# Module Interface Documentation - Using the Trace Function Method (TFM)

**David Lorge Parnas and Marius Dragomiroiu**
**Software Quality Research Laboratory**
**Faculty of Informatics and Electronics**
**University of Limerick, Ireland**

## Abstract

**A new approach to the documentation (description or specification) of interfaces for information hiding components, the Trace Function Method (TFM), is described. The motivation and design assumptions behind the method are explained. The concepts of event, event descriptor, and trace are defined. Basic functions on event descriptors and traces are introduced. The method is illustrated on some simple examples.**

# Introduction

Needed: a way to describe software without code.

Software Reference Documentation

Defining the required content of documents

Not about tabular expressions

# Specifications and Other Descriptions

- **A description states properties of a product; it may include both 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.**

# 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 another description unless those properties are included in the specification.**
- **Purchasers are obligated to accept any product that meets the (full) specification that they provided to the supplier.**

**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 guidelines requires that we document the interface information precisely and without revealing any internal information.**

**Multiple-Interface Modules (upper face, lower face, restricted face)**

# 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, such as the pioneering work of Guttag[GH78]**
- **axiomatic, such as the pioneering proposal of Zilles[LZ75]**
- **mixtures of the above.**

# 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

- 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**
- **Abstraction**
- **Ability to distinguish the essential from incidental information**
- **Organization**

# What's new in TFM

**The method described in this paper, TFM, deviates from past efforts in several significant ways:**

- **Not equational or axiomatic - "closed form"**
- **Full use of multidimensional (tabular) expressions - same as other documents.**
- **Almost conventional logic [Pa93].**
- **Limitations removed.**
- **Can document modules that communicate through global variables**
- **states the most often needed information directly**
- **abstracts from internal implementation details**
- **clearly distinguishes the essential information from other information**
- **allows the use of standard mathematical concepts**
- **dictates a strict organization for the information to ease retrieval**

# Communication with Software Modules

## Software modules 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 eventually be used for communication.
- 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, must be regarded as a global variable that is one of the inputs.
- Time and such things as “cpu cycles consumed”, which are often considered special in some inexplicable way, are also easily considered as global variables and require no special treatment.

## Shared/global variables are 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.**

# Event descriptors

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

**PGM is reserved for program invoked at an event**

**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**

**Example of an abbreviated event descriptor.**

| **PGM** | ‘io | ‘in | io’ | out’ |
|---|---|---|---|---|
| name of program invoked in event | value of io before the event | value of in before the event | value of io after the event | value of out after the event |

# 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.

A prefix of a trace T contains the first n elements of T.

A 5 element trace:

| PGM | ‘io1 | ‘in2 | io1’ | out1’ |
|---|---|---|---|---|
| name of program invoked | value of io1 before the event | value of in2 before the event | value of io1 after the event | value of out1 after the event |
| name of program invoked | value of io1 before the event | value of in2 before the event | value of io1 after the event | value of out1 after the event |
| name of program invoked | value of io1 before the event | value of in2 before the event | value of io1 after the event | value of out1 after the event |
| name of program invoked | value of io1 before the event | value of in2 before the event | value of io1 after the event | value of out1 after the event |
| name of program invoked | value of io1 before the event | value of in2 before the event | value of io1 after the event | value of out1 after the event |

Note that “trace” is a 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 values of the inputs.**

**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.**

# 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 document as acceptable would be considered unacceptable by users, or if any user-acceptable behavior is not described, the purported specification 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, that purported description is incorrect.**

# When is an implementation of a module correct?

## Two stages:

- Produce a TFM description of the behavior of the implementation
- Compare the TFM description with the TFM 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.**
- **Additional objects are named as operands in the same way as operands of other types.**
- **Each object has a separate trace,“T.<object name>”.**

# 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).

# Basic functions and predicates on traces

**L(T) (length)**

**r(T) (most recent)**

**o(T) (oldest)**

**p(T) (precursor)**

**(T) (subsequent)**

**rn(n,T) (most recent n)**

**pn(n,T) (precursor of most recent n)**

**on(n,T) (oldest n)**

**sn(n,T) (subsequent n)**

**mrcall(pg,T) (most recent)**

# Useful function generators on traces

**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)**

***et(P)(T)* (extracted trace)**

**irst(P)(T) (index recent such that)**

**iost(P)(T) (index oldest such that**

# Date Module

**Output Variables**

| Variable Name | Type |
|---|---|
| <id>.day | <integer> |
| <id>.month | <integer> |
| <id>.year | <integer> |
| <id>. **Value** | <integer> |

**Input Variables**

| Variable Name | Type |
|---|---|
| PGM | <program name> |
| in1 | <integer> |
| in2 | date |

