Alan Wassyng

Software Quality Research Laboratory
Department of Computing and Software
McMaster University

Engineering High Quality Software Applications

Presented at
CUSEC 2005

Where we are

SQRL
Software Quality Research Laboratory

Where we are - really

- Why was that funny?
- Yes - software quality is not what it should be
- BUT - we have already achieved outstanding successes. Software is pervasive and works remarkably well much of the time
- We have seen humorous comparisons (usually with cars) that present both sides of software progress:
  - cars could be faster than a jet plane, smaller than a bicycle and cost $2000 (new)
  - changing the tires on a car would require removing the engine (temporarily) - occasionally after such a change, the car may refuse to turn right

Engineering safety-critical software

- I’m going to tell you a story - not a fairy story
- A true story - maybe a legend even
- Embellished only slightly

- And - it’s a story with a happy ending

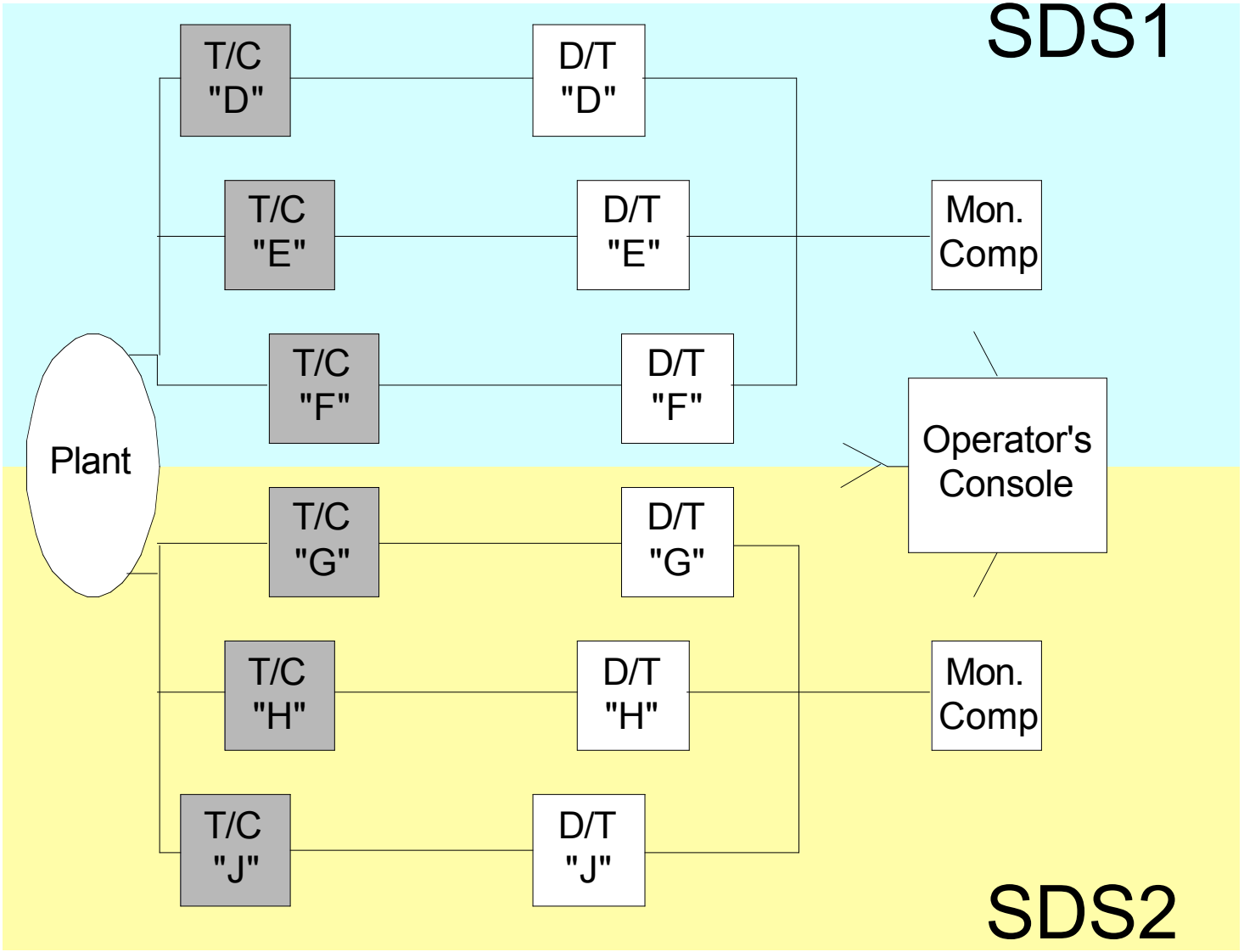
Shutdown System Context

Darlington Shutdown Systems

T/C = Trip Computer

D/T = Display/Test Computer

Mon. Comp = Monitor Computer



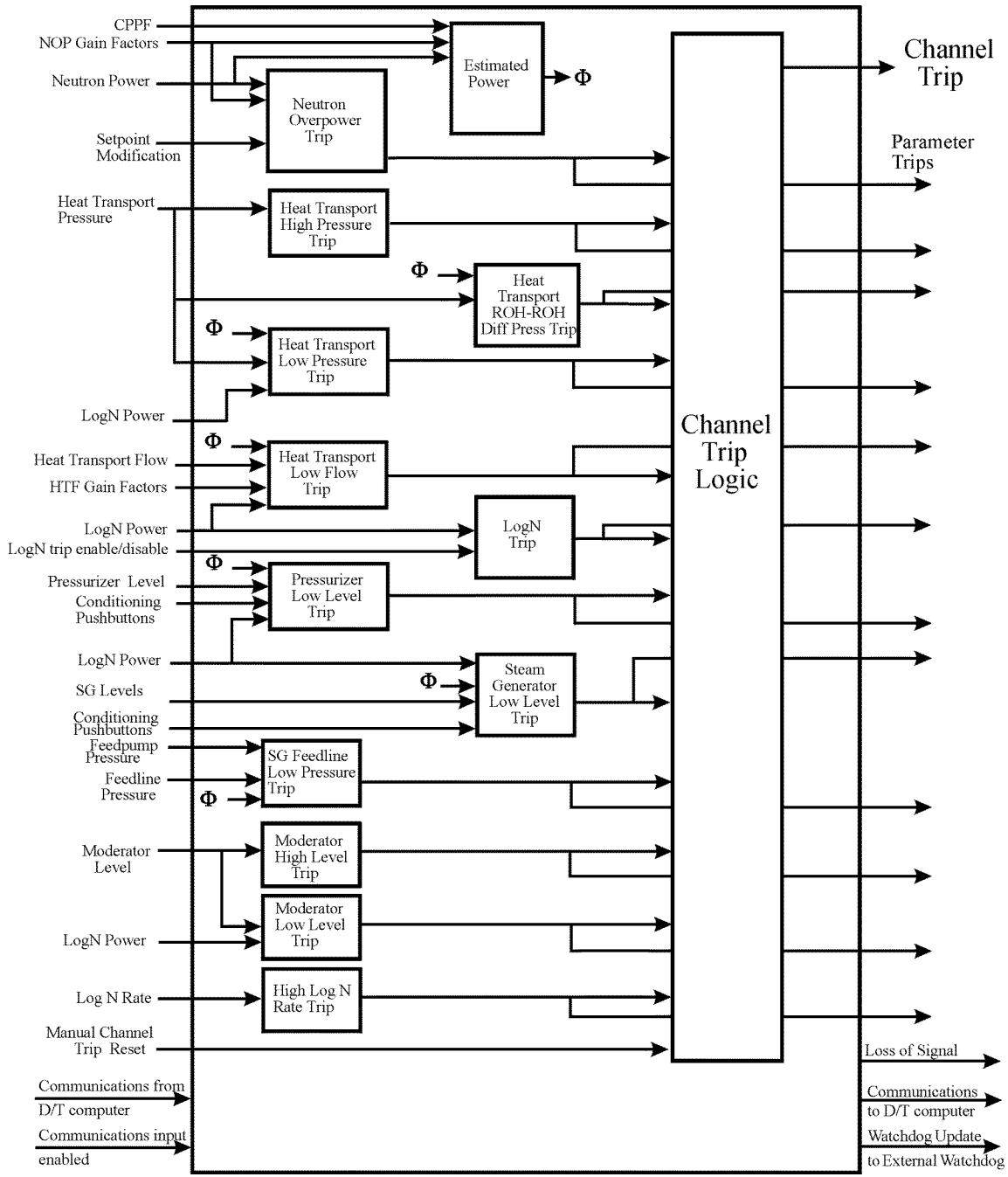
Software Size

Shutdown System One

A single trip computer

60 modules
280 access programs
40,000 lines of code
(including comments)
- 33,000 FORTRAN
- 7,000 assembler

