Module Interface Documentation and Testing

David Lorge Parnas
Department of Electrical and Computer Engineering
McMaster University, Hamilton, Ontario Canada L8S 4K1

Outline

(1) What is a module? What is an object?

(2) Black boxes and traces

(3) Multi-object operations and traces

(4) Simplified traces for deterministic modules

(5) Predicates on traces

(6) Canonical representations of abstract states

(7) Module interface documentation

(8) Module simulation, simulators as oracles

(9) Traces as test cases, test case (trace) generation

(10) How long must a test case be? Module design to reduce test case length

(11) Documentation and grey box testing

(12) Integrated module testing tools

Modules and Objects(1)

History

Modules introduced as a management tool

People should be able to work independently

Information Hiding introduced as design guideline.

- design is not management
- good design makes management easier.

Early IH Modules created a single object.

Abstract Data Type Concept allowed many objects of the same type to be created by a single module.

Objects are nothing more than program variables.

Language designers add meaning to terms, e.g. by introducing inheritance.

Language designers restrict programmers by means of implementation assumptions. (e.g. calls)

It is important not to overload these terms and stick to the basic meanings.

Language designers allow old design errors to look like new technology.

Why We Need Module Interface Descriptions

(1) Multiperson projects

(2) Multiversion projects

(3) Our inability to do much" (E. W. Dijkstra).
- Each subtask should have a definition independent of the rest of the job.

(4) Making early decisions explicit and precise.
- Intramodule assumptions.
- Decision postponement.

(5) To allow an independent test group to work

(6) To allow test case generation

(7) To allow test result evaluation

Modules and Objects (2)

It is wise to design software by designing classes of objects.

Each object is implemented by a module (a set of programs) using a data structure that is "hidden from" (never accessed directly by) programs outside the module.

Changing the state of the object, or getting information about the object's state, is only done by invocations of programs from the module.

Definition:

An *object* is a finite state machine. The input alphabet of the object is the set of operations that one can perform upon the object. The output alphabet of the object is the set of values that can be returned by such operations. ❑

Describing or specifying objects is very different from describing or specifying programs.

Because the data structure is hidden, we must describe behaviour in terms of event sequences.

Descriptions of Objects (1)

We want “black box” descriptions of objects,

For such descriptions, references to the data structure used in the module are inappropriate.

For black box descriptions of finite state machines, the only information that can be mentioned are the externally visible events, i.e. sequences of inputs and outputs. We call such sequences *traces*.

Definition:

A *trace* of a finite state machine is a finite sequence of pairs, each containing a member of the input alphabet and a member of the output alphabet. A trace, T, is considered *possible* for machine M, if M could react to the sequence of inputs in T by emitting the sequence of outputs in T. ❑

A description must tell its reader whether or not any specific trace is possible for the object(s) in question.

Simplified Traces for Deterministic Modules

Most of the modules that interest us are deterministic.

The output is determined by the input history.

The outputs in the trace are redundant

The outputs can be deleted.

A trace becomes a sequence of inputs.

The output is specified as a function of the trace.

Descriptions of Objects (2)

Descriptions and specifications of objects can both be written as predicates on classes of traces.

Competence sets are not usually needed; programs implementing objects are assumed to terminate.

Any description or specification must be interpreted as a predicate on traces.

The issue is how to write such predicates

- in a way that is easily read and used as a reference
- in a way that can be checked for completeness
- in a way that can be used for testing
- in a way that does not suggest implementations
- in a way that does not unnecessarily restrict implementations.

Module vs. Program Specifications

Programs do not hide data.

Program effects can be described in terms of data structure.

Program effects are visible immediately.

Modules have hidden data.

Module specifications may not mention the data structure.

Module effects can have delayed visibility.

We use relational specifications for programs.

We use "trace assertions" for modules.

The two are completely compatible.

Canonical Representation of Abstract States

Definition:

Two objects are in the same *abstract state* if the user, working through the interface, cannot distinguish their states by some sequence of operations. ❑

There may be many internal states corresponding to each abstract state.

We need a way to represent abstract states.

We can use traces as representations. The history is all that the user can know about the object state, but ...

Many traces may result in the same abstract state.

We can use a canonical trace as a representation of the abstract state.

Definition:

A *canonical trace* is a unique trace chosen to represent the equivalence class of traces resulting in the same abstract state. ❑

We have been using canonical traces as unique representations of abstract states.

We are finding other canonical representations, consisting of queues and stacks, to be better (more readable).

