Safety-Critical Systems 4: Engineering of Embedded Software Systems

© Jan Bredereke
University of Bremen

WS 2002/03

0. Introduction

Topic of This Lecture

intersection of:

- engineering
- embedded systems
- software systems

Engineering

- the disciplined use of science, mathematics and technology to build useful artefacts
- engineers design by means of documentation
  - key step: design validation
  - maintenance requires good documentation

Embedded Systems

- definition: an embedded computer system is considered a module in some larger system
- some distinguishing characteristics:
  - designer not free to define interface
  - interface constraints may be strict and arbitrary, but we can't ignore them
  - interfaces will change during development

Examples of Embedded Systems

- computer in autonomous wheelchair

  constraints: devices
  sensor data
  physics of wheelchair

- telephone switching system

  constraints: other company's switches
  own older switches
  international standards
  telephone number rules

The Safety-Critical Systems Lecture Series

SCS1: Basic concepts - problems - methods - techniques (SoSe02)

SCS2: Management aspects - standards - V-Models - TQM - assessment - process improvement (SoSe01, SoSe03)

SCS3: Formal methods and tools - model checking - testing - partial verification - inspection techniques - case studies (WiSe01/02)

SCS4: Engineering of Embedded Software Systems

Overview of SCS4

1. *rigorous description* of requirements
   1.1 system requirements
   1.2 software requirements
   1.3 further issues
   1.4 tabular expressions
2. *what information* should be provided in computer system documentation?

3. *decomposition* into modules
   - 3.1 the criteria to be used in decomposing systems into modules
   - 3.2 structuring complex software with the module guide
   - 3.3 hierarchical software structures
   - 3.4 designing software for ease of extension and contraction
   - 3.5 design of abstract interfaces
4. *families* of systems
   - 4.1 motivation: maintenance problems in telephone switching
   - 4.2 families of programs
   - 4.3 families of requirements

Style of This Course

- lecture 2 SWS (Vorlesung)
  - “This is obvious, isn’t it?”
- seminar 2 SWS (Übung)
  - “Oops, applying it here is difficult!”

Web Page of Lecture

www.tzi.de/*agbs*/lehre/ws0203/scs4

available for download:

- slides
- assignments
- announcements
- links
- . . .

Text for Reading

- lecture based on a number of research papers
- references will be given during course
  - mostly, not online :-(
  - important ones available for copying from secretary

Mark (“Schein”)

- n assignments during term, $7 \leq n \leq 14$
- assignments can be solved in *groups of two*
- $n - 1$ assignments must be *handed in*
- *average of* $n - 1$ best marks must be $\geq 60\%$
- *oral exam (“Fachgespräch”)* at end of term
  - 15-20 min
  - in the groups of two
  - individual marks

1. Rigorous Description of Requirements

Text for Chapter 1

[PaMa95] Parnas, D. L. and Madey, J. *Functional documents for computer systems*. Sci. Comput. Programming **25**(1), 41–61 (Oct. 1995).

Four-variable model, structure of requirements documentation and software documentation.

[Pet00] Peters, D. K. *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster Univ., Hamilton, Canada (Jan. 2000).

Most current version of four-variable model and tabular notation. (Is also on testing).
Relevant: Chapters 1.1, 5, Appendix A

Additional Background for Chapter 1

[vSPM93] van Schouwen, A. J., Parnas, D. L., and Madey, J. *Documentation of requirements for computer systems*. In "IEEE Int'l. Symposium on Requirements Engineering – RE'93", pp. 198–207, San Diego, Calif., USA (4–6 Jan. 1993). IEEE Comp. Soc. Press.

Example for the four-variable approach (water level monitoring system).

[LaRö01] Lankenau, A. and Röfer, T. *The Bremen Autonomous Wheelchair – a versatile and safe mobility assistant*. IEEE Robotics and Automation Magazine, "Reinventing the Wheelchair" **7**(1), 29–37 (Mar. 2001).

General description of the Bremen autonomous wheelchair "Rolland".

The Role of Documentation in Computer System Design

- professional engineer:
  - makes plan on paper
  - analyses plan thoroughly
  - then builds system, using plan
- engineer revising the system:
  - understands system through old plan
  - changes plan
  - analyses plan thoroughly
  - then builds system, using plan

- Computer hardware is made this way.
- Computer software usually is not.
- But standard engineering practice can be applied, too.
- Documentation
  - as a design medium
  - input to analysis
  - input to testing
  - facilitates revision or replacement

Education of Engineers Can't Start Too Early. . .

from a text book on engineering:

title page

good example

1.1 System Requirements

How Can We Document System Requirements?

- identify the relevant environmental quantities
  - physical properties
    - temperatures
    - pressures
  - positions of switches
  - readings of user-visible displays
  - wishes of a human user
- represent them by mathematical variables

- define carefully the association
  of env. quantities and math. variables
- specify a relation on the math. variables

Functions of Time

- env. quantities q_i described by functions of time
- types of values of env. quantities: $q_i \in Q_i$
- environmental state function:
  $S : \mathbb{R} \to Q_1 \times Q_2 \times \ldots \times Q_n$
- set of possible env. states:
  $St =_{\text{df}} Q_1 \times Q_2 \times \ldots \times Q_n$

Example: Electronic Thermometer

$\longrightarrow$ blackboard. . .

Monitored vs. Controlled Quantities

- controlled quantities:
  their value may be required to be changed by the system
- monitored quantities:
  shall affect the system behaviour
- some quantities are both
- time: is a monitored quantity
  (in real-time systems)

- monitored state function:
  $\underset{\sim}{m}^t : \mathbb{R} \rightarrow Q_1 \times Q_2 \times \ldots \times Q_i, \qquad 1 \leq i \leq n$
- controlled state function:
  $\underset{\sim}{c}^t : \mathbb{R} \rightarrow Q_j \times Q_{j+1} \times \ldots \times Q_n, \qquad 1 \leq j \leq n$
- $j \leq i + 1$
- environmental state function: $(\underset{\sim}{m}^t, \underset{\sim}{c}^t)$
- set of all $\underset{\sim}{m}^t$: M
- set of all $\underset{\sim}{c}^t$: C
- "behaviour": an S for a single execution

The Relation NAT

- constraints on the environmental quantities
- constraints by nature, by previously installed systems
- is part of the requirements document
- validity is responsibility of customer

- $\mathrm{NAT} \subseteq \mathrm{M} \times \mathrm{C}$
- $\mathrm{domain}(\mathrm{NAT}) = \{\underset{\sim}{m}^t \mid \underset{\sim}{m}^t$ allowed by env. constraints$\}$
  - if $\underset{\sim}{m}^t \notin \mathrm{domain}(\mathrm{NAT})$ then designer may assume that these values never occur
- $\mathrm{range}(\mathrm{NAT}) = \{\underset{\sim}{c}^t \mid \underset{\sim}{c}^t$ allowed by env. constraints$\}$
  - if $\underset{\sim}{c}^t \notin \mathrm{range}(\mathrm{NAT})$ then system cannot make these values happen
- $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in \mathrm{NAT}$ iff environmental constraints allow that $\underset{\sim}{c}^t$ are controlled quantities if $\underset{\sim}{m}^t$ are monitored quantities
- NAT usually not a function
  - the system should have some choice

The Relation REQ

- further constraints on the environmental quantities
- constraints by system
- is part of the requirements document
- validity is responsibility of system designer

- $\mathrm{REQ} \subseteq \mathrm{M} \times \mathrm{C}$
- $\mathrm{domain(REQ)} \supseteq \mathrm{domain(NAT)}$
  $= \{\underset{\sim}{m}^t \mid \underset{\sim}{m}^t$ allowed by env. constraints$\}$
- $\mathrm{range(REQ)} = \{\underset{\sim}{c}^t \mid \underset{\sim}{c}^t$ allowed by correct system$\}$
- $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in \mathrm{REQ}$ iff system should permit that
  $\underset{\sim}{c}^t$ are controlled quantities if $\underset{\sim}{m}^t$ are monitored quantities
- REQ usually not a function
  - one can tolerate “small” errors in the values of controlled quantities

Contract

- REQ states what the system designer must provide
- NAT states what the customer must provide

Black-Box View

- the requirements document is entirely in terms of environmental quantities
- no reference to internal quantities
- no reference to internal state, only to the history of env. quantities
- $\Rightarrow$ no restriction on system designer

Specifying Behaviour

what's next?

- modes and mode classes
- conditions, events
- four-variable approach for system design and software requirements
- tabular notation

Modes and Mode Classes, Informally

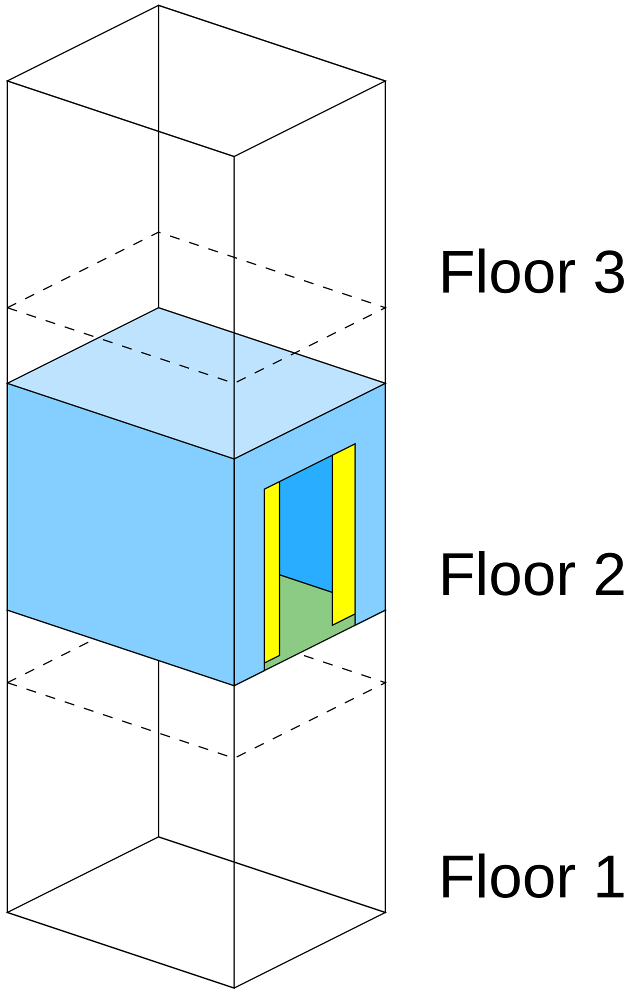
Definition 1 (Mode, informally)
An (environmental) mode is a set of (environmental) states.

