36:1–41, 1993.

[24] J.D. Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2):14–32, 1993.

[25] John D Musa. *Handbook of Software Reliability Engineering.* Mcgraw-Hill, 1996.

[26] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* John Wiley & Sons, 2011.

[27] N. Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. *Software Engineering, 2005. ICSE 2005.*, pages 284–292, 2005.

[28] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. pages 408–417, March 1984.

[29] David L. Parnas and Sergiy a. Vilkomir. Precise Documentation of Critical Software. *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pages 237–244, November 2007.

[30] David Lorge Parnas. Document based rational software development. *Knowledge-Based Systems*, 22(3):132–141, April 2009.

[31] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[32] D.L. Parnas. Precise documentation: The key to better software. *The Future of Software Engineering*, pages 125–148, 2011.

[33] D.L. Parnas and Marius Dragomiroiu. Module Interface Documentation-Using the Trace Function Method (TFM). *submitted to IEEE Trans. Softw. Eng.*, pages 1–34, 2006.

[34] D.L. Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer programming*, 25(1):41–61, 1995.

[35] DK Peters. Using test oracles generated from program documentation. *Software Engineering, IEEE*, 24(3):161–173, 1998.

[36] DK Peters. Requirements-based monitors for real-time systems. *Software Engineering, IEEE*, 2002.

[37] Colm Quinn, Sergiy Vilkomir, David Parnas, and S. Kostic. Specification of software component requirements using the trace function method. In *Software Engineering Advances, International Conference on*, volume 00, pages 50–50. IEEE, 2006.

[38] Stewart Shapiro. Classical Logic (Stanford Encyclopedia of Philosophy).

[39] Emad Shihab, ZM Jiang, and Bram Adams. Prioritizing the creation of unit tests in legacy software systems. *Software: Practice*, pages 1–22, 2011.

[40] Yabo Wang. Specifying and Simulating the Externally Observable Behavior of Modules. *SQRL Report*, (54), 2008.

[41] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley Professional, 1999.

[42] J.A. Whittaker and M.G. Thomason. A Markov chain model for statistical software testing. *Software Engineering, IEEE Transactions on*, 20(10):812–824, 1994.

[43] D.M. Woit. Specifying operational profiles for modules. In *ACM SIGSOFT Software Engineering Notes*, volume 18, pages 2–10. ACM, 1993.