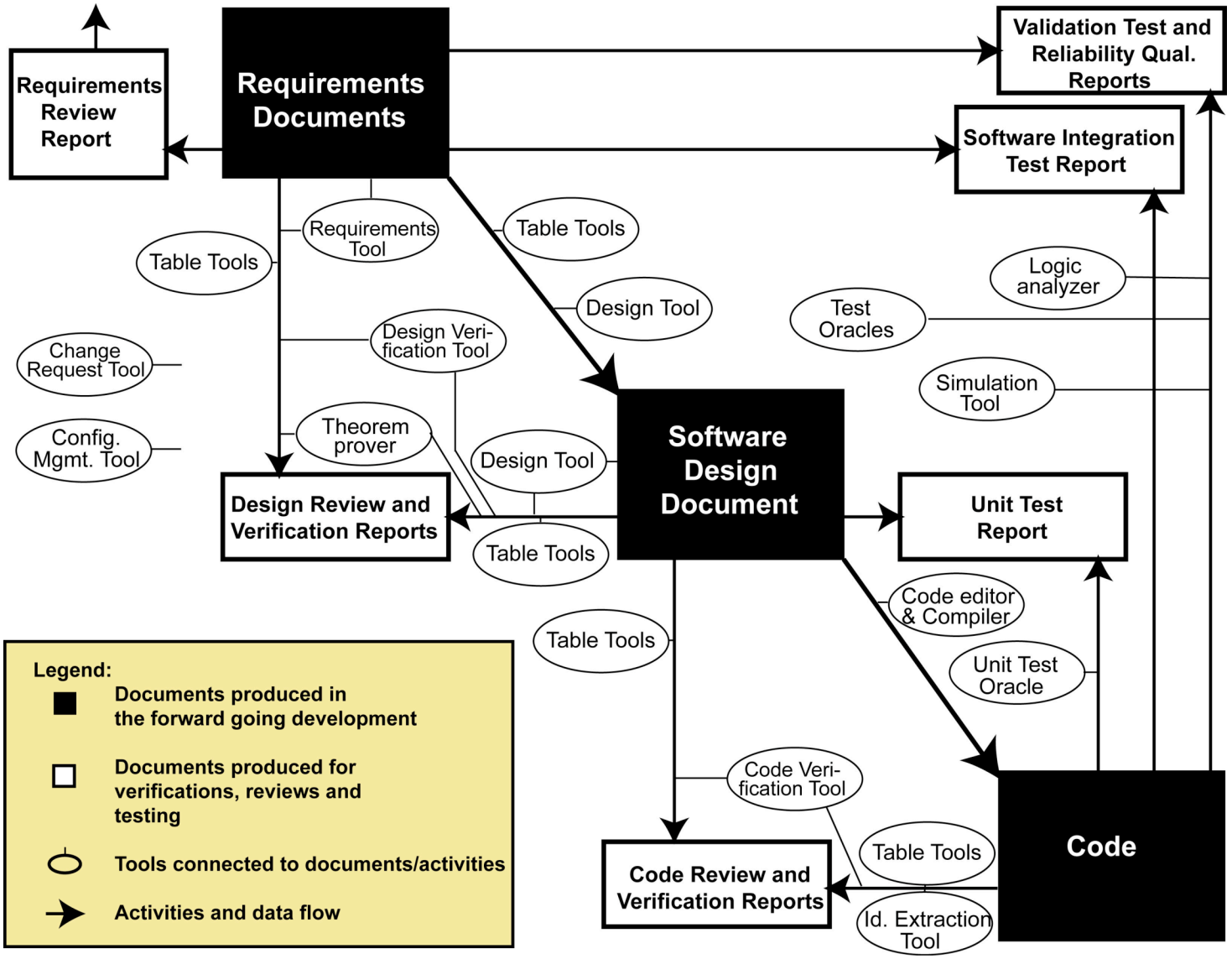
84 monitored variables
27 controlled variables



Notation

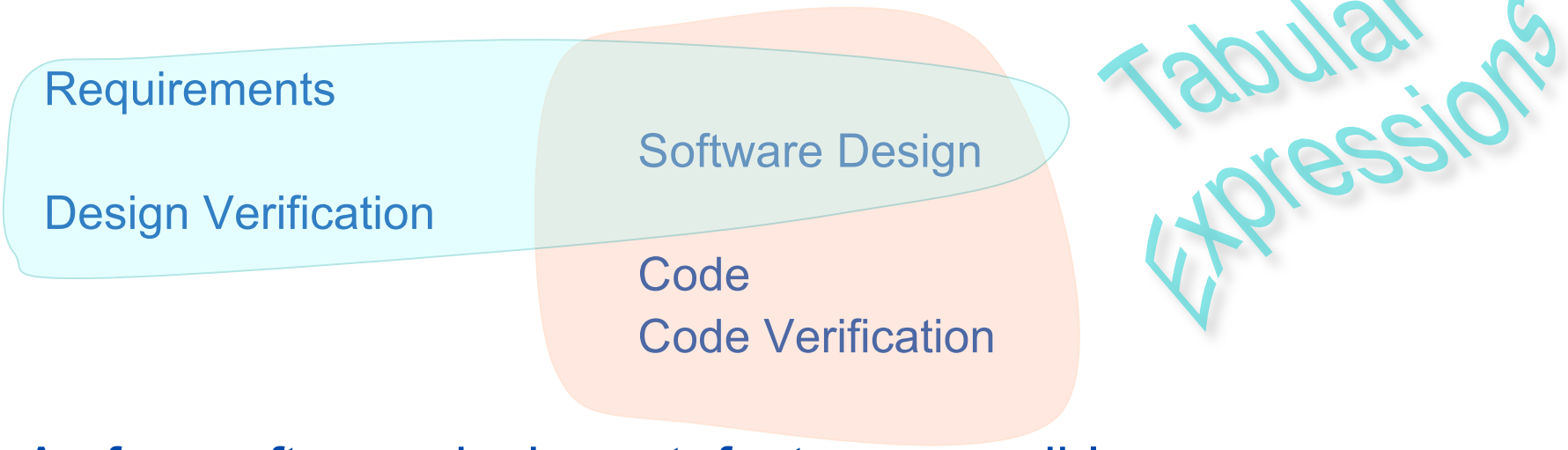
- *m_ name* is a monitored variable, *c_name* is a controlled variable, *k_name* is a constant, *e_name* is an enumerated token in a type, etc.
- *m_name* represents current value of the variable
- m_name_{-1} represents "previous" value of *m_name*

Software Development Lifecycle



Requirements

- Mathematically based
- A priority was that domain experts would be able to read and understand all the details
- Integrated:



- As few software design artefacts as possible
- Practical - must deal with real industrial applications

Tabular Expressions

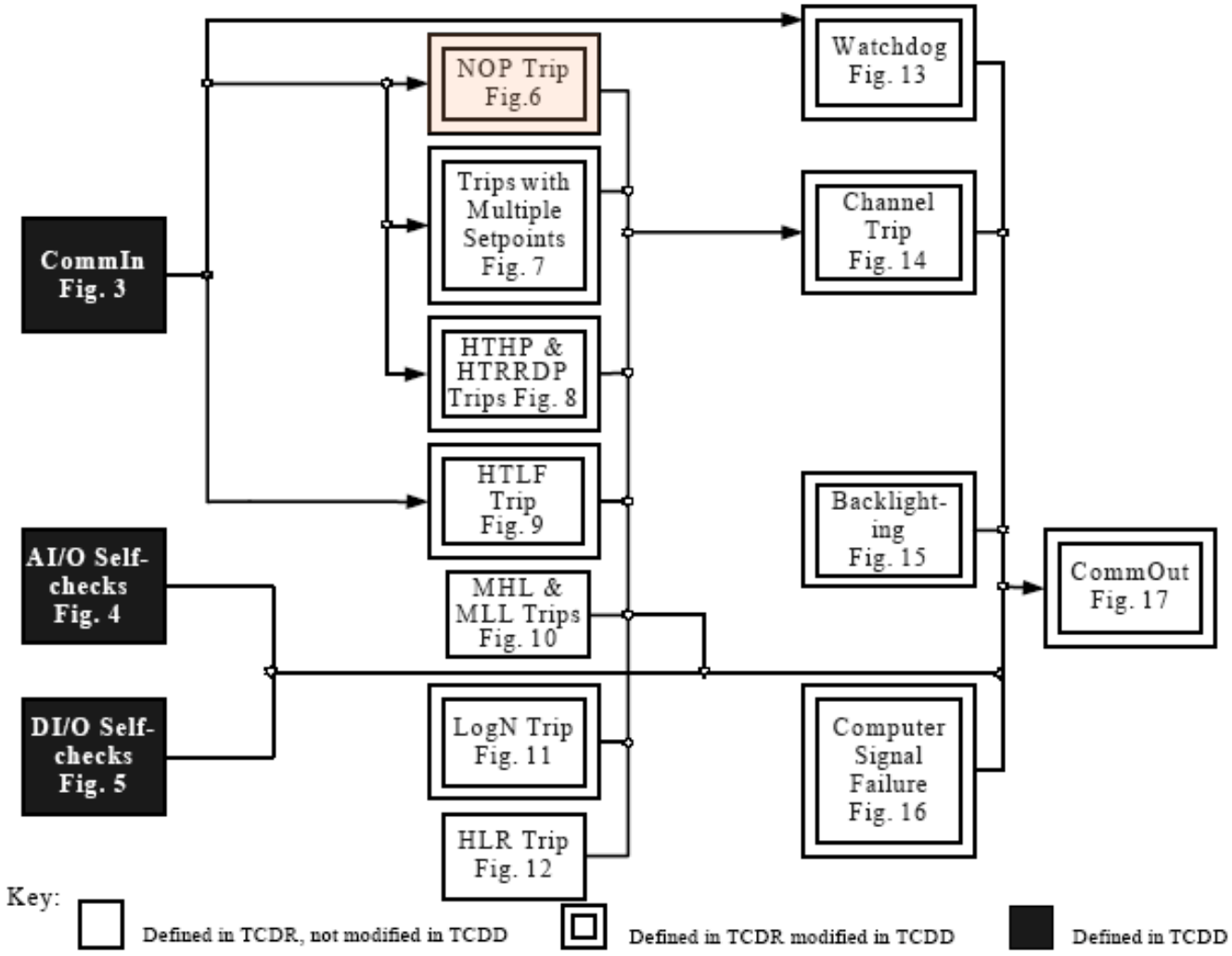
Software Quality Research Laboratory

Condition	Result: f_name
Condition 1	res 1
Condition 2	res 2
... ...	...
Condition n	res n