## Access Programs

| Program Name | Oname | Value | in1 | in2 | Abbreviated Event Descriptor |
|---|---|---|---|---|---|
| SETDAY | <id> | | <integer> | | (**PGM**:SETDAY, ‘in, day’) |
| SETMONTH | <id> | | <integer> | | (**PGM**:SETMONTH, ‘in, month’, ) |
| SETYEAR | <id> | | <integer> | | (**PGM**:SETYEAR, ‘in, year’) |
| GETDAY | <id> | <integer> | | | (**PGM**:GETDAY, **Value**’, ‘day) |
| GETMONTH | <id> | <integer> | | | (**PGM**:GETMONTH, **Value**’, ‘month) |
| GETYEAR | <id> | <integer> | | | (**PGM**:GETYEAR, **Value**’, ‘year) |
| NEWDATE | <id> | | | <id> | (PGM:NEWDATE, ‘<in2>, <in2>’) |
| DELETEDATE | | | | <id> | (**PGM**:DELETEDATE, ‘<in2>, <in2>’) |
| COPYDATE | | | | <id> | (**PGM**:COPYDATE, ‘<in2>) |

## Auxiliary Functions

day(T) ≡

| | | |
|---|---|---|
| (T = _) ∨ **PGM**(r(T) = NEWDATE | | 0 |
| ¬(T = _) ∧ | (**PGM**(r(T)) = SETDAY) | ‘in(r(T)) |
| | (**PGM**(r(T)) = COPYDATE) | day(‘in2(r(T)) |
| | (**PGM**(r(T)) = DELETEDATE) | ▨ |
| | ¬(**PGM**(r(T)) = SETDAY ∨<br>**PGM**(r(T)) = COPYDATE ∨<br>**PGM**(r(T)) = DELETEDATE ∨<br>**PGM**(r(T) = NEWDATE ) | day(p(T)) |

month(T) ≡

| | | |
|---|---|---|
| (T = _) ∨ **PGM**(r(T) = NEWDATE | | 0 |
| ¬(T = _) ∧ | (**PGM**(r(T)) = SETMONTH) | ‘in(r(T)) |
| | (**PGM**(r(T)) = COPYDATE) | month(‘in2(r(T)) |
| | (**PGM**(r(T)) = DELETEDATE) | ▨ |
| | ¬(**PGM**(r(T)) = SETMONTH ∨<br>**PGM**(r(T)) = COPYDATE ∨<br>**PGM**(r(T)) = DELETEDATE ∨<br>**PGM**(r(T) = NEWDATE ) | month(p(T)) |

year(T) ≡

| | | |
|---|---|---|
| (T = _) ∨ **PGM**(r(T) = NEWDATE | | 0 |
| ¬(T = _) ∧ | (**PGM**(r(T)) = SETYEAR) | ‘in(r(T)) |
| | (**PGM**(r(T)) = COPYDATE) | year(‘in2(r(T))) |
| | (**PGM**(r(T)) = DELETEDATE) | ▨ |
| | ¬(**PGM**(r(T)) = SETYEAR ∨<br>**PGM**(r(T)) = COPYDATE ∨<br>**PGM**(r(T)) = DELETEDATE ∨<br>**PGM**(r(T) = NEWDATE ) | year(p(T)) |

**Value** (T) ≡

| | |
|---|---|
| **PGM**(r(T)) = GETDAY | day'(T) |
| **PGM**(r(T)) = GETYEAR | year'(T) |
| **PGM**(r(T)) = GETMONTH | month'(T) |
| ¬( **PGM**(r(T)) = GETDAY ∨ **PGM**(r(T)) = GETYEAR ∨ **PGM**(r(T)) = GETMONTH ) | |

**Output Functions**

<id>.day ≡ day($T_{<id>}$)

<id>.month ≡ month($T_{<id>}$)

<id>. year ≡ year($T_{<id>}$)

<id>.Value ≡ Value($T_{<id>}$)

# Time storage module

**Output Variables**

| **Variable Name** | **Type** |
|---|---|
| hr | <integer> |
| min | <integer> |

**Access Programs**

| **Program Name** | **‘in** | **Abbreviated Event Descriptor** |
|---|---|---|
| SET HR | <integer> | (**PGM**:SET HR, ‘in, hr’) |
| SET MIN | <integer> | (**PGM**: SET MIN, ‘in, min’) |
| INC | | (**PGM**:INC, hr’, min’) |
| DEC | | (**PGM**:DEC, hr’, min’) |

