Module Interface Documentation - Using The Trace Function Method (TFM)

David Lorge Parnas
Marius Dragomiroiu

Abstract
The Trace Function Method (TFM) for documenting (both describing and specifying) interfaces for information hiding modules and components is described. We begin by explaining the motivation for the method. The concepts of event, event descriptor, and trace are defined. Basic functions on event descriptors and traces are introduced. Finally, the method is illustrated on some simple examples.

Needed: A Way To Describe Software Without Writing Code.

In 1969 my manager told me:

• We should not need to write code to tell a programmer what the code they should write must do.
• A reviewer should be able to find out what a program is supposed to do without studying the code.
• Someone who wants to use software should be able to know what it does without reading the code.

Later we learned that:

• If we have several interchangeable implementations, we should be able to write down what they have in common.

In 1969 my manager told me:
Software Reference Documentation

Introductory documentation is designed to make it easy for a beginner to learn something completely new.
Reference documentation is designed to make it easy for an expert to look up specific details.
Contrast your first textbook on English with a Dictionary.
This discussion is about reference documentation.
It is not about introductory, sales, or FAQ documentation.

Defining The Required Content Of Documents

Everyone working on a project must know where to find specific information.
Everything they should want to know should be in the documentation.
As far as possible, nothing should be in two documents.
We define every document as a representation of a mathematical relation. This tells us what information should be in a document but not how it should be represented.
A Module Interface Document (MID) should define the value of output variables by a relation with the module history.
For deterministic modules, this is a function.

This Talk Is Not About Tabular Expressions

Tabular Expressions are multidimensional form of mathematical expressions. Usually tabular expressions are easier to read than conventional expression.
Each expression is an indexed set of grids.
Each grid is an indexed set of expressions.
An expression may be either a conventional expression or a tabular expression of a type that has been previously defined.
The definition of the meaning of a tabular expression is an equivalent conventional expression.
This talk uses, but does not define, tabular expressions.

Document Roles

Engineering usage:
- A description states properties of a product; it may include a mixture of incidental and required properties.
- A specification is a description that states only required properties.
- A full specification is a specification that states all required properties.
The same notation may be used for all 3.
These classifications are a matter of intent not notation.
There is no such thing as a “specification language”.

This Talk Is Not About Tabular Expressions

Tabular Expressions are multidimensional form of mathematical expressions. Usually tabular expressions are easier to read than conventional expression.
Each expression is an indexed set of grids.
Each grid is an indexed set of expressions.
An expression may be either a conventional expression or a tabular expression of a type that has been previously defined.
The definition of the meaning of a tabular expression is an equivalent conventional expression.
This talk uses, but does not define, tabular expressions.
Obligations Of Those Who Agree On A Specification

- Implementers may either accept the task of implementing that specification, or report problems with the specification and propose a revision; they may not accept the task and then (knowingly) build something that does not satisfy the specification.
- Users must be able to count on the properties stated in a specification. However, they must not base their work on any properties mentioned in any other description unless those properties are implied by the specification.
- Purchasers are obligated to accept any product that meets the specification that they agreed on.

Other descriptions may be useful for understanding particular implementations.

Software Design Issues

Programs, Components and Modules

Design Principles
- divide and conquer
- loose coupling
- separation of concerns
- encapsulation
- information hiding

Applying these principles requires us to document the interface information precisely and without revealing any internal information.

Multiple-Interface Modules (upper face, lower face, restricted face, other interfaces)

Earlier Approaches To Module Interface Documentation

Approaches to methods of writing module interface specifications can be divided into four classes:
- pragmatic, such as [Pa72a], [CP84], [CPS84]
- algebraic, following the pioneering work of Guttag[GH78]
- axiomatic, following the pioneering work of Zilles[LZ75]
- mixtures of the above.
- equivalent programs.

What's Wrong With Giving An Equivalent Program

May overspecify!
- bubble sort does not specify a sort.
- Languages often undefined. (pseudo code).
- Often overly complex.
- Often lies rather than abstractions
  - abstraction, one thing that represents many
  - everything derivable from abstraction must be true of all it represents
  - Otherwise it is a lie, and dangerous
- Limitations often not described.
Strengths And Weaknesses Of Various Approaches

Pragmatic approaches
- Work most of the time
- Major limitations in infrequently occurring cases.