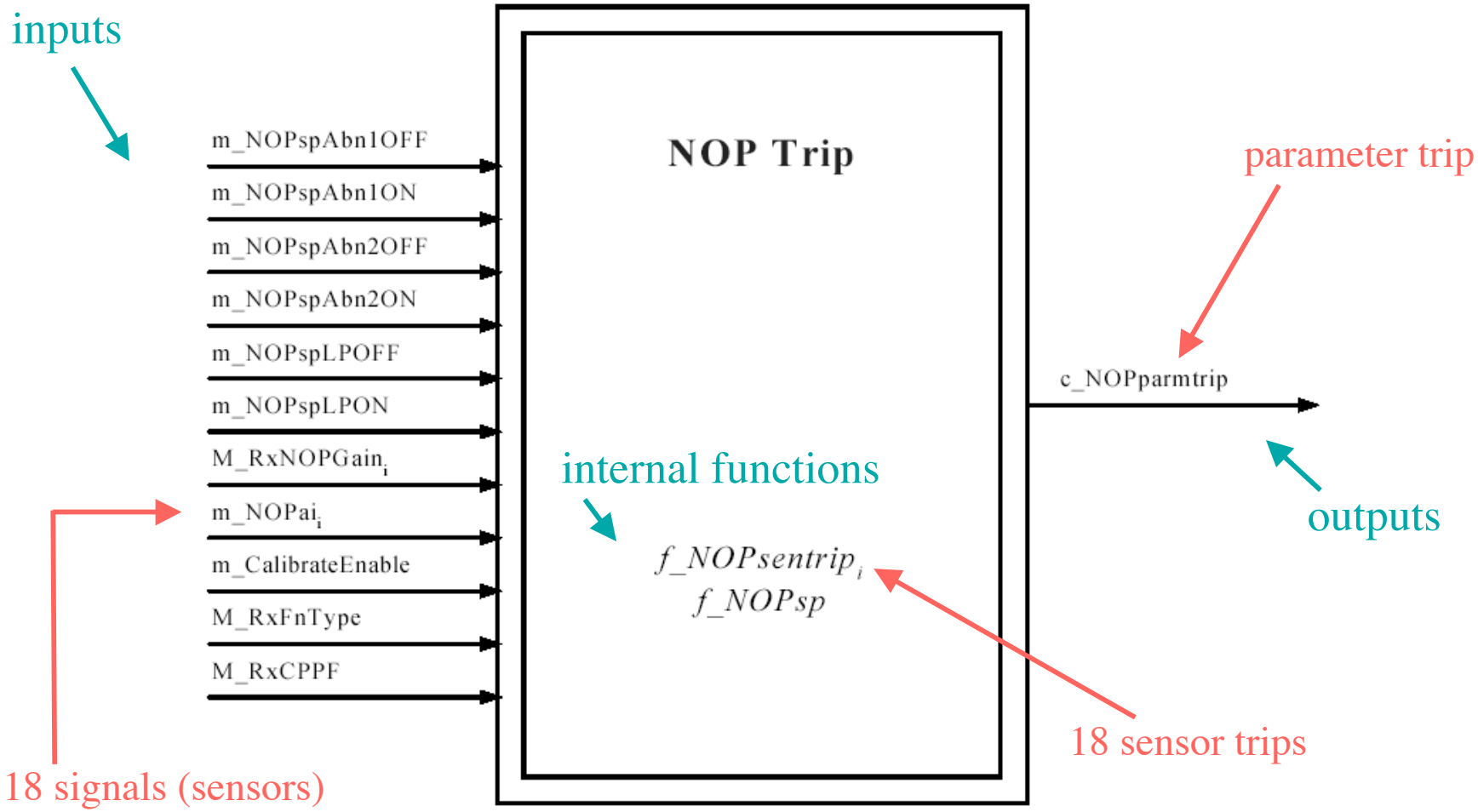
If Condition 1 then f_name = res 1
Elseif Condition 2 then f_name = res 2
Elseif …
Elseif Condition n then f_name = res n

and Condition i ∧ Condition j ⇔ FALSE ∀i,j = 1,..,n, i ≠ j
Condition 1 ∨ Condition 2 ∨ … ∨ Condition n ⇔ TRUE

Requirements - SDS1



Requirements - NOP Trip 1



Requirements Model

- Finite State Machine (FSM) Model
  - C(t) - vector of values of controlled variables at time t
  - M(t) - vector of values of monitored variables at time t
  - S(t) - vector of values of state variables at time t
  - t_0 - time of initialisation
  - $S(t_0)$ must be known

$C(t_k) = R(M(t_k), S(t_k))$

$S(t_{k+1}) = NS(M(t_k), S(t_k)),$ $k=0,1,2,3,...$

Requirements - NOP Trip 2

Software Quality Research Laboratory

2.1.3.9.1 Neutron Overpower Parameter Trip

2.1.3.9.1.1 Inputs/Natural Language Expressions

Input	NL Expression	Reference
$f\_NOPsentrip_i$, i=1,..,18	-	2.1.3.9.2.4

2.1.3.9.1.2 c_NOPparmtrip

Condition	*Result* **c_NOPparmtrip**
Any ($i \in 1,..,18$) ($f\_NOPsentrip_i$ = e_Trip) *{Any NOP sensor is tripped}*	e_Trip
All (i = 1,..,18) ($f\_NOPsentrip_i$ = e_NotTrip) *{All NOP sensors are not tripped}*	e_NotTrip

Requirements - NOP Trip 3

2.1.3.9.2 Neutron Overpower Sensor Trips

Determines whether there is an NOP sensor trip, which is used to determine whether there is an associated parameter trip.

2.1.3.9.2.1 Inputs/Natural Language Expressions

Input	NL Expression	Reference
f_NOPsp	-	2.1.3.9.3.3
$f\_NOPGain_i$, i=1,..,18, k_CalNOPHiLimit, k_CalNOPLoLimit, k_NOPOffset, m_NOPai_i, i=1,..,18	Calibrated i^{th} NOP signal, i=1,..,18	2.1.4.12

2.1.3.9.2.4 $f\_NOPsentrip_i$, i=1,..,18

{For each i = 1,..,18}

Condition	*Result* **$f\_NOPsentrip_i$**
f_NOPsp ≤ Calibrated i^{th} NOP signal *{Calibrated NOP $signal_i$ is now in the trip region}*	e_Trip
f_NOPsp - k_NOPhys < Calibrated i^{th} NOP signal < f_NOPsp *{Calibrated NOP $signal_i$ is now in the deadband region}*	No Change
Calibrated i^{th} NOP signal ≤ f_NOPsp - k_NOPhys *{Calibrated NOP $signal_i$ is now in the non-trip region}*	e_NotTrip

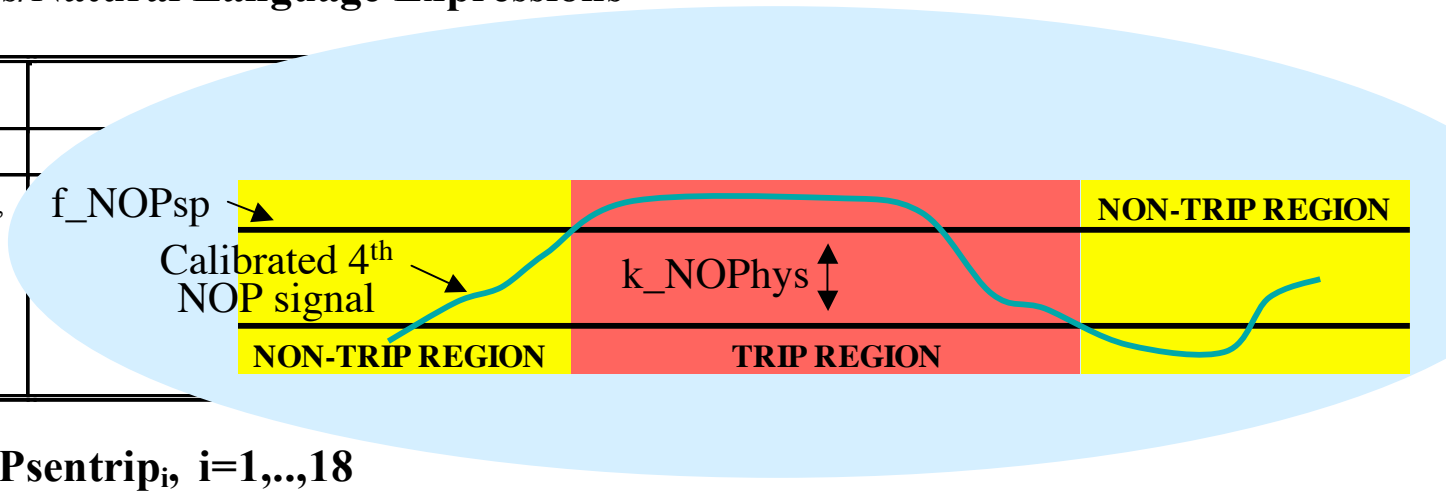
Requirements - NOP Trip 3

2.1.3.9.2 Neutron Overpower Sensor Trips

Determines whether there is an NOP sensor trip, which is used to determine whether there is an associated parameter trip.

2.1.3.9.2.1 Inputs/Natural Language Expressions

Input
f_NOPsp
$f\_NOPGain_i$, i=1,..,18, k_CalNOPHiLimit, k_CalNOPLoLimit, k_NOPOffset, m_NOPai_i, i=1,..,18



2.1.3.9.2.4 $f\_NOPsentrip_i$, i=1,..,18

{For each i = 1,..,18}

Condition	*Result* **$f\_NOPsentrip_i$**
f_NOPsp ≤ Calibrated i^{th} NOP signal *{Calibrated NOP $signal_i$ is now in the trip region}*	e_Trip
f_NOPsp - k_NOPhys < Calibrated i^{th} NOP signal < f_NOPsp *{Calibrated NOP $signal_i$ is now in the deadband region}*	No Change
Calibrated i^{th} NOP signal ≤ f_NOPsp - k_NOPhys *{Calibrated NOP $signal_i$ is now in the non-trip region}*	e_NotTrip