Definition 2 (Mode Class, informally)
A mode class is a partitioning of the state space.

Discussion of Modes etc.

- there may be several mode classes
- system is always in one mode of every mode class
- mode class and its modes may be defined by a transition table
- one state change may imply two mode changes
  - no "simultaneous events"
- if time is monitored, the system never returns into the same state
  - modes are handy for equivalence classes of states

Example: Lift Controller



Lift Controller: Relevant Environmental Quantities

- height of lift
- elevation motor command
- position of doors
- door motor command
- (buttons left out for simplicity here)

Lift Controller: Environment Variables

Variable	**mon.**	**ctrl.**	**Description**	**Value Set**	**Unit**	**Notes**
^{m}height	•		height of lift	$\mathbb{R}$	m	1
^{c}elevMotorCommand		•	elevation motor command	$\{^{C}$up, ^{C}off, ^{C}down$\}$	—	
^{m}doorPos	•		position of doors	$\mathbb{R}$	m	2
^{c}elevMotorCommand		•	door motor command	$\{^{C}$open, ^{C}off, ^{C}close$\}$	—	

Notes

1. The height is relative to the lowest position physically possible, upward is positive.
2. This is how far the doors are opened. 0 m means entirely closed, positive means open.

Lift Controller: the Relation NAT

what to state rigorously (not done here):

- height is ≥ 0 m and $\leq$ max. height
- the acceleration and deceleration of the lift is bounded ($\rightarrow$ use differential equations)
- door position is ≥ 0 m and $\leq$ max. width
- ...

Conditions

Definition 3 (Condition)

A condition is a function $\mathbb{R} \rightarrow \mathbb{B}$,
defined in terms of the env. state function.
It is finitely variable on all intervals of system operation.

Definition 4 (Cnd)

Cnd is the tuple of all conditions.
We assume an order on the conditions.
We assume Cnd to be finite.

Lift Controller: Conditions

Name	Condition
^{p}doorClosed	^{m}doorPos $= 0$ m
^{p}at1stFloor	$\lvert {}^{m}$height $- 0.5$ m$\rvert \leq 1$ cm
^{p}at2ndFloor	$\lvert {}^{m}$height $- 4.5$ m$\rvert \leq 1$ cm
^{p}at3rdFloor	$\lvert {}^{m}$height $- 8.5$ m$\rvert \leq 1$ cm

Cnd $= ({}^{p}$doorClosed$, {}^{p}$at1stFloor$, {}^{p}$at2ndFloor$, {}^{p}$at3rdFloor$)$

Events

Definition 5 (Event)

An event e, is a pair, (t, c), *where*
$e.t \in \mathbb{R}$ *is a time at which one or more conditions change value and*
$e.c \in \{T, F, @T, @F\}^n$ *indicates the status of all conditions at* $e.t$, *as follows:*

$e.c[i]$	p_i
T	$`p_i(e.t) \wedge p_i{}'(e.t)$
F	$\neg `p_i(e.t) \wedge \neg p_i{}'(e.t)$
@T	$\neg `p_i(e.t) \wedge p_i{}'(e.t)$
@F	$`p_i(e.t) \wedge \neg p_i{}'(e.t)$

Event Space

Definition 6 (Event Space)

The event space is the set of all possible events:

$$EvSp \ =_{df} \mathbb{R} \times \{T, F, @T, @F\}^n$$

- any particular finite duration behaviour defines a finite set of events Ev $\subset$ EvSp

Lift Controller: Events

- $(5\ \mathsf{s}, (\mathsf{F}, @\mathsf{T}, \mathsf{F}, \mathsf{F}))$
- $(7\ \mathsf{s}, (@\mathsf{T}, \mathsf{T}, \mathsf{F}, \mathsf{F}))$
- $(20\ \mathsf{s}, (@\mathsf{F}, \mathsf{T}, \mathsf{F}, \mathsf{F}))$
- $(22\ \mathsf{s}, (\mathsf{F}, @\mathsf{F}, \mathsf{F}, \mathsf{F}))$
- $(29\ \mathsf{s}, (\mathsf{F}, \mathsf{F}, @\mathsf{T}, \mathsf{F}))$
- ...

History

- we are often interested in the values of the conditions for a specific interval of time
- a history is
  - the set of initial values for the conditions and
  - the sequence of events in the time interval
- (formal definition omitted here)

Modes and Mode Classes

Definition 7 (Mode Class)

A (environmental) mode class is an equivalence relation on possible histories, such that:
if $\mathrm{MC}(H_1, H_2)$ *and*
if $\hat{H}_1$ *and* $\hat{H}_2$ *are the extensions of* H_1 *and* H_2
by the same event,
then $\mathrm{MC}(\hat{H}_1, \hat{H}_2)$.

Definition 8 (Mode)

An (environmental) mode is one such equivalence class.

Lift Controller: Mode Classes

- some useful mode classes:
  - ^{Cl}door: ^{Md}doorClosed, ^{Md}doorOpen
  - ^{Cl}floor: ^{Md}in1stFloor, ^{Md}in2ndFloor, ^{Md}in3rdFloor
  - ^{Cl}atFloor: ^{Md}atAFloor, ^{Md}betweenFloors
- definition of mode classes:
  - through conditions
  - by transition tables

  (see later)

Tabular Notation

- tabular notations often useful to represent functions in computer system documentation
- extensive work on different table formats exists
- precise semantics has been defined for these table formats
- one format specifically for mode transition tables

Lift Controller: the Relation REQ

- conditions defined in terms of (monitored) variables
- event classes defined in terms of conditions
- mode classes defined in terms of event classes
- controlled variables defined in terms of mode classes

^{Cl}floor:

Mode	Event Class	New mode
^{Md}in1stFloor	@T(^{p}at2ndFloor)	^{Md}in2ndFloor
^{Md}in2ndFloor	@T(^{p}at1stFloor)	^{Md}in1stFloor
	@T(^{p}at3rdFloor)	^{Md}in3rdFloor
^{Md}in3rdFloor	@T(^{p}at2ndFloor)	^{Md}in2ndFloor

- the mode remains the same when between floors

"Simultaneous" Events

- modes of a mode class must be disjoint
- $\rightarrow$ event classes for one mode must be disjoint
- event expressions can comprise more than one event
- assume that causally independent changes of conditions never occur at exactly the same time ($t \in \mathbb{R}$)
- watch out for condition changes that are causally related!

Piecewise Continuous Behaviour

- often, environmental quantities have piecewise continuous behaviour over time
  - height of lift
  - position of lift door
- each continuous piece can be described well by a differential equation
- switching from piece to piece can be described well by mode changes

Lift Controller: Piecewise Continuous Behaviour

one of the constraints by NAT:

$\frac{d}{dt}{}^{m}$height $=$

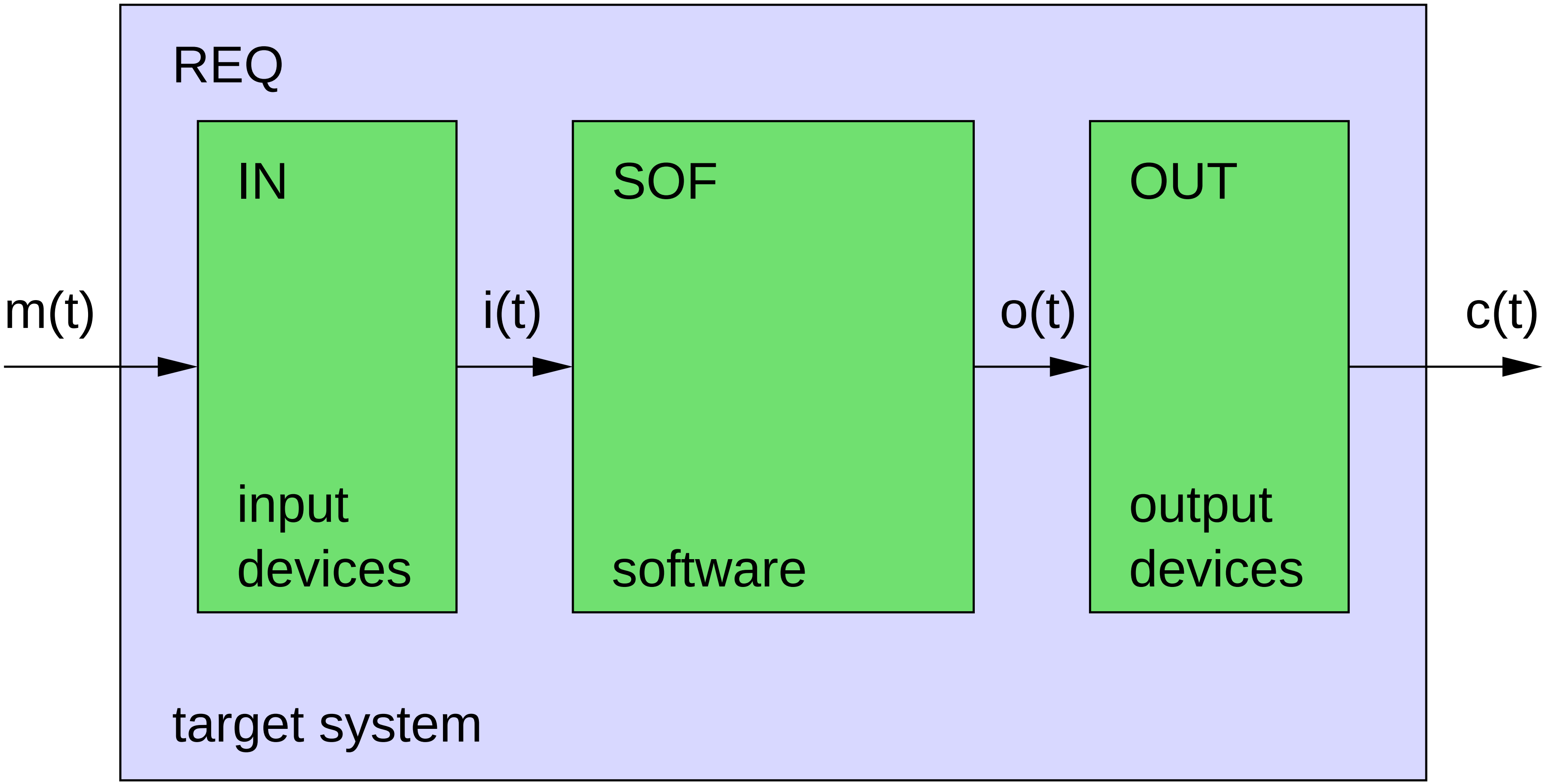
inmode(^{Md}up)	^{C}liftSpeed
inmode(^{Md}standStill)	0 cm/s
inmode(^{Md}down)	$-{}^{C}$liftSpeed