Module Interface Documentation: Example: 12 Element Queue

(1) Syntax

ACCESS PROGRAMS

Program Name	Value	Arg#1
ADD		<integer>
REMOVE		
FRONT	<integer>	

(2) Canonical representation

$(\text{rep} = < [a_i]_{i=1}^{n} > \wedge\ (0 \le n \le 12)$

(3) Trace Extension Functions[1]

ADD([rep],a) ≡

conditions	new rep	extension class
n = 12	rep	%full%
n < 12	rep.a	

REMOVE([rep]) ≡

conditions	new rep	extension class
rep = _	rep	%empty%
rep ≠ _	$< [a_i]_{i=2}^{n} >$	

FRONT([rep]) ≡

conditions	new rep	extension class	Value returned
rep = _	rep	%empty%	
rep ≠ _	rep		a_1

[1] We use "." to denote sequence concatenation. [ ] enclose implicit arguments to functions.

Module Interface Documentation

How this document defines a predicate on traces.

(1) The syntax restricts the set of traces to those that could be executed.

(2) Using the trace extension functions repeatedly we can find the canonical representation for any trace allowed by the syntax. Even "error cases" are included in this description.

(3) For each trace, after finding its canonical representation, the value functions tell us the values that will appear in the trace.

(4) If a trace cannot be generated in this way, it is not in the set of traces characterised by the predicate, i.e. it should not happen.

(5) Extension classes are optional. There is a default class denoted by the empty string.

Module Interface Documentation

How can we check this document for completeness?

(1) There must be an extension function for each program mentioned in the syntax.

(2) There must be a value column or table for each program that returns values. Note that any parameter may return values; there should be a column or table for each.

(3) Each table must cover all possible values.

Module Interface Documentation

How could we use this documentation for simulation?

(1) Check the syntax for each call

(2) Determine the canonical representation for the abstract state.

(3) Determine values.

Non-Determinism

Extension “functions” can be relations

Values returned can be relations

Simulator can either

- make a random choice
- follow all possible paths
- check a predicate if comparing with an implementation.

These simulation techniques can be used as test oracles.

Traces as Test Cases

Traces (without the output values) represent test cases.

Using the specification we can generate syntactically correct test cases or, if we want them, incorrect cases.

For early testing, we can use the tables to generate test sets that cover every row in the tables and provide uniform case coverage.

For reliability estimation, we want the distribution of test cases to resemble actual usage. We must provide a description of the “operational profile”.

We need distributions for argument values.

We need to classify abstract states and provide probability distributions for extensions in each class.

Some Approaches:

- Transition probability matrix
- Transition matrix based on extension combined with extension probability matrix
- Classification Predicates combined with extension probability matrix

How long must test cases be?

In theory, long enough to return the module to a previously tested (internal) state.

In practice, this is not generally possible.

If we want to design for testability, we can implement so that there is a 1:1 mapping between internal states and abstract states. This will usually slow the implementation down.

Test parameterised specifications with small values of the parameters.

It may be useful to initialise regularly.

Connecting Object and Program Descriptions

Module Design Documentation

Module = private data structure + set of access programs

Design, usually in designer's head, must be written down

Three essential elements:

(1) description of the data structure

(2) abstraction function

Domain: data states

Range: canonical abstract representations

(d, tr) is in the abstraction function if tr, is the abstract representation of concrete data state, d

(3) program function (LD-relation)

- one for each possible invocation of each access program

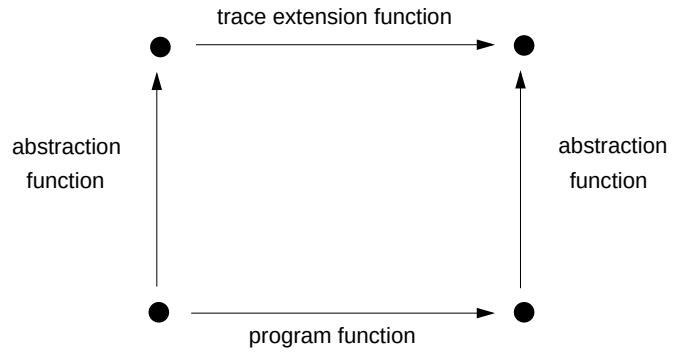
This is the information you need for pre-coding design reviews.

This is the information that guides the programmer.

This information can be used for "grey box" testing.

How can the workability of a design be verified?

For all possible invocations, the following must hold:



Top level shows abstract state view.

Lower level shows implementation view.

Abstraction function maps between them.