Requirements - NOP Trip 4

2.1.3.9.3.3 f_NOPsp

Condition	*Result* **f_NOPsp**
NOP Low Power setpoint is requested	k_NOPLPsp
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is requested	k_NOPAbn2sp
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is cancelled & NOP Abnormal 1 setpoint is requested	k_NOPAbn1sp
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is cancelled & NOP Abnormal 1 setpoint is cancelled	k_NOPnormsp

and

$k\_NOPLPsp \leq k\_NOPAbn2sp \leq k\_NOPAbn1sp \leq k\_NOPnormsp$

Requirements - NOP Trip 5

2.1.4.40.4 Definition

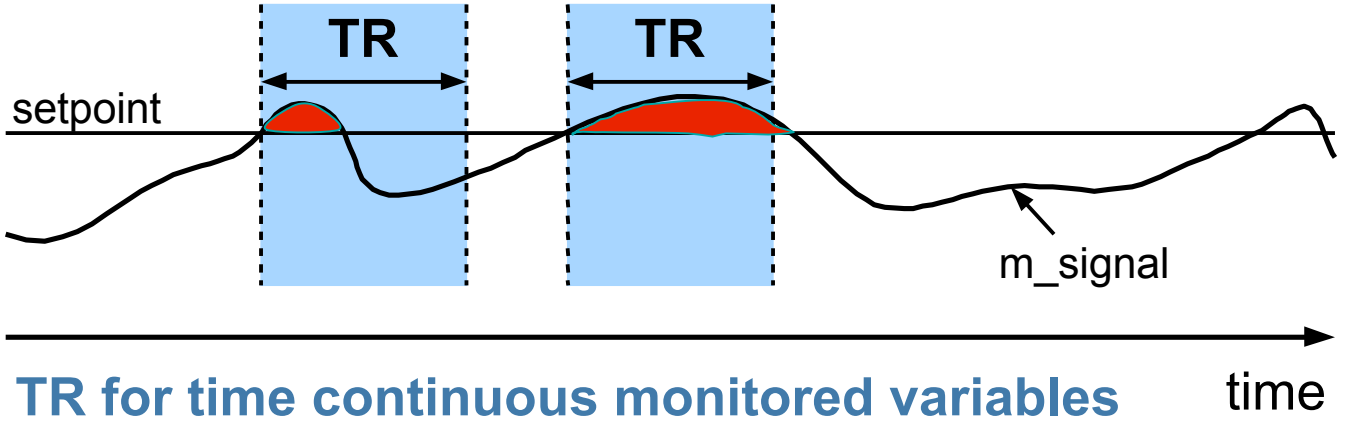
Condition	*Result* **NOP Abnormal 1 setpoint is requested or cancelled**
f_NOPspAbn1ON = e_pbStuck OR f_NOPspAbn1OFF = e_pbStuck	requested
f_NOPspAbn1ON = e_pbNotDebounced & f_NOPspAbn1OFF = e_pbNotDebounced	No Change
f_NOPspAbn1ON = e_pbNotDebounced & f_NOPspAbn1OFF = e_pbDebounced	cancelled
f_NOPspAbn1ON = e_pbDebounced & f_NOPspAbn1OFF = e_pbNotDebounced	requested
f_NOPspAbn1ON = e_pbDebounced & f_NOPspAbn1OFF = e_pbDebounced	requested

Tolerances

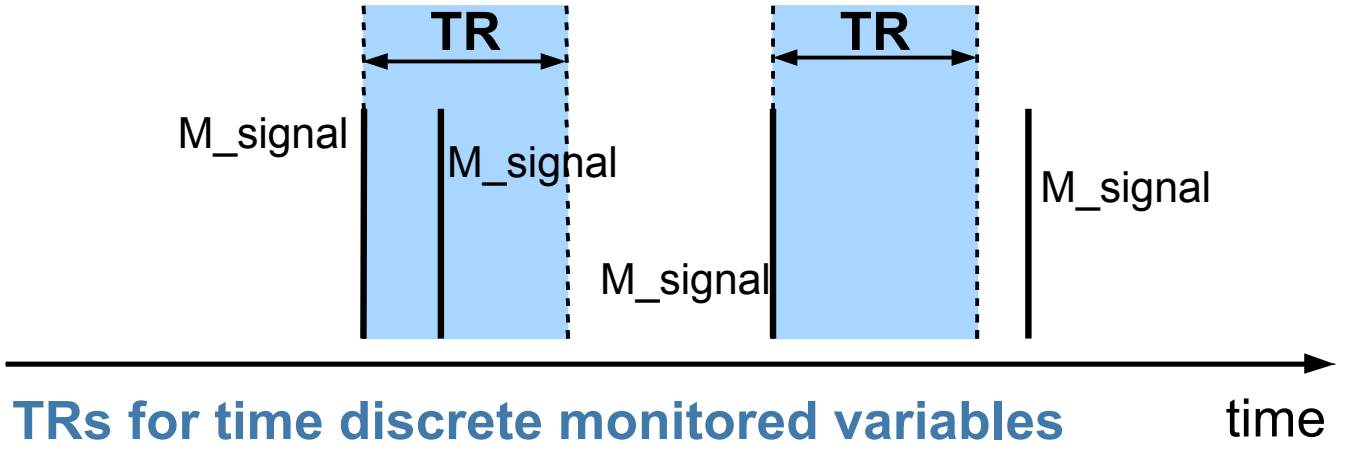
- Tolerances & Timing Requirements
  - Requirements model/document describes behaviour of an idealised system - impossible to meet
  - Need tolerances in general
  - Need timing tolerances in particular

Timing Resolution

An event that lasts for TR or longer must be detected

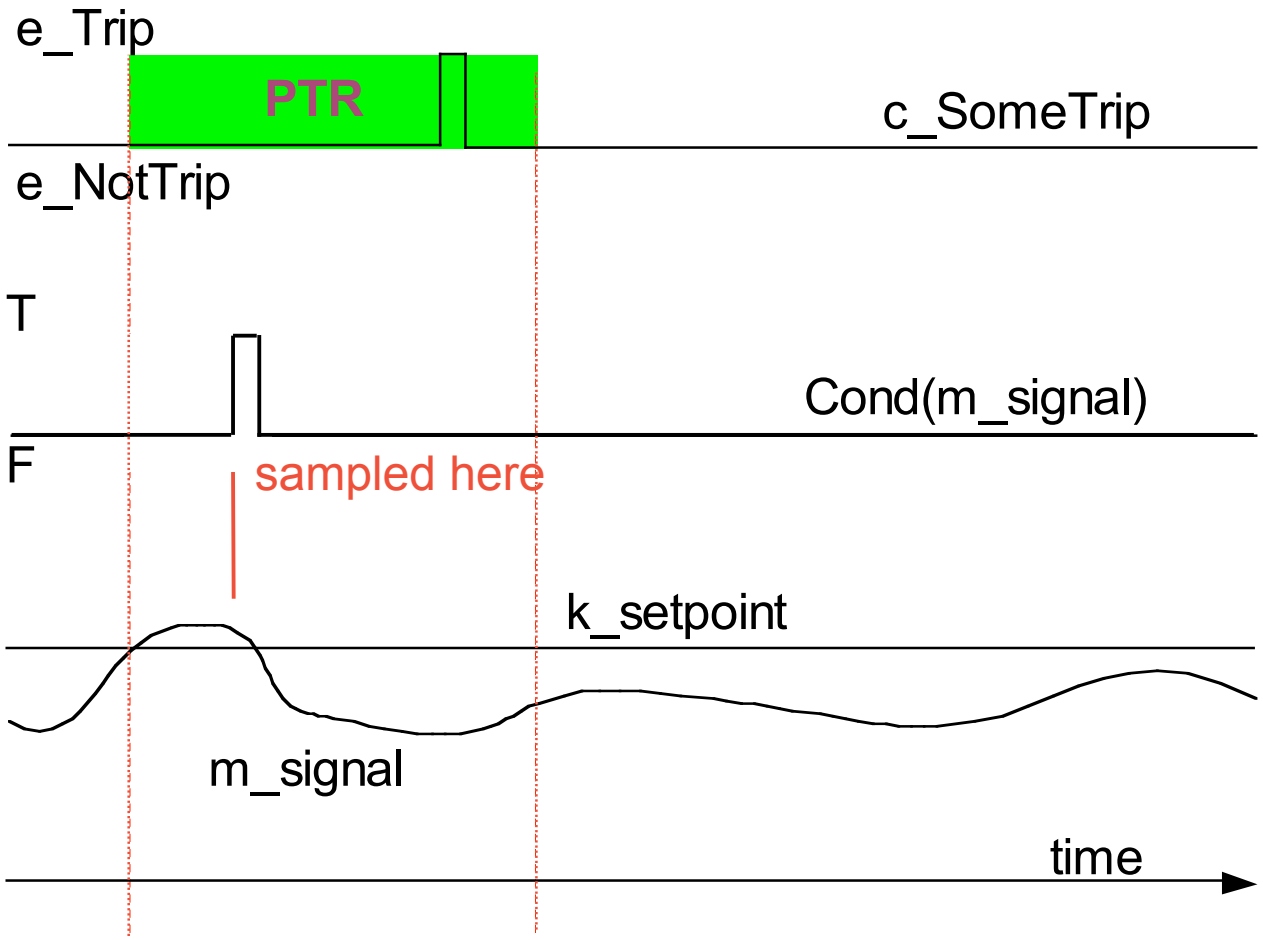


Two events at least TR apart must both be detected



Performance Timing Requirements

- The PTR for a m-c pair specifies an allowable processing delay
- The PTR is measured from the time that the value of m crosses the system boundary until the time that the value of c crosses the system boundary



Timing Issues

- My main research interest is to cope with timing issues
  - What if time constants have tolerances?
  - How do performance timing issues and functional timing issues interact?
  - Unfortunately, it appears that existing theory does not cope adequately with these problems!

Lesson 1

- Precision of mathematical requirements acted as a catalyst for probing questions from domain experts at an early stage of development.

Lesson 2

SQRL
Software Quality Research Laboratory

- Mathematical precision builds a false sense of confidence.
  - Differences in the interpretation of the model can lead different readers of the requirements to different interpretations of the requirements!
  - The math may be precise, but readers may still read it incorrectly.