1.2 Software Requirements

System Design

- decisions on what to do in hardware/software
- results in:
  - hardware requirements
  - software requirements

The Four-Variable Approach for System Design and Software Requirements



Input and Output Quantities

- input state function:
  $\underset{\sim}{i}^t : \mathbb{R} \longrightarrow \mathrm{I}_1 \times \mathrm{I}_2 \times \ldots \times \mathrm{I}_n$
- output state function:
  $\underset{\sim}{o}^t : \mathbb{R} \longrightarrow \mathrm{O}_1 \times \mathrm{O}_2 \times \ldots \times \mathrm{O}_m$
- set of all $\underset{\sim}{i}^t$: I
- set of all $\underset{\sim}{o}^t$: O
- behaviour required of
  - the input devices: $\mathrm{IN} \subseteq \mathrm{M} \times \mathrm{I}$
  - the output devices: $\mathrm{OUT} \subseteq \mathrm{O} \times \mathrm{C}$
  - the software: $\mathrm{SOF} \subseteq \mathrm{I} \times \mathrm{O}$

Software Acceptability

- the software requirements SOFREQ are determined completely by REQ, NAT, IN, and OUT
  - $((\mathrm{IN} \cdot \mathrm{SOFREQ} \cdot \mathrm{OUT}) \cap \mathrm{NAT}) = \mathrm{REQ}$
  - SOFREQ usually difficult to calculate precisely
- a software SOF is acceptable if
  SOF with IN and OUT and NAT imply REQ:
  $((\mathrm{IN} \cdot \mathrm{SOF} \cdot \mathrm{OUT}) \cap \mathrm{NAT}) \subseteq \mathrm{REQ}$
  - some design decisions make life easier
    - ▷ remove some non-determinism
    - ▷ $\mathrm{SOF} \subseteq \mathrm{SOFREQ}$
  - SOF must still be acceptable

First Application of Four-Variable Method

- software cost reduction project (SCR)
  - developed the method
  - US Naval Research Laboratory (NRL)
- specification of the complete software requirements for the A-7 aircraft's TC-2 on-board computer
  - reverse-engineering of existing system
  - with help from domain experts (pilots, . . .)
- maintained over lifetime of system
  - first release: Nov. 1978
  - end of project: Dec. 1988
- 473 pages

Example: Autonomous Wheelchair "Rolland"

- Univ. of Bremen, AG B. Krieg-Brückner (Thomas Röfer, Axel Lankenau, . . .)
- joystick-to-motor line wiretapped
- ring of sonar sensors
- safety module
- driving assistant
  - turning on the spot skill
  - obstacle avoidance skill
  - . . .

Rolland: Specification of Safety-Relevant Behaviour

- very recent research work on “mode confusion” problems
- requirements documented by A. Lankenau, J. Bredereke
- reverse-engineering work
- language: CSP
  - different formalism
  - model-checking tool available
  - CSP starts out with events, not variables
  - otherwise same software engineering approach used
- slides: presentation ignores CSP

Rolland: Relevant Environmental Quantities

- the joystick command
- the wheelchair motors command
- the actual wheelchair motors status
- location of the obstacles near the wheelchair

Rolland: Environment Variables

Variable	mon.	ctrl.	Description	Type	Notes
^{m}t	•		current time	$\mathbb{R}$	
^{m}joystickCommand	•		the user intended motion as indicated with the joystick	^{t}JoystickCommandVector	
^{c}motorsCommand		•	command for the wheelchair motors	^{t}MotorsCommandVector	
^{m}motorsActual	•		the actual motors status of the wheelchair	^{t}MotorsCommandVector	
^{m}obsLoc	•		location of relevant obstacles	^{t}obstacleLocs	
^{m}orientation	•		the current orientation of the wheelchair	^{t}orientationRange	1

Notes

1. The orientation is relative to the world (inertial system). At program start, the orientation is 0°.

Rolland: Environment Variable Types

Type	Description	Values	Unit
^{t}JoystickCommandVector	a joystick command vector (i,d). i: fraction of max. joystick inclination, d: direction of the joystick inclination	^{t}inclinationRange $\times$ ^{t}orientationRange	(%, °)
^{t}inclinationRange	fraction of max. joystick inclination	$\{x \in \mathbb{R} \mid 0 \leq x \leq 100\}$	%
^{t}orientationRange	a direction. 0: straight ahead 90: left 180: straight back -90: right	$\{x \in \mathbb{R} \mid -180 < x \leq 180\}$	°

Type	Description	Values	Unit
^{t}MotorsCommandVector	A command vector (s,a) sent to the motors as target value. s: speed value, restricted by physical limitations of the wheelchair, a: angle of the wheelchair's steering wheels	^{t}speedRange $\times$ ^{t}steeringAngleRange	(cm/s, °)
^{t}speedRange	physical wheelchair speed range (167 cm/s is 6 km/h)	$\{x \in \mathbb{R} \mid -167 \leq x \leq 167\}$	cm/s
^{t}steeringAngleRange	angle of steering wheels of wheelchair. -60: right 0: straight 60: left	$\{x \in \mathbb{R} \mid -60 \leq x \leq 60\}$	°
^{t}obstacleLocs	...	...	...

(the rather complex type ^{t}obstacleLocs is omitted in the slides)

Rolland: Observations

- precise link between environmental quantities and mathematical variables
  - definitions in rigorous prose
  - explicit units
  - explicit meaning of individual values of a range
- tabular format suitable
- duplication of description avoided

Rolland: Conditions and Events

- simple for Rolland
- not specified separately
- specified in-place in the relations (see later)

Rolland: the Relation NAT

complete description would comprise:

- the wheelchair obeys to commands after a delay
  - acceleration/deceleration
  - steering
- obstacles don't move by themselves
- obstacles are always visible for the sonar sensors

- simplified specification for case study:

$$\exists t_d \in (0 \ldots {}^{C}\mathsf{maxDelMot}]\ .$$
$$\quad {}^{m}\mathsf{motorsActual} = {}^{c}\mathsf{motorsCommand}({}^{m}\mathsf{t} - t_d)$$

- restrictions of value ranges already specified by types
- convention: if omitted, ${}^{m}\mathsf{t}$ is parameter implicitly

Rolland: the Relation REQ

- was specified in case study only implicitly
  - because of reverse-engineering approach
- explicitly: IN, SOF, OUT, and NAT
- we can assume $\mathrm{SOFREQ} = \mathrm{SOF}$ and then derive $\mathrm{REQ} = ((\mathrm{IN} \cdot \mathrm{SOFREQ} \cdot \mathrm{OUT}) \cap \mathrm{NAT})$

Rolland: Input Variables

Input Variable	Description	Type	Notes
^{i}joystickUnitCommand	the user intended motion as indicated with the joystick	^{t}JoystickUnitCommandVector	
^{i}motorsUnitActual	the actual motors status of the wheelchair	^{t}MotorsUnitCommandVector	
^{i}obsLoc	location of relevant obstacles	^{t}obstacleLocsMap	1
^{i}orientation	the current orientation of the wheelchair	^{t}odoOrientationRange	2

Notes

1. This does not include obstacles that cannot be detected by the wheelchair's sonar sensors, because of their known technical limitations (surface structure dependance, objects visible only at sensor level, etc.)
2. The orientation is relative to the world (inertial system). At program start, the orientation is 0°. This information is only reliable over short distances due to odometry drift.

Rolland: Input and Output Variable Types

Type	Description	Values	Unit
^{t}MotorsUnitCommandVector	A command vector (s,r) interpreted by the motors unit. s: speed value, restricted by safety and comfort limitations of the wheelchair, r: curve radius of the wheelchair's steering wheels	^{t}SpeedCommandRange $\times$ ^{t}RadiusRange	(cm/s, cm)
^{t}JoystickUnitCommandVector	a command vector (s,r) containing the interpreted joystick command, interpretation as above	^{t}MotorsUnitCommand-Vector	(cm/s, cm)
^{t}SpeedCommandRange	speed range used for target commands (coming from the joystick and sent to the motor) (84 cm/s is 3 km/h)	$\{x \in \mathbb{N} \mid -42 \leq x \leq 84\}$	cm/s

Type	Description	Values	Unit
^{t}RadiusRange	curve radius range < 0: right curve > 0: left curve 0: straight other values between -50 and $+50$ are physically impossible and are interpreted as -50 and $+50$, respectively	$\mathbb{N}$	cm
^{t}odoOrientationRange	a direction, as computed by odometry.	$\{x \in \mathbb{N} \mid -180 < x \leq 180\}$	$^\circ$
^{t}obstacleLocsMap	...	...	...

(the rather complex type ^{t}obstacleLocsMap is omitted in the slides)

Rolland: Output Variables

Output Variable	**Description**	**Type**	**Notes**
^{o}motorsUnitCommand	the command for the wheelchair motor unit	^{t}MotorsUnitCommandVector	

Rolland: the Relation IN

^{i}joystickUnitCommand $=$

^{m}joystickCommand.d $> 90 \lor$ ^{m}joystickCommand.d < -90	(round(^{m}joystickCommand.i$/100 \cdot -42$), calcRadius(calcSteeringAngle(^{m}joystickCommand.d)))
$\neg$(^{m}joystickCommand.d $> 90 \lor$ ^{m}joystickCommand.d < -90)	(round(^{m}joystickCommand.i$/100 \cdot 84$), calcRadius(calcSteeringAngle(^{m}joystickCommand.d)))

Note: round, calcSteeringAngle, and calcRadius are functions defined in the Dictionary and omitted in the slides.