Algebraic and Axiomatic Approaches
- Elegant
- Counter-Intuitive
- Fewer limitations than pragmatic but still limited.

Trace Assertion Approaches (TAM)
- Still Fewer Limitations
- Still counter-intuitive.
- Less elegant

Readability
No guarantee that a document is readable. An approach can allow and help writers to produce easily used documents

- Directness: Evaluation not deduction
  - no searching for applicable transformation rules
  - direct evaluation of functions
- Abstraction: only observables are mentioned.
- All required observables are described
- Ability to distinguish the essential information from incidental information
- A place for everything; everything in its place.

Axiomatic vs. Functional Approaches

Axiomatic approach
- Given a set of statements accepted as true (axioms)
- Given a set of rules of inference (transformations)
- Search for a sequence of applications of inference rules to axioms or proven statements to answer a question.

Functional Approach
- Given a set of expressions applying functions to variables
- Assign values to all variables.
- Evaluate functions using assigned values and earlier results.

Engineers - prefer evaluation (closed form).

What's New In TFM
The method described in this talk, TFM, deviates from past efforts in several significant ways:
- Not equational or axiomatic - evaluation of “closed form” expressions.
- Full use of multidimensional (tabular) expressions - same as other documents but no special tables for TFM.
- Almost conventional logic [Pa93].
- Limitations of algebraic and axiomatic documents removed.
- Can document modules that communicate through global variables
- States the most often needed information directly
- Abstracts from all implementation details
- Clearly distinguishes the essential information from other information
- Allows the use of standard mathematical concepts
- Supports a strict (arbitrary) organization for the information to ease retrieval and avoid duplication and inconsistency.
Communication With Software Modules

Software modules have two distinct data structures.
- a hidden (internal) data structure
- a global data structure

Note that:
- The “value” of a function program is treated as a global variable.
- When programs communicate using parameters, the arguments are placeholders for the shared/global variables that will be used.
- Often, the event is the invocation of one of the module’s externally accessible programs. A global variable that contains the name of the program invoked at an event, is a global variable, i.e. one of the inputs.
- Time, “cpu cycles consumed”, etc. which are often considered special, are also easily considered as global variables and require no special treatment.

Shared/global variables are the fundamental way that modules communicate.

Events

A software module may be viewed as a finite state machine operating at discrete points in time, which we call events.

At each event:
- reading some global variables (e.g. via input parameters), and
- changing its internal state, and
- changing the value of some of the global variables.

Remember: time is a global variable.

An event is instantaneous, i.e. the time between these activities cannot be observed by other components.

There are no simultaneous events. If two events occur at the same time, they are a single event.

Event Descriptors

Each element of the global data structure must have a unique identifier.

“PGM” is reserved for the name of a program invoked,

A full event descriptor specifies the values of every variable in the global data structure before and after the event.

Abbreviated event descriptors contain only input/output variables.

Example of an abbreviated event descriptor.

<table>
<thead>
<tr>
<th>PGM</th>
<th>in1</th>
<th>in2</th>
<th>out1</th>
</tr>
</thead>
<tbody>
<tr>
<td>name of program invoked in event</td>
<td>value of in1 before the event</td>
<td>value of in2 before the event</td>
<td>value of in1 after the event</td>
</tr>
</tbody>
</table>

Traces

A trace is a finite sequence of event descriptors; it describes a sequence of events.

A subtrace of a trace T is a sequence of the event descriptors that is contained within a trace T.

Prefix of T is a subtrace containing the first elements of T.

A 5 element trace:

<table>
<thead>
<tr>
<th>PGM</th>
<th>in1</th>
<th>in2</th>
<th>in3</th>
<th>out1</th>
</tr>
</thead>
<tbody>
<tr>
<td>name of program invoked in event</td>
<td>value of in1 before the event</td>
<td>value of in2 before the event</td>
<td>value of in3 before the event</td>
<td>value of out1 after the event</td>
</tr>
</tbody>
</table>

Note that “trace” is a purely formal concept.

A history, is a trace that accurately describes all of the events that affected a module after its initialization.
Trace Function (TFM) Component Interface Documentation

A TFM component interface document comprises:

• a complete description of the component's inputs (their type), and
• a complete description of the component's outputs (their type), and
• a description of a set of relations, each one describing the relation of the value of an output to the history of the events affecting or involving the component.