Software Design

- The software design is decomposed for different reasons compared with the requirements
- Decompose for maintainability as a primary concern - Information Hiding!
- “secrets” are identified from changes that are predicted to be likely to occur. Each secret is hidden in a module, so the implementation of the module can be changed without affecting other modules

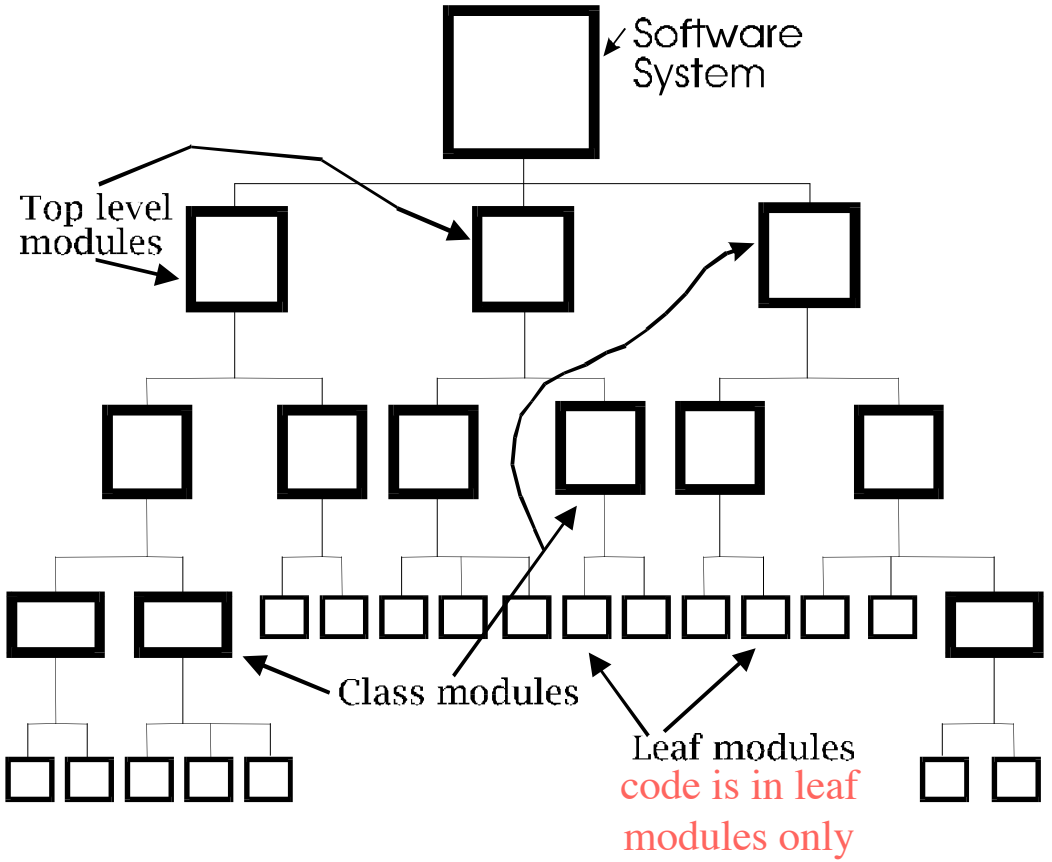
Parnas, D.: On the criteria to be used in decomposing systems into modules. Communications of the ACM December (1972), 1053-1058

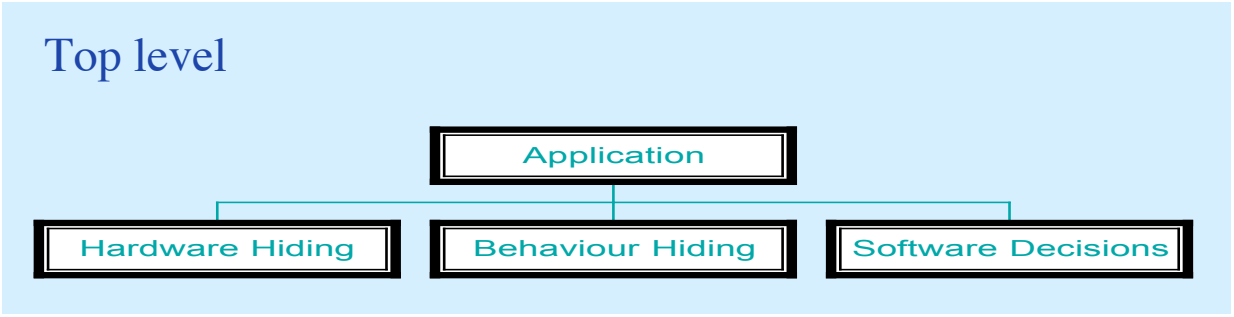
Modules

A module is a conceptual design unit in the decomposition of software functionality.

Modules can be developed in a hierarchical structure where one module can “contain” other modules.

Leaf modules communicate through externally visible access programs.

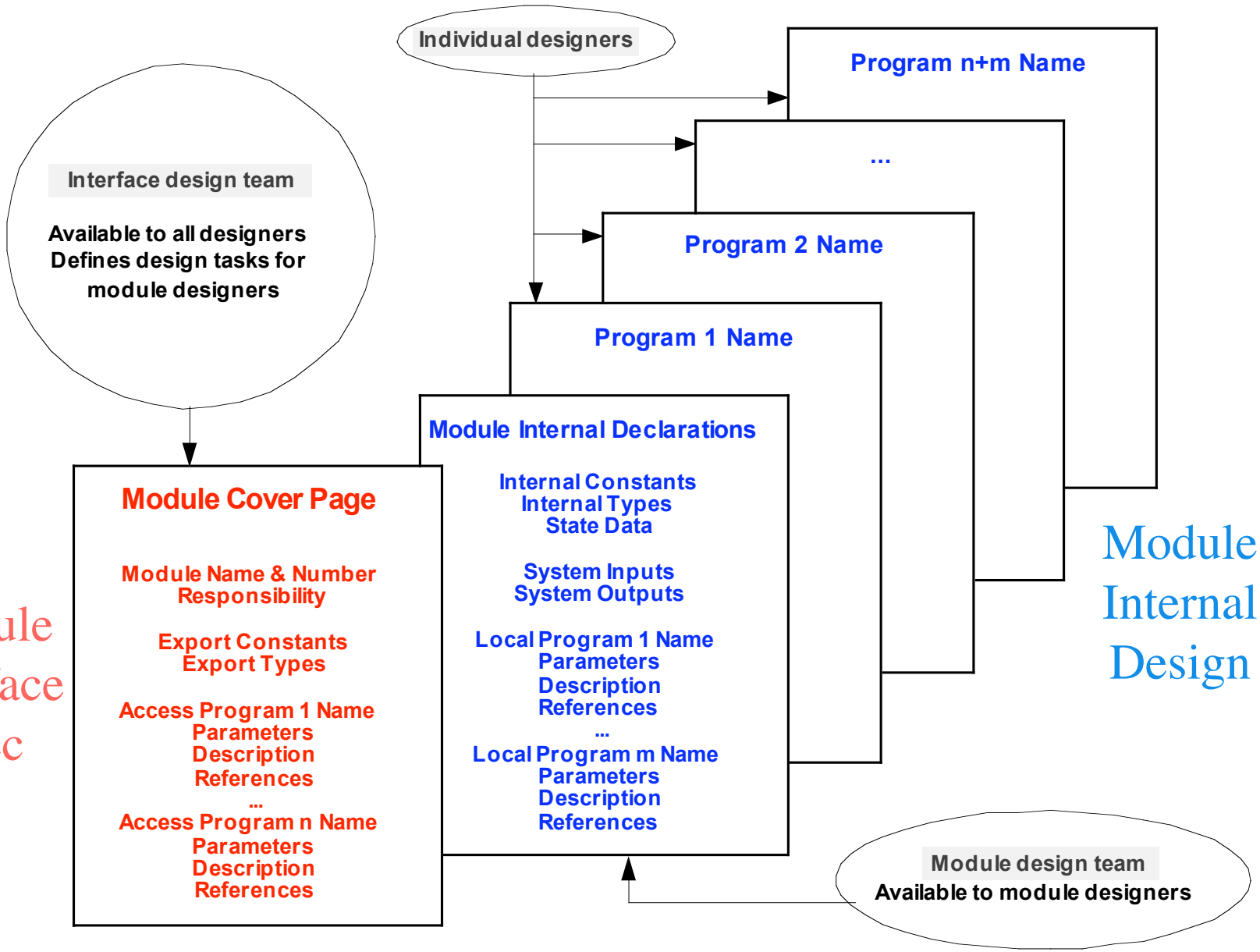




“Rational Design Process”

- Parnas talked about a “rational design process” - we set out to implement such a process
- Major components
  - Module Guide
    - For each module - secret, responsibility
  - Module Interface Specification
    - External description of module interface behaviour
  - Module Internal Design
    - Details of how each module implements its interface specification

Documentation of Modules

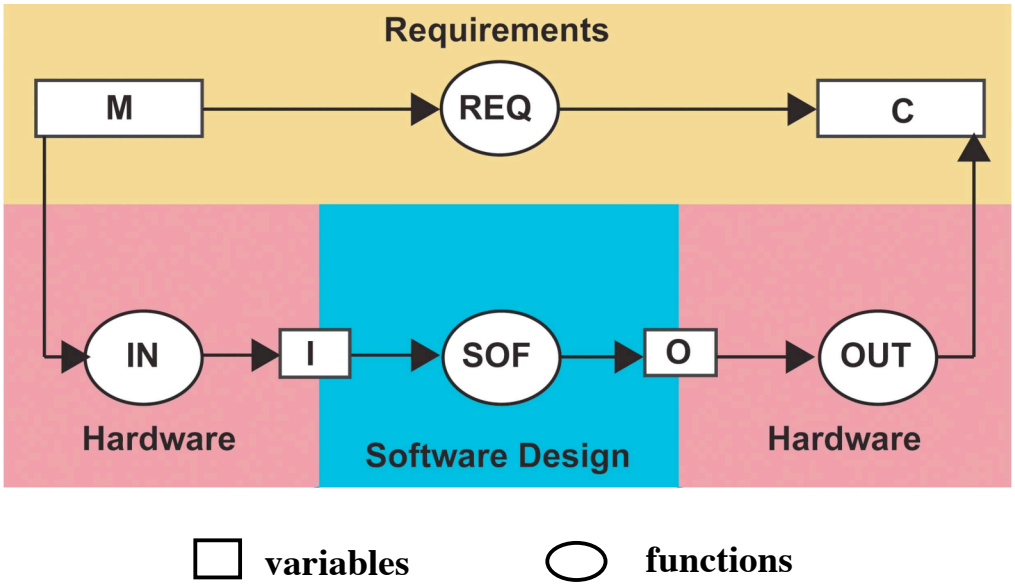


Module Interface Specification

- Traditionally, researchers have looked for ways of making this specification as “black-box” as possible, and have approached the problem without thinking enough about the previous phase - the requirements
- If we assume we have mathematical requirements, then those requirements should be useful for the MIS
- Problems:
  - Software variables are different from domain variables
  - Software design decomposition is different from requirements decomposition
- So - try and solve those problems and then we have a more integrated methodology!