^{i}motorsUnitActual $=$

^{m}motorsActual.s $\geq -42 \land$ ^{m}motorsActual.s ≤ 84	(round(^{m}motorsActual.s), calcRadius(^{m}motorsActual.a))
^{m}motorsActual.s < -42	(-42, calcRadius(^{m}motorsActual.a))
^{m}motorsActual.s > 84	(84, calcRadius(^{m}motorsActual.a))

^{i}orientation $=$ round(^{m}orientation)

^{i}obsLoc $= \ldots$

Rolland: the Relation OUT

$^{c}\text{motorsCommand} = (^{o}\text{motorsUnitCommand.s}, \text{calcMotorSteeringAngle}(^{o}\text{motorsUnitCommand.r}))$

Rolland: the Relation SOF

- complex behaviour, see specification in CSP *editor*

- specify output variables in terms of input variables
- use mode classes as appropriate

1.3 Further Issues

System Modes vs. Environmental Modes

- environmental mode
  - equivalence class of histories
  - change depends on occurrence of events
  - initial env. mode depends on history before system turned on
- system mode
  - equivalence class of system states
  - change depends on *detection* of events
  - initial system mode is fixed
- *ideally*, system and env. modes should be equivalent

"Ideal" Behaviour is Impossible

- *accuracy* of measurement of analogue monitored quantities
- *tolerance* of analogue controlled quantities
- important analogue monitored quantity: *time*
  - detection of events needs time
  - reaction to events needs time

A Useful Heuristics for “Real” Behaviour

- specify “ideal” behaviour relation
- specify separately accuracy and tolerance relations and concatenate these relations
  - do not forget this!
- may not work for more complex timing
  - then need explicit “transition” modes

Example: Logic Probe

- device giving a short pulse of 100 ms when button pressed

^{Cl}probe =

Mode	Event Class	New Mode
^{Md}test	@T(^{m}Pulse $=$ ^{C}Down)	^{Md}pulse
^{Md}pulse	@T(Since(@T(^{Md}pulse)) $>$ 100 ms)	^{Md}test

Maximum Delay: 2 ms

Logic Probe With Delay, Expanded

- same behaviour, but without delay specification
- implicit transition modes made explicit for demonstration

^{Cl}probe =

Mode	Event Class	New Mode
${}^{Md}\widehat{\text{test}}$	@T(^{m}Pulse = ^{C}Down)	^{Md}test–pulse
^{Md}test–pulse	@T(^{c}Requiv $\leq$ 320 Ω)	${}^{Md}\widehat{\text{pulse}}$
	@T(Since(@T(^{Md}test–pulse)) $\geq$ 2 ms)	
${}^{Md}\widehat{\text{pulse}}$	@T(Since(@T(^{Md}pulse)) $>$ 100 ms)	^{Md}pulse–test
^{Md}pulse–test	@T(^{c}Requiv $\geq$ 500 kΩ)	${}^{Md}\widehat{\text{test}}$
	@T(Since(@T(^{Md}pulse–test)) $\geq$ 2 ms)	

- ^{c}Requiv: a controlled variable reflecting the mode (needed!)

Using Discrete Clocks

- many embedded software systems:
  cycle read→process→write→. . .
- read and write at discrete points of time
  - system requirements should permit such implementations
- concise requirements by specifying the required *resolution* of time
  - resolution = smallest significant increment of time

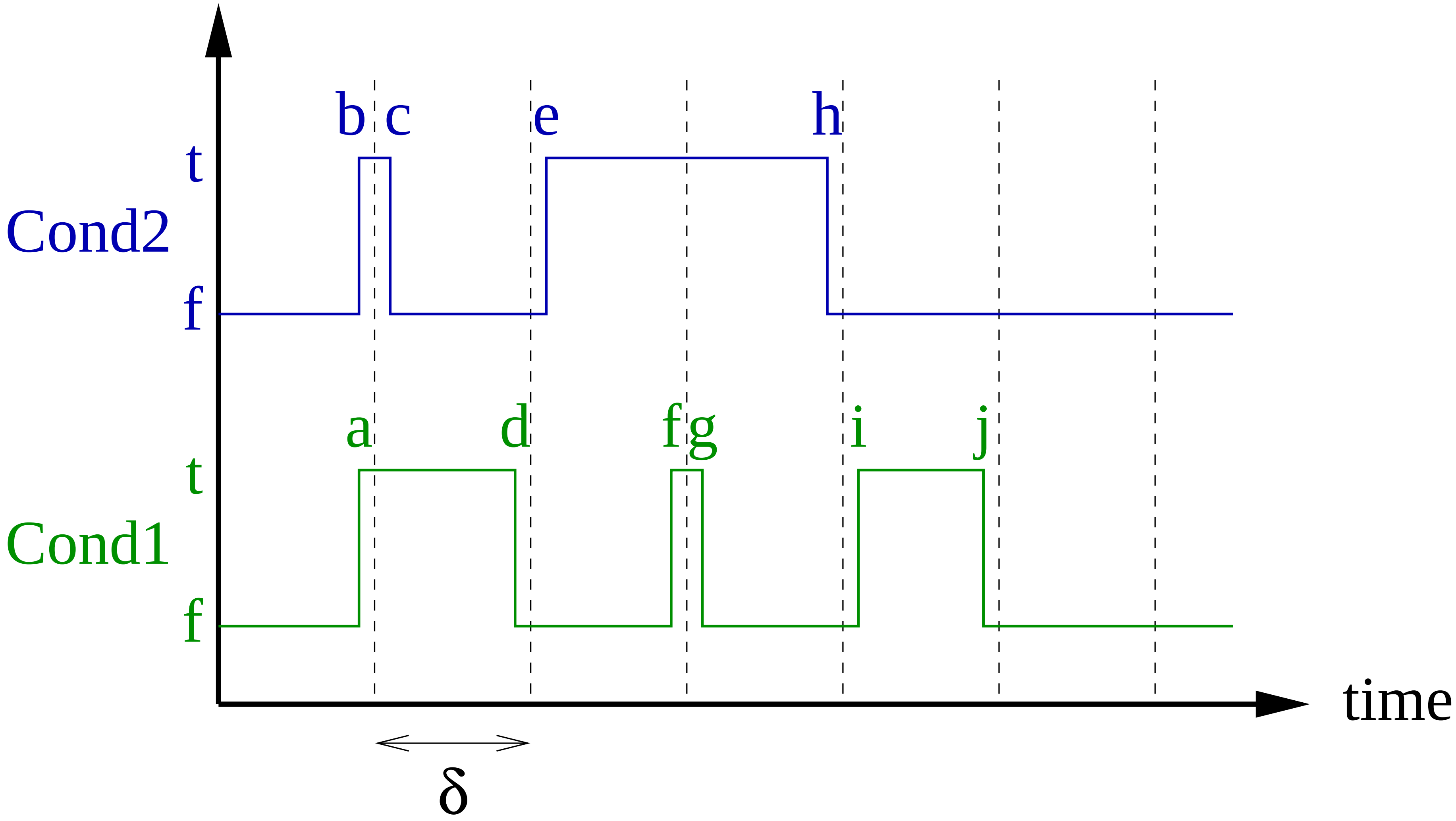
Implications for System when Specifying a Resolution of Time δ

- system clock frequency $\geq \frac{1}{\delta}$
  - sufficient to sample monitored quantities at rate of $\frac{1}{\delta}$

Implications for Requirements when Specifying a Resolution of Time δ

- changes in environment that occur within δ may be considered simultanteous
- system can only be required to detect conditions that have held for at least δ
- max. measurement accuracy for instants: $+0 \ / \ -\delta$
- max. measurement accuracy for time intervals: $\pm\delta$
- min. delay tolerance for response to any event: δ

Example: Time Resolution



Useful Standard Functions For Time

- implicitly interpreted w.r.t. a particular behaviour on the interval of the system's operation $[t_i, t_f]$

Prev(e, t)	the set of events of event class e that occur prior to t
Last(e, t)	the time of the latest event of event class e before t
First(e, t)	the time of the earliest event of event class e before t

Drtn(p_i, t)	the duration that condition p_i has been continuously true up to time t
totalDrtn(p_i, t_1, t_2)	the total amount of time that condition p_i has been true between times t_1 and t_2
Since(e, t)	the time since the latest event of event class e before t

- if time argument t is current time t_f, it will be omitted by convention
- precise definitions in [Pet00, pp. 49]

Repetition: Events

An event e, is a pair, (t, c), where
$e.t \in \mathbb{R}$ is a time at which one or more conditions change value and
$e.c \in \{T, F, @T, @F\}^n$ indicates the status of all conditions at $e.t$, as follows:

$e.c[i]$	p_i
T	$`p_i(e.t) \wedge p_i'(e.t)$
F	$\neg`p_i(e.t) \wedge \neg p_i'(e.t)$
@T	$\neg`p_i(e.t) \wedge p_i'(e.t)$
@F	$`p_i(e.t) \wedge \neg p_i'(e.t)$

Some Useful Event Class Notation

Notation	$e.c[i]$
$*$	true
$\oslash$	false
–	F $\vee$ T
t	T $\vee$ @F
f	F $\vee$ @T
t′	T $\vee$ @T
f′	F $\vee$ @F

- t$(p_i) =$ ‘$p_i(e.t) \wedge$ true
- t′$(p_i) =$ true $\wedge\, p_i{}'(e.t)$

Example: Telephone Connection

- table describes the connection mode
  between any two users u and v
- from a large requirements specification (Bredereke)

current mode	**conditions** ^{m}connectReq(u,v)	inmode(^{Md}connection-ResourceAvail(u,v))	^{m}connectRsp(v)	**next mode**
^{Md}Idle(u,v)	@T	t′	–	^{Md}Setup(u,v)
	@T	f′	–	^{Md}OTeardown(u,v)
^{Md}Setup(u,v)	–	T	@T	^{Md}Established(u,v)
	@F	$*$	–	^{Md}Idle(u,v)
	–	@F	$*$	^{Md}OTeardown(u,v)
^{Md}Established(u,v)	–	$*$	@F	^{Md}OTeardown(u,v)
	@F	$*$	–	^{Md}TTeardown(u,v)
	–	@F	–	^{Md}BTeardown(u,v)
^{Md}OTeardown(u,v)	@F	$*$	–	^{Md}Idle(u,v)
^{Md}TTeardown(u,v)	–	$*$	@F	^{Md}Idle(u,v)
^{Md}BTeardown(u,v)	–	$*$	@F	^{Md}OTeardown(u,v)
	@F	$*$	–	^{Md}TTeardown(u,v)

Tabular vs. Scalar Notation for Event Classes

tabular	scalar
p_i	
T	WHILE(p_i)
F	WHILE$(\neg p_i)$
@T	@T(p_i)
@F	@F(p_i)
$*$	*(not useful)*
–	CONT(p_i)
$\oslash$	*(not useful)*

tabular	scalar
p_i	
t	WHEN(p_i)
f	WHEN$(\neg p_i)$
t'	*(no notation defined)*
f'	*(no notation defined)*

Example: Tabular Expressions

^{Cl}floor:

current mode	conditions			next mode
	^{p}at1stFloor	^{p}at2ndFloor	^{p}at3rdFloor	
^{Md}in1stFloor	–	@T	–	^{Md}in2ndFloor
^{Md}in2ndFloor	@T	–	–	^{Md}in1stFloor
	–	–	@T	^{Md}in3rdFloor
^{Md}in3rdFloor	–	@T	–	^{Md}in2ndFloor

Example: Scalar Expressions

^{Cl}floor:

Mode	Event Class	New mode
^{Md}in1stFloor	@T(^{p}at2ndFloor)	^{Md}in2ndFloor
^{Md}in2ndFloor	@T(^{p}at1stFloor)	^{Md}in1stFloor
	@T(^{p}at3rdFloor)	^{Md}in3rdFloor
^{Md}in3rdFloor	@T(^{p}at2ndFloor)	^{Md}in2ndFloor

Requirements Feasibility

$\longrightarrow$ blackboard. . .