Note that histories includes all past behavior including the actual outputs; this means that one can use information about both past outputs and past inputs to determine the possible output values after the last event in a trace.

When Is A Trace-based Document Complete?

A TFM document is complete if there is a relation for every output and the complete set of possible traces for which the value of each output is defined is included in the domain of the corresponding relation.

If we have a description of the set of possible traces, we can determine completeness.

The domain of the description could also be a superset of that set.

When Is A Trace-based Document Consistent?

Because each output is defined separately (dependent only on inputs and earlier values of other outputs), the document is consistent if each individual relation is consistently defined.

Using tabular notation, consistency of a function/relation definition is usually easy to establish.

What Is A TFM Specification?

A TFM specification of a component M characterizes the set of traces that are be considered acceptable for M.

If

• any of the behaviors described in the specification as acceptable would be considered unacceptable by users, or
• any user-acceptable behaviors are excluded by the document,

it is the purported specification that is incorrect.

If an implementation shows behavior not allowed by a correct specification, the implementation is incorrect.
What Is A TFM Description?

A TFM description of an implementation of a module M is a TFM document that characterizes the set of traces that are possible with that implementation.

If the implementation exhibits any behavior not included in a document proposed as a complete description, or if the description describes behavior that never happens, it is the purported description that is incorrect.

When Is An Implementation Of A Module Correct?

Two stages:

• Produce a description of the implementation's behavior.
• Compare the description with the specification

In the comparison we determine:

• that the two documents match syntactically, i.e. that the inputs and outputs match in name and type,
• that each relation in the description is a subset of the corresponding relation in the specification,
• that the domain of each relation in the description contains the domain of the corresponding relation in the specification.

Modules That Create More Than One Object

• Viewing a component as creating many objects is only useful if the objects are independent - no side-effects.
• One can prepare much of the interface documentation as if the component created only one object.
• Each object must have a identifier.
• The identifier is prepended to the name of the operation in the style of “OO” languages.
• Additional objects are named as operands in the same way as operands of other types.
• Each object has a separate trace, “T.<object name>”.
• If an operation on one object will affect another, the group must be treated as a single object.

Primitive Functions On Event Descriptors

If e is an event descriptor and “V” is the unique name of a variable,

• “V(e)” denotes the value of V immediately before the event described by e
• “V'(e)” denotes the value of V immediately after that event
• PGM(e) is the name of the program invoked at that event (if any).

All of this information is in the event descriptor for e and consequently, in the trace.
**Primitive Functions On Traces**

- $L(T)$ (length) \(<\text{trace} \Rightarrow \text{integer}>\)
  
  - $L(T)$ is the number of event descriptors in $T$; $L(T)$ is 0 if $T$ is empty.
- $r(T)$ (most recent) \(<\text{trace} \Rightarrow \text{event descriptor}>\)
  
  - $r(T)$ is the last (most recent) event descriptor in the trace
- $o(T)$ (oldest) \(<\text{trace} \Rightarrow \text{event descriptor}>\)
  
  - $o(T)$ is the oldest event descriptor in the trace; it is undefined if $L(T) = 0.$
- $p(T)$ (precursor) \(<\text{trace} \Rightarrow \text{trace}>\)

  
  - $p(T) = L(T) = 0$
  
  - $L(T) = 1$ 
  
  - $L(T) > 1$ 

There are more in the papers.

**Useful Function Generators**

- $P$ is a predicate on event descriptors.
- $T$ is a trace.

- $ex(P)(T)$ (exists)
  
  - $ex(P)(T)$ is true if and only if $T$ contains an event descriptor that satisfies $P.$
- $ost(P)(T)$ (oldest such that)
  
  - $ost(P)(T)$ (index oldest such that)
- $irst(P)(T)$ (index recent such that)

There are more in the papers.