4 Variable Model

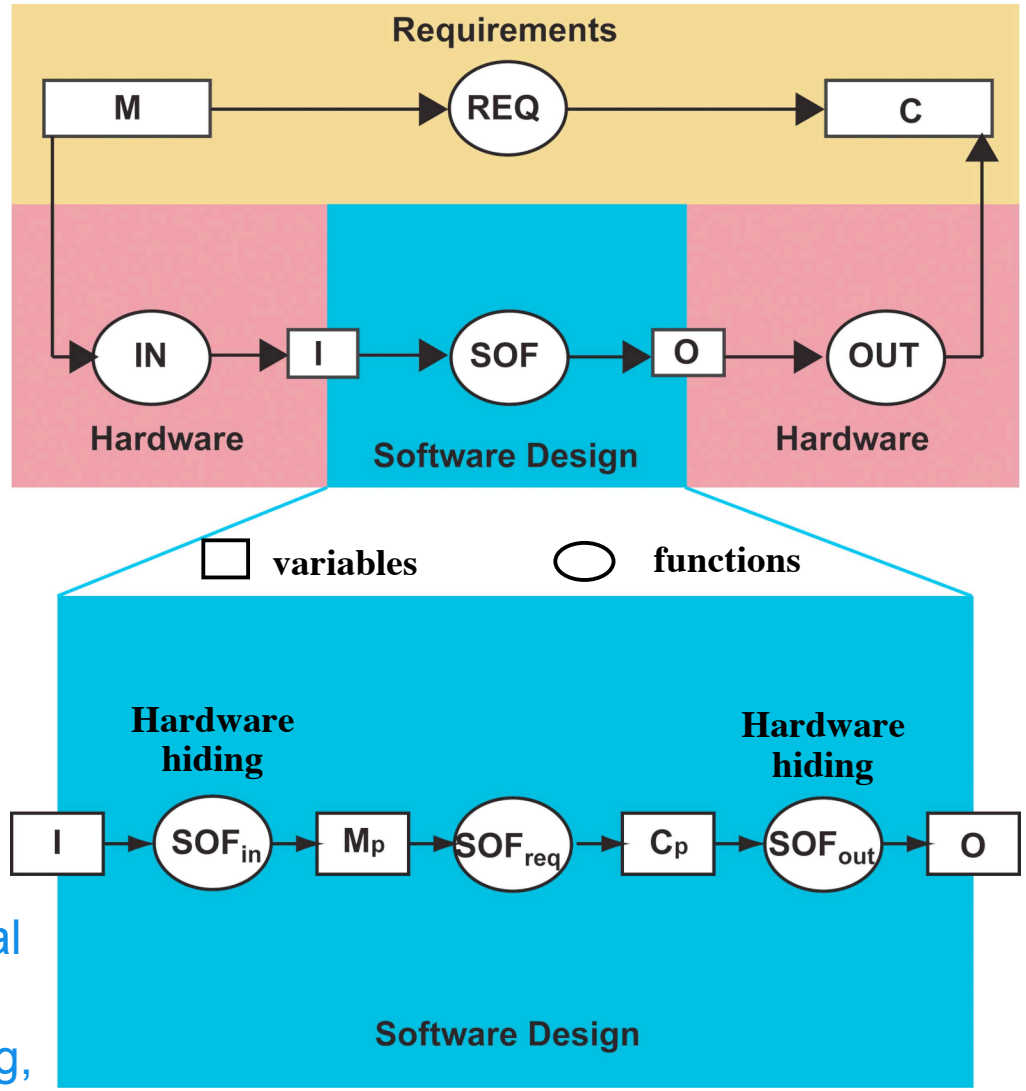
- Are the software variables really different from the domain variables as far as the MIS is concerned?



Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming, 25, (1995), 41-61

Modified 4 Variable Model

- Are the software variables really different from the domain variables as far as the MIS is concerned?
- Not if we look at the modified 4 variable model - which is what developers normally choose to implement anyway



Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming, 25, (1995), 41-61

MIS Example

Software Quality Research Laboratory

n.m MODULE Watchdog

Determines the watchdog system output.

	Name	Value	Type
Constants:	(None)		

	Name	Definition
Types:	(None)	

Access Programs:

EWDOG

Updates the state of the watchdog timer Digital Output.

References: *c_Watchdog*, 'Watchdog test active'.*

IWDOG

Initializes all the Watchdog module internal states and sets the initial watchdog output.

References: Initial Value, Initialization Requirements.

SWDOG

NCPARM: `t_boolean` - in

Signals to Watchdog module that a valid watchdog test request is received if NCPARM = $TRUE. Note that NCPARM is a "Conditional Output Call Argument"; calling the program with NCPARM = $FALSE has no effects on the module.

References: 'Watchdog test active'. *

Provide spec for access program

Needs to be more specific

MIS Example - NOP

4.23 MODULE NPParTrip

Provides the current NOP parameter trip status to drive the NOP parameter trip output.

	Name	Value	Type
Constants:	(None)		

	Name	Definition
Types:	(None)	

Access Programs:

EPTNP

Determines the current NOP parameter trip status and posts the parameter trip output state to DigitalOutput module.
References: *c _ N OPparmtrip*

GPTSNP

return: t_boolean
Returns the current NOP parameter trip status. A return value of $TRUE or $FALSE indicates that the parameter is tripped or not tripped respectively.
References: *c _ N OPparmtrip*

IPTNP

Initializes all the NPParTrip module internal states.
References: *Initial Value: NPParTrip*

Module "cover page" (MIS) shows external behaviour only.

MID Example - NOP

ACCESS PROGRAM EPTNP

	Name	Ext_value	Type	Origin
Inputs:	l_ST	GSTNP(l_ST)	ARRAY 1 TO KNUMNP OF t_boolean	NPSnrTrip

	Name	Ext_value	Type	Origin
Updates:	(None)			

	Name	Ext_value	Type	Origin
Outputs:	l_TrpDO	SDONP(l_TrpDO)	t_boolean	DigitalOutput
	PTSNP	-	t_boolean	State

Local Terms:

l_NoSTrp	(ALL i=1..KNUMNP)(l_ST[i] = $FALSE)

safe state
if there is one

VCT: EPTNP

	l_NoSTrp	NOT(l_NoSTrp)
l_TrpDO	$FALSE	$TRUE
PTSNP	$FALSE	$TRUE

Software Design Verification

- Assumes we use “supplementary function tables” in the design documentation
  - Verify software design against pseudo-requirements - have same data flow topology
  - Verify pseudo-requirements against requirements - only need to verify those blocks that are different

Software Design Verification

Proof obligation (general form)

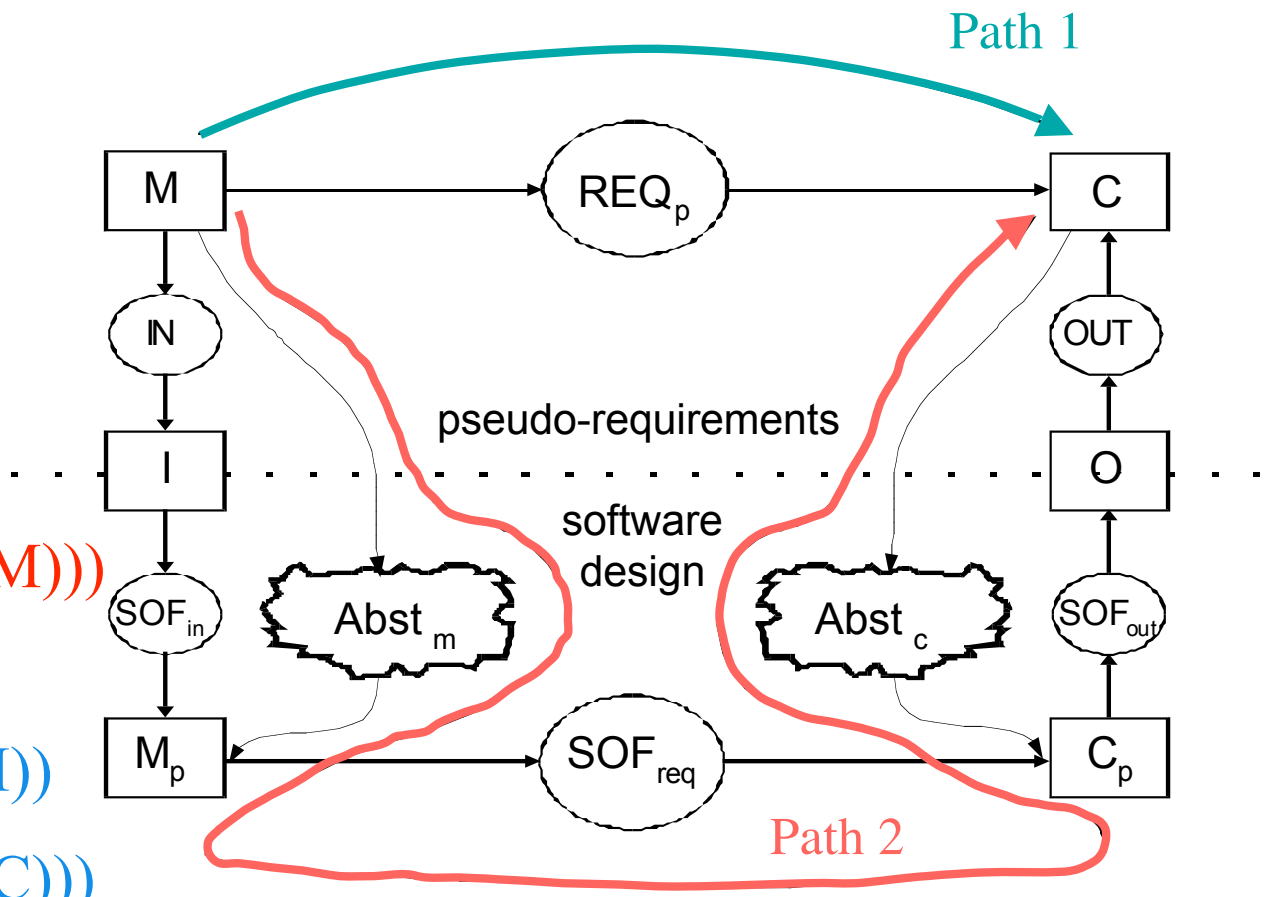
Result via path 1 must equal result via path 2