Fail-Soft Behaviour in the Four-Variable Approach

- repetition: acceptability of a software SOF:
  $((\mathrm{IN} \cdot \mathrm{SOF} \cdot \mathrm{OUT}) \cap \mathrm{NAT}) \subseteq \mathrm{REQ}$
- if devices are broken, software is not constrained at all
- specify weaker versions of IN, OUT, and SOF that hold if some devices are broken
- software must satisfy the conjunction of of all requirements specified this way

Merit Functions

- although all behaviours in REQ are acceptable, some are preferable over others
- examples:

  processing speed: quicker responses preferred

  soft real-time constraints: failure to respond within specified time not catastrophic, but undesirable

  safety margins: controlled values may approach certain thresholds, but the larger the safety margin the better

  stability: large oscillations in controlled values are undesirable

Definition 11 (Merit function)

A merit function is a function of a behaviour that indicates which behaviours are preferred over which others – the higher the merit function value the more preferred the behaviour.

- related to “objective function” in control systems and optimization

Limitations of the Approach

necessary:

1. env. quantities can be expressed as functions of time that are either
   - piecewise-continuous (for real-valued quantities), or
   - finitely variable (for discrete-valued quantities)
2. the acceptable behaviour can be characterized by a relation on the env. quantities

Environmental Quantities Not Expressible

- if cannot be expressed effectively
  - example: compiler
  - source code = array of characters???
- if not usefully viewed as functions of time
  - example: compiler
  - only two instants of time relevant (start, termination)

- approach unsuitable for "information processing" systems in particular

Requirements Relation Not Expressible

- non-behavioural properties
  - maintainability
  - code size
- internal properties
  - number of times an instruction is invoked
    (if not externally observable)
- requirements not preserved under sub-setting of behaviours

Requirements Not Preserved Under Sub-Setting of Behaviours

- average response time over all behaviours
  - different from average over a single behaviour (which can be expressed)
  - usually, such statistical properties can be approximated reasonably well and specified with reference to a lengthy execution

- possibilistic properties
  - important for security
  - "if behaviour A is possible, then behaviour B must also be possible"
  - this is not the same as
    $A \in \mathrm{REQ} \Rightarrow B \in \mathrm{REQ}$
  - what is acceptable in an implementation
    is different from what is possible
  - usually, REQ is non-deterministic, but the implementation is not
  - intruders must not be able to infer information
    from the possibility of A and the impossibility of B

1.4 Tabular Expressions

Text for Chapter 1.4

[Pet00] Peters, D. K. *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster Univ., Hamilton, Canada (Jan. 2000).

Brief introduction into tabular expressions in Chapter 5.2.3. Most current version of notation.

[JaKh99] Janicki, R. and Khedri, R. *On a formal semantics of tabular expressions*. CRL Report 379, McMaster University, Hamilton, Ontario, Canada (Sept. 1999).

Extensive description and definition of tabular expressions. Notation not entirely up to date.

Introduction to Tabular Expressions

- this must be done real slow,
  so we do it on the blackboard

2. What Information Should Be Provided in Computer System Documentation?

Text for Chapter 2

[PaMa95] Parnas, D. L. and Madey, J. *Functional documents for computer systems*. Sci. Comput. Programming **25**(1), 41–61 (Oct. 1995).

Structure of the requirements documentation and software documentation.

Additional Background for Chapter 2

[PaCl86] Parnas, D. L. and Clements, P. C. *A rational design process: how and why to fake it*. IEEE Trans. Softw. Eng. **12**(2), 251–257 (Feb. 1986).

Structure of the documentation vs. structure of the development process.

Overview of Documents

- system requirements document
- system design document
- software requirements document
- software behaviour specification
- software module guide
- module interface specification
- uses-relation document
- module internal design document

- communication: service specification document
- communication: protocol design document

Specification Form vs. Specification Content

- this overview: concerned with content only
- formalism must be adapted to situation
- choice of some formalism alone does not guarantee completeness of content!
  - “formal” vs. “rigorous”

The System Requirements Document

description of:

- environmental quantities of concern
- association of env. quantities to math. variables
- relationships between values of these due to environmental constraints (NAT)
- relationships between values of these due to new system (REQ)

- descriptions are black-box
- details: see Chapter 1.1

Structure of the System Requirements Document

required sections:

- environmental quantities
- environmental constraints
- system behaviour
- dictionary
  - definitions of:
    - math. functions and relations
    - words that are not common natural language
    - words that have special meaning in application domain

optional sections:

- system overview
  - informal
  - possibly including non-behavioural requirements
- notational conventions
  - if non-standard notation used
  - variable naming
  - special variable mark-up
  - . . .
- anticipated changes
  - important to reduce effort for later changes
  - see also Chapter 4

The System Design Document

- introduces input and output variables

description of:

- relationships between monitored and input variables (IN)
- relationships between output and controlled vars. (OUT)
- relationships between input and output variables (SOF) (software requirements)
  - in separate document, see below

- details: see Chapter 1.2

The Software Requirements Document

- software requirements (SOFREQ) implicitly determined by
  - system requirements document
  - system design document

  } = software requirements doc.

  (NAT, REQ, IN, OUT)
- usually design step:
  explicit, more deterministic
  software behaviour specification (SOF)
- details: see Chapter 1.2

The Software Behaviour Specification

- SOF
- details: see Chapter 1.2
- particularly important for multi-processor / multi-computer / network systems
  - allocation of tasks to individual computers
  - hierarchy of software behaviour specifications

Software Modules

Definition 12 (Module)
A module is a programming work assignment.

- (see other definitions of “module” later in lecture)
- assume information hiding principle was used (see below)
- black-box description of module’s behaviour

The Software Module Guide

- division of software into modules
- states responsibilities of each module
- informal "guide"
  - rigorous module interface specification necessary to start implementation
- details: see Chapter 3.2 later in lecture

The Module Interface Specification

- each module implements one or more finite state machines (FSMs)
  - FSMs also called *objects* or *variables*
- description of module interface is black-box description of these objects
  - every "*program*" (= method/function/. . .) belongs to exactly one module
  - programs use objects created by other modules as components of their data structure

Writing Module Interface Specifications

- similar to documenting software requirements
- simplifications possible
  - many software modules are entirely internal
    - ▷ no environmental quantities
    - ▷ all communication through external invocation of the module's programs
  - state set finite
  - state transitions can be treated as discrete events
  - often: real-time can be neglected, only the sequence of events matters
    - ▷ replace time-functions by traces

Formalisms for Module Interface Specifications

- "Trace Assertion Method" proposed by Parnas *et.al.* was never used much
- many other formalisms known and in use:
  - CSP
  - Z (/ Object-Z)
  - SDL
  - StateCharts
  - . . .

  (CSP and Z (/ Object-Z): see lecture Safety-Critical Systems 3)

  advantages/disadvantages depend on application domain

The Uses-Relation Document

- range and domain of “uses” relation:
  subsets of set of access-programs of the modules
  - (P, Q) in relation if program P uses program Q
- document often is a binary matrix
- constrains work of programmers
- determines viable subsets of the software
- for details, see Chapter 3.4 later in lecture

The Module Internal Design Document

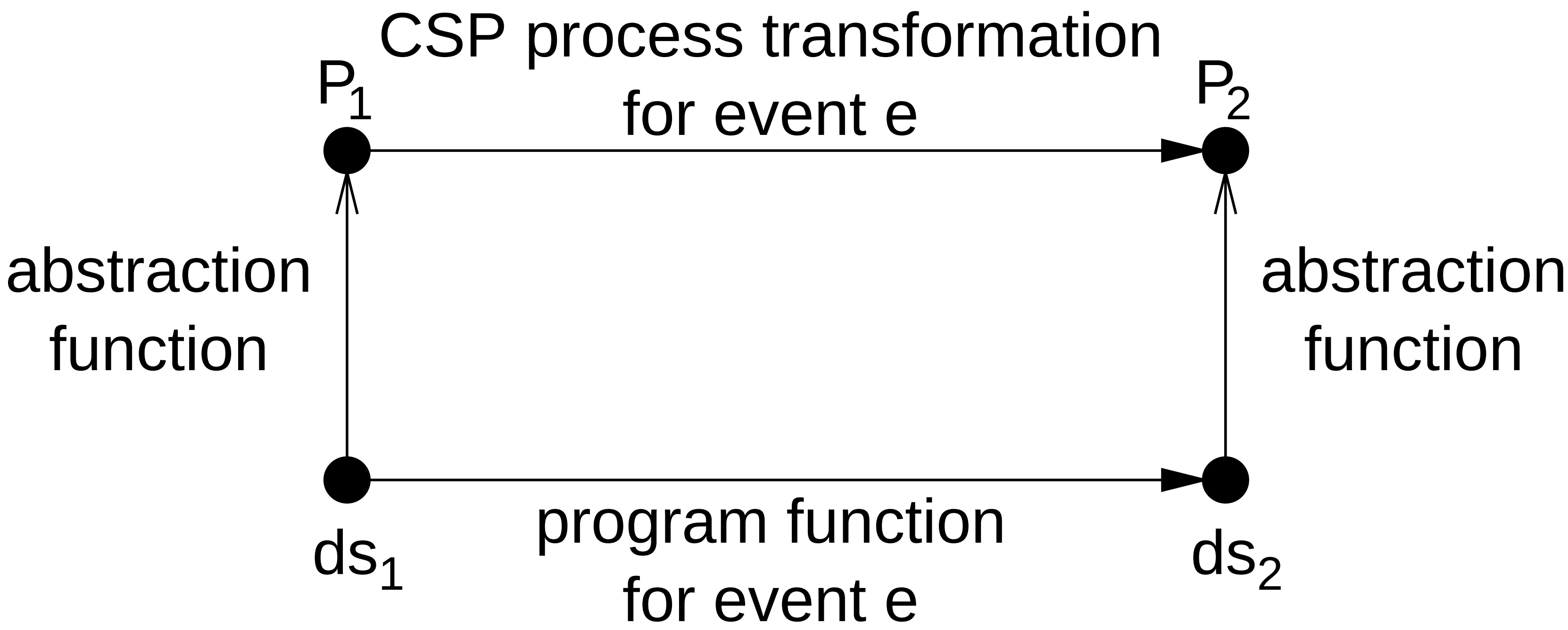
- for each module
- describe module's data structure
- state intended interpretation of data structure
  (in terms of external interface)
- specify effect of each access-program on data structure
- "clear-box description"
- sufficiently precise to verify the workability of the design
  (together with module interface specification)