---

**Date Storage Module: Schematic View (optional)**

- $PGM$ to $day$
- $in1$ to $month$
- $in2$ to $year$
- $value$

**Date Storage Module: Variable Declarations**

**Output Variables**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id&gt;.day</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.month</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.year</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.Value</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

**Input Variables**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM</td>
<td>&lt;program name&gt;</td>
</tr>
<tr>
<td>in1</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>in2</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>date</td>
<td></td>
</tr>
</tbody>
</table>
Date Storage Module: Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Name</th>
<th>Value</th>
<th>in1</th>
<th>in2</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETDAY</td>
<td>&lt;id&gt;</td>
<td>&lt;integer&gt;</td>
<td></td>
<td></td>
<td>(PGM:SETDAY, 'in', 'day')</td>
</tr>
<tr>
<td>SETMONTH</td>
<td>&lt;id&gt;</td>
<td>&lt;integer&gt;</td>
<td></td>
<td></td>
<td>(PGM:SETMONTH, 'in', 'month')</td>
</tr>
<tr>
<td>SETYEAR</td>
<td>&lt;id&gt;</td>
<td>&lt;integer&gt;</td>
<td></td>
<td></td>
<td>(PGM:SETYEAR, 'in', 'year')</td>
</tr>
<tr>
<td>GETDAY</td>
<td>&lt;id&gt;</td>
<td>&lt;integer&gt;</td>
<td></td>
<td></td>
<td>(PGM:GETDAY, Value, 'day')</td>
</tr>
<tr>
<td>GETMONTH</td>
<td>&lt;id&gt;</td>
<td>&lt;integer&gt;</td>
<td></td>
<td></td>
<td>(PGM:GETMONTH, Value, 'month')</td>
</tr>
<tr>
<td>GETYEAR</td>
<td>&lt;id&gt;</td>
<td>&lt;integer&gt;</td>
<td></td>
<td></td>
<td>(PGM:GETYEAR, Value, 'year')</td>
</tr>
<tr>
<td>NEWDATE</td>
<td>&lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td>(PGM:NEWDATE, 'in2', 'in2')</td>
</tr>
<tr>
<td>DELETEDATE</td>
<td>&lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td>(PGM:DELETEDATE, 'in2', 'in2')</td>
</tr>
<tr>
<td>COPYDATE</td>
<td>&lt;id&gt;</td>
<td></td>
<td></td>
<td></td>
<td>(PGM:COPYDATE, 'in2')</td>
</tr>
</tbody>
</table>

Date Storage Module: Auxiliary Functions

\[\text{day}(T) = \]
\[
\begin{align*}
\text{day}(T) & = \\
& = \text{day}(r(T)) \\
& = \text{day}(p(T))
\end{align*}
\]

\[\text{month}(T) = \]
\[
\begin{align*}
\text{month}(T) & = \text{month}(r(T)) \\
& = \text{month}(p(T))
\end{align*}
\]

Output Functions

\[\text{year}(T) = \]
\[
\begin{align*}
\text{year}(T) & = \\
& = \text{year}(r(T)) \\
& = \text{year}(p(T))
\end{align*}
\]

\[\text{Value}(T) = \]
\[
\begin{align*}
\text{Value}(T) & = \text{Value}(r(T)) \\
& = \text{Value}(p(T))
\end{align*}
\]
Time Storage Module

Output Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hr</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>min</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>in</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET HR</td>
<td>&lt;integer&gt;</td>
<td>(PGM SET HR, 'in', hr')</td>
</tr>
<tr>
<td>SET MIN</td>
<td>&lt;integer&gt;</td>
<td>(PGM SET MIN, 'in', min')</td>
</tr>
<tr>
<td>INC</td>
<td>&lt;integer&gt;</td>
<td>(PGM INC, 'in', min')</td>
</tr>
<tr>
<td>DEC</td>
<td>&lt;integer&gt;</td>
<td>(PGM DEC, 'in', min')</td>
</tr>
</tbody>
</table>

Output Functions

\[ hr(T) = \begin{cases} 
  PGM(r(T)) = \text{SET HR} & \text{if } 0 \leq \text{hr}(T) < 24 \\
  \text{hr}(p(T)) & \text{if } \text{hr}(T) \geq 24 
\end{cases} \]

\[ \text{min}(T) = \begin{cases} 
  PGM(r(T)) = \text{SET MIN} & \text{if } 0 \leq \text{min}(T) < 59 \\
  \text{min}(p(T)) & \text{if } \text{min}(T) \geq 59 
\end{cases} \]