**Output Functions**
**hr(T) ≡**

| | | |
|---|---|---|
| **PGM**(r(T)) = SET HR ∧ | $0 \leq \text{’in}(r(T)) < 24$ | ‘in(r(T)) |
| | $\neg (0 \leq \text{in}(r(T)) < 24)$ | hr((p(T))) |
| **PGM**(r(T)) = SET MIN | | hr((p(T))) |

| | | | |
|---|---|---|---|
| **PGM**(r(T)) = INC ∧ | min(p(T))= 59 ∧ | hr(p(T))= 23 | 0 |
| | | ¬ hr(p(T))= 23 | 1+ hr((p(T))) |
| | ¬ (min(p(T))=59) | | hr((p(T))) |
| **PGM**(r(T)) = DEC ∧ | ¬ (min(p(T))= 0) | | hr((p(T))) |
| | min(p(T))= 0 ∧ | ¬ (hr(p(T)))= 0 | hr((p(T)))-1 |
| | | hr(p(T))= 0 | 23 |
| T= _ | | | 0 |

min(T) ≡

| | | |
|---|---|---|
| **PGM**(r(T)) = SET HR | | min(p(T)) |
| **PGM**(r(T)) = SET MIN ∧ | 0 ≤ ‘in(r(T)) ≤ 59 | ‘in(r(T)) |
| | ¬ (0 ≤ ‘in(r(T)) ≤ 59) | min(p(T)) |
| **PGM**(r(T)) = INC ∧ | min(p(T)) = 59 | 0 |
| | ¬ (min(p(T))=59) | min(p(T)) + 1 |
| **PGM**(r(T)) = DEC ∧ | ¬ (min(p(T))= 0) | min((p(T))) –1 |
| | min(p(T))= 0 | 59 |
| T= _ | | 0 |

# A stack of limited range and depth

**Output Variables**

| **Variable Name** | **Type** |
|---|---|
| top | <integer> |
| depth | <integer> |
| exc | {none, range, depth, empty} |
| Value | <integer> |

**Access Programs**

| Program Name | **'Value** | 'in | Abbreviated Event Descriptor |
|---|---|---|---|
| PUSH | | <integer> | (**PGM**:PUSH, 'in, top', depth',exc') |
| POP | | | (**PGM**:POP, top', depth', exc') |
| TOP | <integer> | | (**PGM**:TOP, **Value'**, exc') |
| DEPTH | <integer> | | (**PGM**:DEPTH, **Value'**) |

**Auxiliary Functions**

$\text{inrange}(i) \equiv LB \leq i \leq UB$

$\text{noeffect}(e) \equiv (\text{PGM}(e)=\text{PUSH} \wedge (\neg \text{inrange}(\text{'in}(e))) \vee \text{PGM}(e)=\text{TOP} \vee \text{PGM}(e) = \text{DEPTH}$

$\text{full}(T) \equiv \text{depth}(T) = d$

$\text{empty}(T) \equiv \text{depth}(T) = 0$

ps(T1,T2) ≡

| | | | |
|---|---|---|---|
| T2 = _ | | | T1 |
| (T2 ≠ _ ) ∧ noeffect(o(T2)) | | | ps(T1,s(T2)) |
| (T2 ≠ _ ) ∧ ¬noeffect(o(T2))∧ | PGM(o(T2))=PUSH ∧ | full(T1) | ps(T1,s(T2)) |
| | | ¬ full(T1) | ps(T1.o(T2),s(T2)) |
| | PGM(o(T2))=POP ∧ | ¬empty(T1) | ps(p(T1), s(T2)) |
| | | empty(T1) | ps(T1, s(T2)) |

strip(T) ≡ ps(_,T)

**Output variable functions**

top(T) ≡

| | |
|---|---|
| strip(T) = _ | |
| strip(T) ≠ _ | ‘in(r(strip(T))) |

**Value**(T) ≡

| | |
|---|---|
| **PGM**(r(T))=TOP | top(p(T)) |
| **PGM**(r(T))=DEPTH | depth(p(T)) |
| **PGM**(r(T))=PUSH | |
| **PGM**(r(T))=POP | |

depth(T) ≡

| | | | | |
|---|---|---|---|---|
| T = _ | | | | 0 |
| (T ≠ _) ∧ | noeffect(r(T)) | | | depth(p(T)) |
| | ¬ noeffect(r(T)) ∧ | **PGM**(r(T))=POP ∧ | depth(p(T))= 0 | 0 |
| | | | depth(p(T)) ≠ 0 | depth(p(T)) - 1 |
| | | **PGM**(r(T))=PUSH ∧ | depth(p(T))= d | d |
| | | | depth(p(T)) ≠ d | depth(p(T)) + 1 |

exc(T) ≡

| | | | |
|---|---|---|---|
| PGM(r(T))=PUSH ∧ | ¬inrange(‘in(r(T))) | | range |
| | *inrange(‘in(r(T))) ∧* | L(strip(p(T))) = d | depth |
| | | ¬(L(strip(p(T))) = d) | none |
| (PGM(r(T))=POP ∨ PGM(r(T))= TOP) ∧ | L(strip(p(T))) = 0 | | empty |
| | ¬ (L(strip(p(T))) = 0) | | none |
| PGM(r(T))=DEPTH | | | none |

# Conclusions

Nobody should think that writing precise documentation is easy;

Simple cases are simple.

Complex cases remain complex.

The key to simple specifications remains good design.

TFM seems to be as good as it gets.