Information in the Module Internal Design Document

1. complete description of data structure
   (may include objects implemented by other modules)
2. abstraction function
   from values of objects
   to descriptions in terms of external program calls
3. program function:
   an LD relation specifying each program as a
   mapping from states before to states after execution

Abstraction Function

for deterministic programs; using CSP:



- if design correct, then diagram commutes for all events
- if program non-deterministic, program funct. is LD relation

Programs

Definition 13 (Program)

A program is a text describing a set of state sequences in a digital (finite state) machine.

- Each state sequence is called an execution of the program.

Documenting the Effect of Individual Programs

- execution
  - starting state
  - final state (if finite)
  - or infinite sequence
- intermediate states often not interesting, only:
  - termination possible?
  - termination guaranteed?
  - if termination possible, then in which final states?

  } LD relation
- if with parameters, then
  functions from parameters to programs

LD Relation

$\longrightarrow$ blackboard. . .

Documenting by LD Relations

- for specification of program
- for actual behaviour of program
- notations:
  many, depending on application area
  - “displays” proposed by Parnas *et.al.*
    were never used much

Communication: The Service Specification Document

- communication system often implemented as a hierarchy of services
- each level can be viewed as a module
- black-box behaviour of a module = service specification

Communication: The Protocol Design Document

- implementation = protocol design
  - using lower-level services
  - using local data structures
- is a kind of internal module design document

A Rational Design Process: How and Why to Fake It

- all this is straight top-down development
- “reality does not work this way!”
- but it pays to pretend that it does
- text: [PaCl86]

A Rational Person

- one who always has a good reason for what he does
- each step is provably the best way to get to the goal

- are you a rational professional?

- top-down approaches: desire for rational software design
  - the search for the philosopher's stone

Why a Rational Design Process Does Not Work

- customer does not know exactly what he wants, customer cannot tell us all he knows
- even if we knew the requirements: we don't know all details necessary for the best design decisions
  - need to backtrack in design
  - minimize lost work

- even if we knew all relevant facts:
  a human cannot handle this huge amount of details
  - separation of concerns helps
  - but before concerns are separated,
    we are bound to make errors
- even if we could master all detail:
  all projects change due to external reasons
  - minimize lost work
- human errors are inevitable
  - even after separation of concerns
- we have preconceived design ideas
  - own invention, from related projects, learned in class
  - try out favorite idea in project

- re-use of software
  - from previous project
  - shared with parallel project
  - off-the-shelf software
  - software not ideal for project, but will save effort

- are small textbook examples rational?
  - no, polished until they show the point nicely

Why a Rational Design Process is Useful Nevertheless

- keeping as close to the process *as possible* helps
  - guideline
- the *documentation* that would have resulted from this process is useful
- this is "faking a rational design process"

Why Use an Ideal Process as a Guideline

- designers need guidance: what to do first?
- even if we cannot know all facts at the beginning:
  trying to find them reduces backtracking
  and thereby improves the design
- measure progress of project
  - relative to ideal process
- an organization needs a standard process for projects
  - to transfer people, ideas, software
  - external review of projects (measure progress)
  - a rational process is a good base
    - ▷ more refined processes (V-model, . . .): → *SCS 2 (SoSe 03)*

What should the Process Description Tell?

- what product to work on next
- what criteria the product must satisfy
- what kind of persons should do the work
- what information they should use

most useful: description in terms of work products

- allows reviews and progress measurement
  - see also course: "Integrierte Softwareentwicklung und Qualitätssicherung mit Together" (WiSe 02/03, Buth)

The Rational Design Process

1. Establish and Document Requirements
2. Design and Document the Module Structure
3. Design and Document the Module Interfaces
4. Design and Document the Uses Hierarchy
5. Design and Document the Module Internal Structures
6. Write Programs
7. Maintain

What is Wrong With Most Documentation Today

- many programmers don't expect (their) documentation to be useful
  - self-fulfilling prophecy
- why is incomplete or inaccurate information not simply added or corrected?

Underlying Organizational Problems of the Documents

- poor organization
- boring prose
- confusing and inconsistent terminology
- myopia

Poor Organization

- documents often organized by either
  - stream of conciousness
    - ▷ ordered by time when thought occurred
  - stream of execution
    - ▷ ordered by system's run-time order
- difficult . . .
  - to find particular information
  - to check completeness
  - to change consistently

Boring Prose

- lots of words used to say what could be said by
  - single programming language statement
  - formula
  - diagram
- certain facts repeated in many sections
- leads to: inattentive reading, undiscovered errors

Confusing and Inconsistent Terminology

- any complex system needs new terminology
  - otherwise documentation far too long
- software documentation often does not provide precise definitions
  - many terms used for same concept
  - many similar, distinct concepts described by same term

Myopia

- documentation written near completion of project
- major decisions taken for granted
- small details are documented
  - to avoid forgetting them
- useful for insiders
- impenetrable for newcomers

How to Avoid Poor Organization

(i.e., avoid "stream of conciousness", "stream of execution")

1. design the structure of each document explicitly
   - by stating the *questions* that it must answer
   - by refining these questions until each defines one section
   - *one and only one place for every fact*
   - several documents of a kind: have a standard organization
2. answer questions (write document) after the structure has been defined
   - each aspect: one section
   - each section: only one aspect

   } is also separation of concerns
3. reviews: for content and also for documentation rules

How to Avoid Boring Prose

- increase density of information
  - use tables, formulas, formal notation
- prevent duplication by above organizational rules
- still not easy reading,
  but provides precise information

How to Avoid Confusing and Inconsistent Terminology

- have a “dictionary”
- typed terms
  - (monitored, controlled, input, output, . . . quantities)
- mark-up of terms with type: ^{m}term1, ^{c}term2, . . .
- separate dictionary for each type
  - easier to check for similar terms
- mechanical checks for
  - terms introduced but not used
  - terms used but not introduced

How to Avoid Myopia

- use documentation as a means of design

- documents written before myopia starts
- documents mature when the maintainer needs them

Faking the Ideal Process

- attempt to produce documents in order of ideal process
- when information is unavailable:
  - note this fact in the document instead
  - continue process as if this information were expected to change
- error found: correct it, and update all documentation
- no design decisions considered to be made until they are documented

Analogous Process: Mathematical Proofs

- often: painful, difficult discovery process
- then: polished
- others may find simpler proof
- the simplest proof is published
- readers interested in truth of theorem, not of its discovery

One Difference to Ideal Documentation

- record all design alternatives considered
  - why considered
  - why rejected

- for ourselves
- for a later maintainer

3. Decomposition Into Modules

Overview of Chapter 3: Decomposition Into Modules

3.1 the *criteria* to be used in decomposing systems into modules

3.2 structuring complex software with the *module guide*

3.3 time and space decomposition of *complex structures*

3.4 designing software for ease of *extension and contraction*

3.1 The Criteria to be Used in Decomposing Systems into Modules

Text for Chapter 3.1

[Par72] Parnas, D. L. *On the criteria to be used in decomposing systems into modules*. Commun. ACM **15**(12), 1053–1058 (1972).

Seminal paper on information hiding and modularization. Still valid.

Additional Background for Chapter 3.1

[HoWe01] Hoffman, D. M. and Weiss, D. M., editors. *Software Fundamentals – Collected Papers by David L. Parnas*. Addison-Wesley (Mar. 2001).

A collection of important Parnas papers. With introductions on their history and current relevance. Includes [Par72].

What is a Module?

- historically: a unit of measure
  - e.g., 2,54 cm
- manufacturers learned to build parts that were one unit large
- word now: the parts themselves
- modules: relatively self-contained systems, combined to make a larger system
- design: often is assembly of many previously designed modules

The Constraints on Modules

- if modules are hardware:
  obvious how to put them together
  - well-known physical constraints
  - well-identified time for module assembly
- if modules are software:
  no obvious constraints
  - software modules can be arbitrarily large
  - their interfaces can be arbitrarily complex
- during software development:
  several different times at which parts are combined,
  several different ways of putting parts together

Modules of Software – When are Parts Put Together?

1. while *writing* software
   - parts: work assignments for programmer(s)
   - when: before compilation or execution
2. when *linking* object programs
   - parts: separately compiled (or assembled) programs
   - when: before execution
3. while *running* a program in limited memory
   - parts: executable programs or data
   - when: during run-time

- literature: uses "module" for all three!
- this ambiguity leads to confusion
- *this lecture: only the first meaning* ("while writing SW")

The Constraints on the Three Structures

what constrains our choice of “modularization”?

- for write-time “modules”:
  - intellectual coherence for programmer
  - ability to understand, verify
  - ease of change
- for link-time “modules”:
  - duplicate names
  - time needed to re-compile and link
- for run-time “modules”:
  - memory size
  - frequency of references to items outside module
  - time needed to load into memory

- these three sets of constraints are independent
- only commonality: the word “module”
- three different design concepts

Old Example for a Confusion

- TSS/360
  - time sharing system by IBM, in the 60's
  - very slow
- a well-known IBM researcher:
  "reason is over-modularization"
  - memory thrashing
  - memory management interpretation
- previous popular wisdom:
  make modules as small as possible
  - work assignment interpretation
- two meanings were confused

Recent Example for a Confusion

- a recent book on “software architecture”
  - presents and compares different styles for organizing large software
  - text book
  - well-known authors
  - uses Parnas’ KWIC example (see below)
- does not distinguish write-time / link-time modules
  - e.g., does run-time performance comparisons for write-time modules
- book not used for this lecture. . .