More Output Functions

Stack With Limited Range And Depth: Declarations

Output Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>depth</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>exc</td>
<td>{none, range, depth, empty}</td>
</tr>
<tr>
<td>Value</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

Input Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM</td>
<td>(push, pop, top, depth)</td>
</tr>
<tr>
<td>in</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>LB</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>UB</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>
Stack With Limited Range And Depth: Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>'Value</th>
<th>in</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>&lt;integer&gt;</td>
<td>(PGM:PUSH, 'in', top, depth', exc)</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td></td>
<td>(PGM:POP, top', depth', exc')</td>
<td></td>
</tr>
<tr>
<td>TOP</td>
<td>&lt;integer&gt;</td>
<td>(PGM:TOP, Value', exc')</td>
<td></td>
</tr>
<tr>
<td>DEPTH</td>
<td>&lt;integer&gt;</td>
<td>(PGM:DEPTH, Value')</td>
<td></td>
</tr>
</tbody>
</table>

The above is for explanation only. It is derivable from the other parts of the specification and is not included.

Stack: Auxiliary Functions

inrange(i) = LB ≤ i ≤ UB

noeffect(e) = (PGM(e)=PUSH ∧ (~inrange('in(e))) ∨ (PGM(e)=TOP) ∨ (PGM(e)=DEPTH)

full(T) = (depth(T)=d)
empty(T) = (depth(T)=0)
unpush(T,n) =

\[\begin{array}{c|c}
T = _ & \\
\neg(T = _) ^ & \\
\neg(noeffect(r(T)) ^ & \\
\neg(full(p(T)) ^ & \\
\neg(n = 1 ^ & \\
\neg(p(T) ^ & \\
\neg(n > 1 ^ & \\
\neg(empty(p(T)) ^ & \\
\neg(empty(p(T)) ^ & \\
\neg(unpush(p(T), n+1) ^ & \\
\end{array}\]

Stack: Output Variable Functions

top(T) =

\[\begin{array}{c|c}
T = _ & \\
\neg(T = _) ^ & \\
\neg(noeffect(r(T)) ^ & \\
\neg(full(p(T)) ^ & \\
\neg(n = 1 ^ & \\
\neg(p(T) ^ & \\
\neg(n > 1 ^ & \\
\neg(empty(p(T)) ^ & \\
\neg(empty(p(T)) ^ & \\
\neg(unpush(p(T), n+1) ^ & \\
\end{array}\]

Value(T) =

\[\begin{array}{c|c}
PGM(r(T)=TOP & \\
PGM(r(T)=DEPTH & \\
PGM(r(T)=PUSH & \\
\end{array}\]

depth(T) =

\[\begin{array}{c|c}
T = _ & \\
\neg(T = _) ^ & \\
\neg(noeffect(r(T)) ^ & \\
\neg(depth(p(T)) = 0 ^ & \\
\neg(depth(p(T)) ≠ 0 ^ & \\
\neg(PGM(r(T)=PUSH ^ & \\
\neg(depth(p(T)) = d ^ & \\
\neg(depth(p(T)) ≠ d ^ & \\
\neg(depth(p(T)) + 1 ^ & \\
\end{array}\]
**Stack: Output Variable Functions**

\[
\text{exc}(T) = \begin{cases} 
\neg \text{inrange}(\text{`in'(r(T)))} & \text{range} \\
\text{inrange}(\text{`in'(r(T)))} & \text{depth}(T) = d \\
\neg \text{depth}(T) = d & \text{none} \\
\text{PGM}(r(T)) = \text{TOP} \land \neg \text{depth}(T) = 0 & \text{none} \\
\text{PGM}(r(T)) = \text{DEPTH} & \text{none}
\end{cases}
\]

**What Can Be Done With TFM Documents**

**Discuss** them with users

**Reviewers** with substance to work on

Allow a user to review by **simulation**.

Give them to a programmer to **implement**.

Give them to **testers**, who

- use them to generate test programs, chose test cases
- use them to evaluate results

Give them to **maintainers**

**Conclusions**

Writing precise documentation will never be easy but

- Simple cases are simple.
- Complex cases remain complex. (reconsider the design)
- The key to simple specifications remains good design.

TFM seems to be as good as we have.

Take it as a challenge!