Example: Design Document for Queue12: Implementation 1

(1) DATA STRUCTURE

CONSTANTS

Constant Name	Definition
QSIZE	12

TYPES

Type Name	Definition
<qds>	array[0..QSIZE-1] of integer

VARIABLES

Type Definition/Name	Variables	Initial Values
<qds>	DATA	"Don't Care"
0..QSIZE-1	F, R	"Don't Care"
<boolean>	FULL	"Don't Care"
<boolean>	old	false

Abbreviation:

$edge \stackrel{df}{=} (R = F + 1) \vee (F = QSIZE\text{-}1) \wedge (R = 0)$
$`edge \stackrel{df}{=} (`R = `F + 1) \vee (`F = QSIZE\text{-}1) \wedge (`R = 0)$
$edge' \stackrel{df}{=} (R' = F' + 1) \vee (F' = QSIZE\text{-}1) \wedge (R' = 0)$
$\langle \mathbf{qs} \rangle \stackrel{df}{=} qds \times 0..QSIZE\text{-}1 \times 0..QSIZE\text{-}1 \times boolean$

(2) ABSTRACTION FUNCTION

af: <qs> → <queue12>

af(DATA,F,R,FULL,old) $\stackrel{df}{=}$

$(\neg edge \vee FULL) \wedge (F \geq R) \wedge old$	< DATA[F]. DATA[F–1] … . DATA[R] >
$(\neg edge \vee FULL) \wedge (F < R) \wedge old$	< DATA[F]. … . DATA[0] . DATA[QSIZE-1]. … . DATA[R] >
$edge \wedge \neg FULL \wedge old$	<>
$\neg old$	—

(3) PROGRAM FUNCTIONS

pf_Name	Arg#1	Value
pf_Q12INIT		<qs> → <qs>
gpf_ADD	<integer>	<qs> × <integer> → <qs>
pf_REMOVE		<qs> → <qs>
pf_FRONT		<qs> → <qs> × <integer>

pf_Q12INIT $\stackrel{df}{=}$

F' =	0
R' =	1
FULL' =	false
DATA' |	true
old =	true

gpf_ADD(a) $\stackrel{df}{=}$ NC(F) ∧ ∀j (j ≠ R') [NC(DATA[j])] ∧ NC(a) ∧

	(‘R = 0) ∧ old ∧			(‘R ≠ 0) ∧ old ∧			¬ old
	‘*edge* ∧		¬ ‘*edge*	‘*edge* ∧		¬ ‘*edge*	
	‘FULL	¬ ‘FULL		‘FULL	¬ ‘FULL		
DATA'[R'] =	‘DATA[‘R]	a	a	‘DATA[‘R]	a	a	‘DATA[‘R]
R' =	‘R	QSIZE-1	QSIZE-1	‘R	‘R – 1	‘R – 1	‘R
FULL' =	‘FULL	false	‘F = QSIZE-2	‘FULL	false	*edge*’	‘FULL

pf_REMOVE $\stackrel{df}{=}$ NC(DATA,R) ∧

	(¬ ‘*edge* ∨ ‘FULL) ∧ old ∧		(‘*edge* ∧ ¬‘FULL) ∨¬ old
	(‘F = 0)	(‘F > 0)	
F' =	QSIZE-1	‘F – 1	‘F
FULL' =	false	false	‘FULL

pf_FRONT $\stackrel{df}{=}$ NC(R,FULL, DATA, F) ∧

	¬ ‘*edge* ∨ ‘FULL old ∧	(‘*edge* ∧ ¬‘FULL) ∨ ¬ old
return value =	‘DATA[‘F]	

Using Module Design Documentation in “Grey Box” Testing.

We can do “black box” testing using conventional black box test case generation strategies.

In “black box” testing we can only check results.

Errors may be discovered long after things went wrong.

Using the design documents, we can check that the abstraction function describes what actually happens. We call this “grey box” testing.

If we check the program functions too (using the oracle generator), it is “clear box” testing.

In “clear box” testing we use the data state to determine test cases.

In “black box” and “grey box” testing, we use traces to generate test cases.

Integrated Testing Tools

ChunMing Li has combined our trace simulator (Yabo Wang), and Test Case Generator (Denise Woit), added some user interface features.

This tool allows you to:

(1) Provide a module interface specification.

(2) Provide an operational profile.

(3) Provide an implementation.

(4) Select some test criteria.

(5) Run tests.

(6) Receive a reliability estimate.

■ McMaster University ■

Future Work

Improved Statistical Measures.

Modernised TAM method.

More general table types.

Integration with other tools.

Better Operational Profile Specification.

More thought on the length of traces.

More usage and feedback.