$REQ_p(M) = Abst_c^{-1}(SOF_{req}(Abst_m(M)))$

and

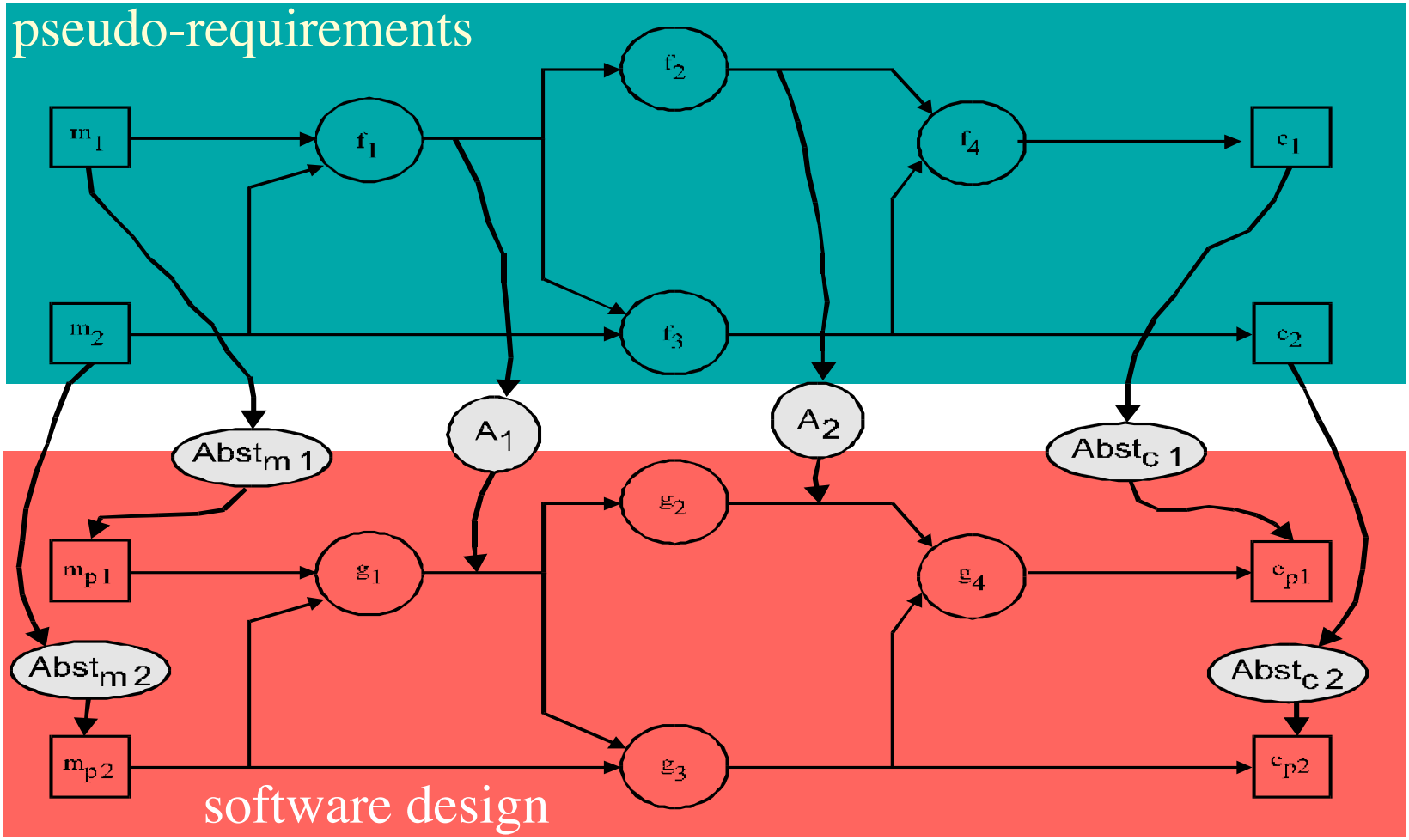
$Abst_m(M) = SOF_{in}(IN(M))$

$C = OUT(SOF_{out}(Abst_c(C)))$



Verification Abstract Example

Choose an arbitrary example:



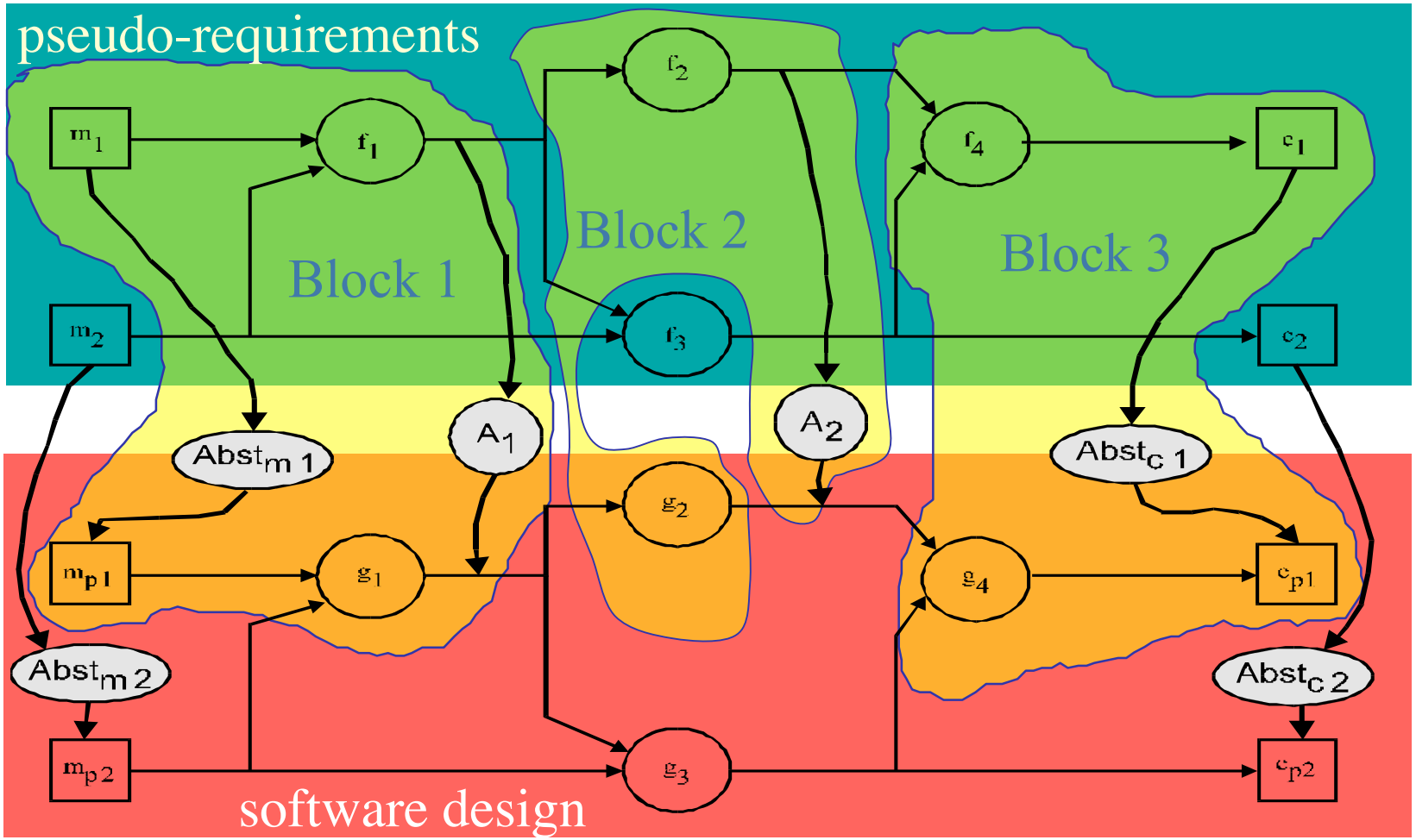
Example: Proof Obligation

The complete proof obligation for our example evaluates to:

$$f_4(f_2(f_1(m_1, m_2)), f_3(f_1(m_1, m_2), m_2)) =$$
$$Abst_{c\,1}^{-1}(g_4(g_2(g_1(Abst_{m\,1}(m_1), Abst_{m\,2}(m_2))),$$
$$g_3(g_1(Abst_{m\,1}(m_1), Abst_{m\,2}(m_2)), Abst_{m\,2}(m_2)))) \quad \ldots (1)$$

$$f_3(f_1(m_1, m_2), m_2) = Abst_{c\,2}^{-1}(g_3(g_1(Abst_{m\,1}(m_1), Abst_{m\,2}(m_2)), Abst_{m\,2}(m_2))) \quad \ldots (2)$$