The Effect of Confusing the Meanings

- inefficiency, if
  - forcing write-time modules to be link-time modules:
    - ▷ overhead for frequently executed call sequences
  - forcing write-time modules to be run-time modules:
    - ▷ overhead for frequent memory loads
- high development/maintenance costs, if
  - forcing run-time modules to be write-time modules:
    - ▷ difficult to program and to maintain

- write-time modules need not be compiled separately
  one may use *macro substitution* or similar

Write-Time Modules

- we want the following properties:
  - can be designed and changed independently
  - can be sub-divided into further modules
- when to stop sub-dividing into modules?
  - when so small that it is easier
    to write a new one than to change it
  - when the cost of specifying the interface exeeds any future benefit
    from having smaller modules
- “module = work assignment” is only a definition,
  need guidelines for designing a module structure

Example: A KWIC Index Production System

- KWIC: "Key Words In Context"
- the KWIC index system accepts an ordered set of lines
- each line is an ordered set of words
- each word is an ordered set of characters
- any line may be "circularly shifted" by repeatedly removing the first word and appending it to the end of the line
- the KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order

Example of a KWIC Index

input	output
THE COLOUR OF MAGIC	COLOUR OF MAGIC THE
THE LIGHT FANTASTIC	EQUAL RITES
EQUAL RITES	FANTASTIC THE LIGHT
MORT	LIGHT FANTASTIC THE
MOVING PICTURES	MAGIC THE COLOUR OF
	MORT
	MOVING PICTURES
	OF MAGIC THE COLOUR
	PICTURES MOVING
	RITES EQUAL
	THE COLOUR OF MAGIC
	THE LIGHT FANTASTIC

Output of The Unix ptx Utility

(ptx: "permuted index")

```
            THE   COLOUR OF MAGIC
                  EQUAL RITES
      THE LIGHT   FANTASTIC
            THE   LIGHT FANTASTIC
  THE COLOUR OF   MAGIC
                  MORT
                  MOVING PICTURES
     THE COLOUR   OF MAGIC
         MOVING   PICTURES
          EQUAL   RITES
                  THE COLOUR OF MAGIC
                  THE LIGHT FANTASTIC
```

Ideas for a Modularization

- pretend: programming task is so large that it must be performed by serveral persons
- how should we modularize the KWIC index software?
  - which modules?
  - which interfaces between modules?

(discussion) *editor*

What are the Criteria for a Modularization?

- well, what are they? *editor*
- does our modularization meet them?

“Conventional” Modularization

1. Input Module

- reads data lines from input medium
- stores them in memory, packed four to a word
- end of word marker: an otherwise unused character
- makes index to show start address of each line

input interface: input format, marker conventions
output interface: memory format

2. Circular Shift Module

- called after input module
- makes index with addresses of first char. of shifts
- output is array of pairs of words (start of line, start of shift)

input interface: memory format
output interface: memory format, perhaps the same

3. Alphabetizing Module

- takes the arrays of modules 1 and 2
- produces an array in format of module 2
- the result is ordered alphabetically

input interface: memory format
output interface: memory format

4. Output Module

- takes the arrays of module 3 and 1
- produces formatted output listing
- maybe: mark start of line, . . .

input interface: memory format
output interface: paper format, conventions, . . .

5. Master Control Module

- controls the sequencing of the other modules
- handles error messages, memory allocation, . . .

interface: names of the program to be invoked

Some Likely Changes

1. input format
   (a) line break characters (`\n` / `\r\n` / `\r`)
   (b) word break characters
   (c) size of a character (7 bit / 8 bit / Unicode)
2. memory formats
   (a) keep all lines in memory?
   (b) pack characters four to a word?
   (c) store shifts explicitly / as index+offset
3. decision to sort all output before starting to print

4. decision to produce all shifts
   (a) eliminate shifts starting with noise words
   (b) eliminate shifts not starting with only-words
5. different alphabetizations
   (a) ignore case
   (b) locale
6. output format
   (a) different visual output layouts
   (b) truncate overlong lines in output
   (c) generate output for different formatting tools

Parnas' Modularization

1. Line Holder Module

- special purpose memory to hold lines of KWIC index

interface programs:

- GET_CHAR(lineno, wordno, charno)
- SET_CHAR(lineno, wordno, charno, char)
- CHARS(lineno, wordno)
- WORDS(lineno)
- LINES
- DELETE_LINE(lineno)
- DELETE_WORD(lineno, wordno)

2. Input Module

- reads from input medium
- calls line-holder programs to store in memory

interface program:

- INPUT

3. Circular Shift Module

- creates "virtual" list of circular shifts
- uses line holder programs to get data from memory
- may or may not create an actual table

interface programs:

- CS_SETUP
- CS_CHAR(lineno, wordno, charno)
- . . . (analogs to the other programs of the input module)

4. Alphabetizer Module

- does actual sorting of the shifts
- may or may not produce a new list
- if it doesn't, it makes a directory

interface programs:

- ALPH
- ITH(lineno)
- . . . (some more supporting programs)

5. Output Module

- does the actual printing
- calls ITH and circular shift programs

interface program:

- OUTPUT

6. Master Control Module

- links all modules together to do the job
- is the main program, but very simple
- calls INPUT, CS_SETUP, ALPH, and OUTPUT

Comparison of the Two Modularizations

- both:
  - small, manageable programs,
    to be programmed independently
  - may use same data representations
  - may use same algorithms
  - may result in identical code after compilation
- different:
  - way of cutting up the system
  - interfaces

- changeability:
  - 2nd modularization better changeable (compare list on slide 187)
- independent development:
  - 1st: cooperation of all teams until best data representation is found
  - 2nd: teams can start independently early
- comprehensibility:
  - 1st: output module can be understood only by understanding some constraints of the alphabetizer, shifter, and input module

The Criteria

criteria for designing *information-hiding* modules:

- identify the design decisions that are likely to change
  - requires experience and judgement
  - is additional work up-front
- have a module for each that is very likely to change

The *Secret* of a Module

- the design decision that might change
  - only the implementor needs to know what decision was made

Examples for Module Secrets

- line holder module
  - how lines are represented in memory
- input module
  - input format
- circular shift module
  - how shifts are represented
- alphabetizer module
  - sorting algorithm
  - time when alphabetization is done
- output module
  - output format

Some Specific Criteria

the following should be hidden in a single module:

- a data structure, its access and modyfying procedures
- a routine and its assembly call sequence
- control block formats (into a control block module)
- character codes, alphabetic orderings, . . .
- sequence of processing

Interface Between Modules

- the *assumptions* that they make about each other

Module Structure

system structure:

- a system's parts and their connections
  - connections: the modules' interfaces (i.e., assumptions)
  - parts: work assignments (modules)

Efficiency and Implementation

- frequency of switching between modules at run-time:
  - steps-in-processing approach: low frequency
  - information-hiding approach: high frequency
- module access programs need not be subroutines
  - the usual space-time tradeoffs apply
  - supporting language constructs:
    - ▷ macros (in C, C++, not in Java)
    - ▷ `inline` functions/methods (in C++, not in Java)
    - ▷ templates (in C++, not in Java)
  - automatically optimizing compilers
    - ▷ they know size of code, but not frequency of calls

Information Hiding and Abstract Data Types

- data abstraction is a special case of information hiding
  - algorithms can be hidden as well
- data types allow many copies of the hidden structure
  - each variable has one copy

Information Hiding and Object-Orientation

- both: group data and programs together
- information hiding: no inheritance
- OO: often no distinction of write-time/link-time modules

Information Hiding and Program Families

- designing not a single program, but a program family
- early: decisions shared by all members
- postpone: decisions likely to change

- see Chapters 3.4 and 4

3.2 Structuring Complex Software with the Module Guide

Text for Chapter 3.2

[PCW85] Parnas, D. L., Clements, P. C., and Weiss, D. M. *The modular structure of complex systems*. IEEE Trans. Softw. Eng. **11**(3), 259–266 (Mar. 1985).

Information hiding; the modules to decompose into.

Additional Background for Chapter 3.2

[Lam88] Lamb, D. A. *Software Engineering: Planning for Change*. Prentice-Hall (1988).

Chapter 5: information hiding; the modules to decompose into.

Why the Gap Between Information Hiding in Theory and in Practice?

(before start of SCR project)

1. idea is impractical for real problems?
2. responsible managers unwilling to bet on unproven idea? (startup problem)
3. examples in papers too unlikely to practical problems?
4. idea needs refinement or extension for complex projects?
5. practitioners not intellectually capable of application?

Why the Gap Between Information Hiding in Theory and in Practice?

1. idea is impractical for real problems?
   - *no*
2. responsible managers unwilling to bet on unproven idea? (startup problem)
3. examples in papers too unlikely to practical problems?
4. idea needs refinement or extension for complex projects?
5. practitioners not intellectually capable of application?
   - *no*

Bridging the Gap

2. responsible managers unwilling to bet on unproven idea? (startup problem)
   - started *SCR project* as an example
3. examples in papers too unlikely to practical problems?
   - SCR: A-7E flight operational program is realistic
4. idea needs refinement or extension for complex projects?
   - see below

Structuring Complex Software Systems Into Modules

- $many$ implementation decisions, $many$ details
- therefore *many modules*
- $\leq$ 25 modules:
  - not difficult to know:
    - ▷ which modules affected by a change
    - ▷ whether coverage complete
  - careful inspection
- hundreds of modules??
  - information hiding alone does not work here!

Needed: the Software Module Guide Document

- tree-structured hierarchy
- additional goals by hierarchy and guide:
  - well-defined concern: easily find relevant modules without looking at all the others
  - number of branches at each node small enough such that designers can argue convincingly that
    - ▷ no overlapping responsibilities of submodules
    - ▷ all responsibilities of module are covered
  - again: understand responsibility of a module without understanding its internal design

The Software Module Guide Document

- how responsibilities are allocated among the major modules
- the criteria used to assign a particular responsibility
- scope and contents of the individual design documents

- large example will follow

When to Write the Software Module Guide

- start after SW behaviour specification (SOF) is complete
- refine top-level modules as concurrent work assignments
  - each refinement step renders more concurrent design work assignments
- the module interface specification writers work out the details
- the module internal design follows

Tracing Requirements

- software module guide derived from
  SW behaviour specification (SOF)
- easy to trace requirements to modules
- easy to trace back a design decision to the requirements

Access to a Module's Access Programs

- any program may use any access program of any module in the guide
  - independent of relative positions in hierarchy
  - but see also the "uses hierarchy" in Chapter 3.4 later on!