Verification Piece-Wise

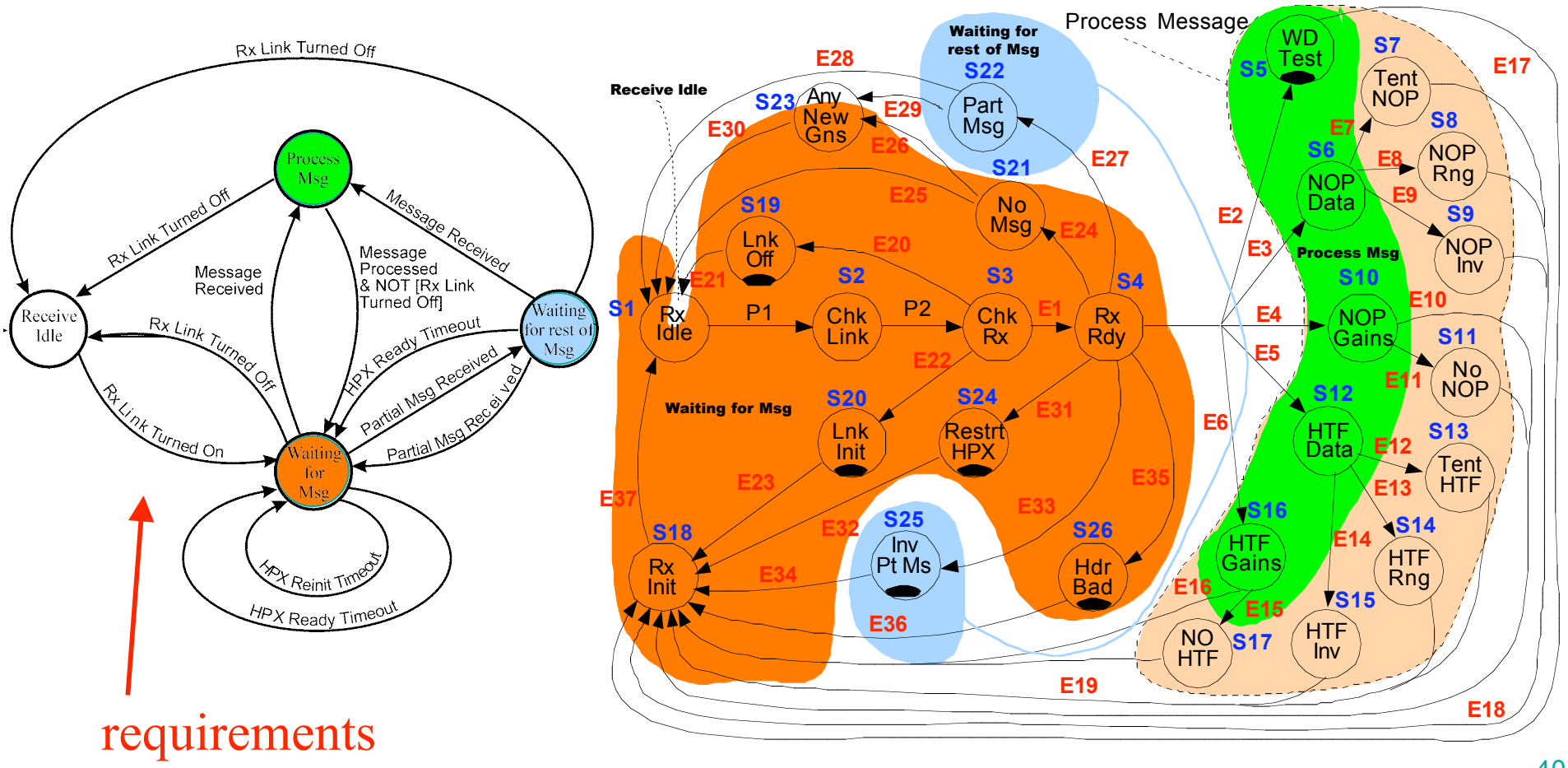


Block 4 (everything else)

Software Design Verification Process

- So, our procedure is to divide the software design and the pseudo-requirements into blocks such that each block in the design has identical inputs and outputs to a corresponding block in the pseudo-requirements
- Each block is verified individually, usually by manipulating function tables. We have also managed to implement more automated block verifications using PVS (so far, only relatively simple blocks)
- Afterwards we deal with the pseudo-requirements to requirements verification, and the monitored/controlled variable abstractions

Software Design Verification - Still Tough!



Coding

- Coding is pleasantly simple since each module internal design is very complete
- Most designs are in the form of tabular expressions (some are psuedo-code), and coding guidelines are quite easy to create and very effective in helping to produce well structured code

MID Example - NOP

ACCESS PROGRAM EPTNP

	Name	Ext_value	Type	Origin
Inputs:	l_ST	GSTNP(l_ST)	ARRAY 1 TO KNUMNP OF t_boolean	NPSnrTrip

	Name	Ext_value	Type	Origin
Updates:	(None)			

	Name	Ext_value	Type	Origin
Outputs:	l_TrpDO	SDONP(l_TrpDO)	t_boolean	DigitalOutput
	PTSNP	-	t_boolean	State

Local Terms:

l_NoSTrp	(ALL i=1..KNUMNP)(l_ST[i] = $FALSE)

safe state
if there is one

VCT: EPTNP

	l_NoSTrp	NOT(l_NoSTrp)
l_TrpDO	$FALSE	$TRUE
PTSNP	$FALSE	$TRUE

Coding Example - NOP


```
. . . . .
C     Variable initialization.
C     < SDD output: l_TrpDO >
      LLTRPD = $TRUE
      PTSNP = $TRUE
C --- < VCT EPTNP > ------------------------------------
      IF (.NOT. ((LLST(1) .EQ. $FALSE) .AND.
     +   (LLST(2)  .EQ. $FALSE) .AND. (LLST(3)  .EQ. $FALSE) .AND.
     +   (LLST(4)  .EQ. $FALSE) .AND. (LLST(5)  .EQ. $FALSE) .AND.
     +   (LLST(6)  .EQ. $FALSE) .AND. (LLST(7)  .EQ. $FALSE) .AND.
     +   (LLST(8)  .EQ. $FALSE) .AND. (LLST(9)  .EQ. $FALSE) .AND.
     +   (LLST(10) .EQ. $FALSE) .AND. (LLST(11) .EQ. $FALSE) .AND.
     +   (LLST(12) .EQ. $FALSE) .AND. (LLST(13) .EQ. $FALSE) .AND.
     +   (LLST(14) .EQ. $FALSE) .AND. (LLST(15) .EQ. $FALSE) .AND.
     +   (LLST(16) .EQ. $FALSE) .AND. (LLST(17) .EQ. $FALSE) .AND.
     +   (LLST(18) .EQ. $FALSE))) GO TO 20000
C     < l_NoSTrp >
C     See RANGE-CHECK NOTE (1.a)
C        < SDD output: l_TrpDO >
         LLTRPD = $FALSE
         PTSNP  = $FALSE
         GO TO 29999
```

comments are used to reference software design

all function tables are implemented in a consistent column order etc

Coding Example - NOP cont.


```
20000 CONTINUE
C     Range check on <l_ST>
C     See RANGE-CHECK NOTE (2.a)
      IF (.NOT.((LLST(1) .EQ. $TRUE) .OR.
     +    (LLST(2)  .EQ. $TRUE) .OR. (LLST(3)  .EQ. $TRUE) .OR.
     +    (LLST(4)  .EQ. $TRUE) .OR. (LLST(5)  .EQ. $TRUE) .OR.
     +    (LLST(6)  .EQ. $TRUE) .OR. (LLST(7)  .EQ. $TRUE) .OR.
     +    (LLST(8)  .EQ. $TRUE) .OR. (LLST(9)  .EQ. $TRUE) .OR.
     +    (LLST(10) .EQ. $TRUE) .OR. (LLST(11) .EQ. $TRUE) .OR.
     +    (LLST(12) .EQ. $TRUE) .OR. (LLST(13) .EQ. $TRUE) .OR.
     +    (LLST(14) .EQ. $TRUE) .OR. (LLST(15) .EQ. $TRUE) .OR.
     +    (LLST(16) .EQ. $TRUE) .OR. (LLST(17) .EQ. $TRUE) .OR.
     +    (LLST(18) .EQ. $TRUE))) GO TO 29998
C     < NOT(l_NoSTrp) >
         GO TO 29999
29998 CONTINUE
C     See RANGE-CHECK NOTE (2.b)
C        ErrorHdler.SFAT($FERNG, KPLN)
         CALL SFAT($FERNG, KPLN)
29999 CONTINUE
C --- < END VCT EPTNP > ----------------------------------
C     < SDD output call: DigitalOutput.SDONP(l_TrpDO)  >
      CALL SDONP(LLTRPD)
. . . . .
```

Code / Code Verification Linkage

- A couple of decisions were made in the coding procedure that make code verification much simpler
  - Comments in the code act as references to the software design, so the verifier knows if the code was produced from a table or from pseudo-code (crucial)
  - The coding procedure is very precise and quite prescriptive in how to produce code from tables and pseudo-code. This means the verifier has a good idea of how to produce a table or pseudo-code from the code so that it has a very similar format to the software design

Code Verification

- Process is quite simple
  - For each module, the verifier constructs documentation that is very similar to the software design, but does it by analysing the code without referring to the software design
  - Then the verifier compares the resulting documentation with the software design
- Coding verification was a pleasant surprise - much easier than anticipated

Testing

- Our philosophy is that no one method will detect all faults - so, even though we do all the reviews and mathematical verifications, we also do: unit testing, software integration testing, validation testing, trajectory-based random testing
- Function tables provide an excellent basis for test cases - they obviously describe boundaries very well

Lesson 3

SQRL
Software Quality Research Laboratory

- Software tool support for these methods is mandatory. The tools should work seamlessly over all the processes in the software development lifecycle
- Integrated tools will come from integrated methods - so first priority is to make the methods as integrated as possible
- Often, mundane repetitive tasks can be automated with a lot less effort than the more theoretically interesting tasks

Lesson 4

- Mathematicians have been known to make mistakes. We should not rely on the formality of our approaches to the extent that we throw away years of experience and so forego normal “best practices”, such as accepted coding principles for the language in use, and (manual) inspections

End of story

Software Quality Research Laboratory

- This story is important because:
  - It consumed 13 years of my life
  - We learned many valuable lessons (I only highlighted a very few) - valuable to many software developers
  - It is one of only a handful of **successful** safety-critical software developments in a real industrial setting

Why the world needs you

- Contrary to current enrollment numbers, there is a shortage of qualified software professionals
- You can help change the norm - less hacking more rigor and discipline. That does not (necessarily) discard agile methods
- Don’t accept development cycles that skimp on requirements, rationale, design - take pride in your work and responsibility for your professional conduct

What we expect of you

- As software ENGINEERS
  - **C**ompetency in your activities related to software
  - **A** disciplined, engineering approach to software development and maintenance
  - **R**esponsible
  - **E**thical
- For your own sakes - go out and do something you enjoy! Have fun!


Thanks