Module Interfaces May Change

- module interfaces are (higher-level) design decisions
  - may change
  - like module contents are design decisions
- encapsulate these interfaces in higher-level modules
- don't mention these sub-modules in guide
  - don't use sub-modules outside this module
- additional local module guide for this module

Difficulties During Structuring

- unstable information that cannot be encapsulated
  - $\longrightarrow$ "restricted" modules
- need to locate "secret" modules in the guide
  - $\longrightarrow$ "hidden" modules

Restricted Modules

- a problem:
  - we should confine information about hardware that could be replaced
  - diagnostic information about that hardware must be communicated to display modules
- restrict use of such modules
  - mark by “(R)” in module guide
  - try to avoid using restricted modules because of potentially high costs of change

Hidden Modules

- often: existence of certain sub-modules is a secret
  - not in the global guide
  - no use outside this module
- sometimes: existence of sub-module is a secret, but guide should clearly state where certain functionality is
  - mention these sub-modules in guide
    - ▷ mark by "(H)" as hidden
  - still no use outside the module

Two Kinds of Module Secrets

- primary secret
  - hidden information specified to the software designer
- secondary secrets
  - implementation decisions made by the designer when implementing

The Classes of Modules in the A-7E Software Module Structure

top-level decomposition:

1. *hardware-hiding* module
2. *behaviour-hiding* module

} secret is in software requirements document

3. *software decision* module } secret is not a requirement

- this top-level decomposition is valid for *nearly all SW systems!*

- hardware-hiding module
  - any programs affected by replacing a device
    - ▷ with different interface
    - ▷ with same general capabilities
  - implements virtual hardware used by rest of software
  - even for “non-embedded” software
    - ▷ any programs affected by likely changes in the operating system
  - primary secrets:
    - ▷ the hardware-software interfaces described in the requirements document
  - secondary secrets:
    - ▷ data structures and algorithms used to implement the virtual hardware

- behaviour-hiding module
  - any programs affected by changes of the required behaviour
  - these programs determine the values to be sent to the "virtual hardware" output devices
  - primary secrets:
    - ▷ the required behaviour

- software decision module
  - hides software design decisions based upon
    - mathematical theorems
    - physical facts
    - programming considerations (efficiency, accuracy)
  - secrets and interfaces determined by software designers
    - secrets are *not* in the requirements document
  - likely reason for changes here:
    - improve performance
    - not: externally imposed changes

Fuzziness in the Top-Level Classification

1. line between requirements and design decided when requirements are written
   - example: requirements can specify an explicit weapon trajectory model or just accuracy requirements
2. line between hardware characteristics and software design
   - software tasks could be cast into hardware
   - software decision module or hardware-hiding module?

3. software design decisions may not be appropriate anymore because of changes in
   - the hardware
   - the behaviour of the system
   - the behaviour of its users
4. all software modules include software design decisions
   - changes in any module may be motivated by efficiency or accuracy considerations

- such fuzziness is not acceptable!

Eliminating Fuzziness in the Top-Level Classification

- by referring to a precise software requirements document
  - specifies the lines between behaviour, hardware, and software decisions

ad 1: line between requirements and design

- if requirements specifies algorithm:
  algorithm is not software design decision
- if requirements specifies constraints only:
  program that implements algorithm is part of software design decision module

ad 2: line between hardware characteristics and software design

- interface specified in software requirements document
- draw line based on likelihood of changes
  - ▷ if likely to cast this software in hardware: classify as hardware-hiding module
  - ▷ otherwise: software design module
- conservative stance in SCR project:
  - ▷ drastic changes less likely than evolutionary changes
  - ▷ slight changes to hardware: hardware-hiding modules affected only
  - ▷ radical changes software→hardware: some software decision modules eliminated or reduced in size

ad 3: software design decisions may not be appropriate anymore because of changes in [. . .]

- module only in software decision module if
  it remains useful even when requirements document is changed
  (although possibly less efficient)

ad 4: all software modules include software design decisions

- module only in software decision module if
  its secrets do not include information from the requirements document

Second-Level Decomposition: Hardware-Hiding Module

1. extended computer module
2. device interface module

Extended Computer Module

- hides that part of the HW/SW interface that is likely to change
  - when computer modified
  - when computer replaced
  - same for operating system, if used
- example A-7E computer:
  - floating point unit or software simulation?
  - single / multi-processor?
- extended computer provides a virtual machine that can be implemented efficiently on all likely platforms

- primary secrets for A-7E computer:
  - number of processors
  - instruction set of the computer
  - capacity for concurrent operations

Device Interface Module

- hides that part of peripheral devices that is likely to change
  - each device might be replaced by an improved one capable of the same tasks
- example A-7E:
  - all angle-of-attack sensors measure angle between reference line on aircraft and the velocity of the air
  - they differ in: input format, timing, amount of noise

- module provides virtual devices
  - sometimes one virtual device corresponds to several hardware devices
  - sometimes the capabilities of a physical unit may change independently: then hide in different modules
- primary secrets for A-7E:
  - those characteristics of the present devices that
    - ▷ are documented in the requirements document
    - ▷ are not likely to be shared by replacement devices

Second-Level Decomposition: Behaviour-Hiding Module

1. function driver module
2. shared services module
   - supports function driver module

Function Driver Module

- a set of individual modules (“function drivers”)
- each function driver is sole controller of
  a set of closely related outputs
  - outputs related closely: if it is easier to describe their values together than individually
  - example: sine of an angle, cosine of same angle
- these outputs go to the virtual devices
- primary secrets: the rules determining the values of the outputs

Shared Services Module

- some aspects are common to two or more function drivers
  - A-7E: they control the same aircraft
  - odometer example: the display mode
- a shared services module hides one such aspect

Searching for a Behaviour-Hiding Module

- documentation users:
  will not know which aspects are shared
- documentation for the function driver modules:
  must have a reference to the shared services modules used
- start search:
  always with function driver

Second-Level Decomposition: Software Decision Module

1. application data type module
   - hides implementation of certain variables
2. physical model module
   - hides algorithms that simulate physical phenomena
3. data banker module
   - hides data-updating policies
4. system generation module
   - hides decisions that are postponed until system generation time
5. software utility module
   - hides algorithms used in several other modules

Application Data Type Module

- supplements data types by extended computer module
- provides data types useful for avionics
  that do not require a computer dependent implementation
- primary secrets: the data representation of the variables
  - variables can be used without units
  - where necessary, the modules provide unit conversion operators which deliver or accept values in specified units

Physical Model Module

- software requires estimates of quantities that cannot be measured directly,
  but can be computed from other observables
- primary secrets: the physical models
- secondary secrets: the implementations of the models

Data Banker Module

- most data:
  produced by one module and consumed by another
- usually: consumer gets value as up-to-date as practical
- data banker: middle-man, determines update policy
- if update policy changes:
  change neither producer nor consumer
- don't use data banker if consumer requires . . .
  - specific members of value sequence
  - values with a specific time (e.g., when an event occurs)

Some Data Update Policies

name	store	when new value produced
on demand	no	whenever a consumer requests the value
periodic	yes	periodically. consumer gets most recently stored value
event driven	yes	whenever data banker is notfied by an event of a possible change
conditional	yes	whenever a consumer requests the value, provided certain conditions are true. otherwise: previously store value

Choice of Updating Policies

- consumers' accuracy requirements
- how often consumers require the value
- max. wait that consumers can accept
- how often the value changes
- cost of producing a new value

- the policy decision does not depend on coding details of consumer or producer
  - data banker usually not rewritten if producer or consumer change

System Generation Module

- primary secrets:
  decisions that are postponed until system generation time
  - system generation parameters
  - choice among alternative implementations
- secondary secrets:
  - method used to generate executable code
  - representation of the postponed decisions
- these programs do not run on on-board computer
  - A-7E: cross-platform build

Software Utility Module

- primary secrets: the algorithms implementing common software functions and mathematical routines
  - resource monitor
  - square root, logarithm, . . .

Third-Level Decomposition: Extended Computer Module

1. data type module
2. data structure module
3. input/output module
4. computer state module
5. parallelism control module
6. sequence control module
7. diagnostics module (R)
8. virtual memory module (H)
9. interupt handler module (H)

Data Type Module

- implements variables and operators for real numbers, time periods, and bit strings
- primary secrets: data representations and data manipulation instructions built into the computer hardware
- secondary secrets:
  - how range and resolution requirements are used to determine representation
  - procedures for performing numeric operations
  - procedures for performing bitstring operations
  - how to compute the memory location of an array index given the array name and the element index

Computer State Module

- keeps track of current state of extended computer (operating / off / failed)
- signals relevant state changes to user programs
  - after extended computer is initialized,
    signals the event that starts initialization of the rest of the software
- primary secret: the way that the hardware detects and causes state changes

Diagnostics Module (R)

- provides diagnostic programs to test
  - the interrupt hardware
  - the I/O hardware
  - the memory
- use is restricted
  because it reveals secrets of the extended computer

Virtual Memory Module (H)

- presents a uniformly addressable virtual memory for use by
  - data type module
  - input/output module
  - sequence control module
- allows using virtual addresses for data and subprograms
- primary secrets:
  - hardware addressing methods for data and instructions in real memory
  - differences in the way that different areas of memory are addressed

- secondary secrets:
  - policy for allocating real memory to virtual addresses
  - programs that translate from virtual address references to real instruction sequences

Third-Level Decomposition: Device Interface Module

1. air data computer
   - how to read barometric altitude, true airspeed, and Mach number
2. angle of attack sensor
   - how to read angle of attack
3. audible signal device
4. computer fail device
5. Doppler radar set
6. flight information displays
7. forward looking radar
8. head-up display (HUD)

9. inertial measurement set (IMS/IMU)
10. panel
11. projected map display set (PMDS)
12. radar altimeter
13. shipboard inertial navigation system (SINS)
14. slew control
15. switch bank
16. TACAN
17. visual indicators
18. waypoint information system
19. weapon characteristics

20. weapon release system
    - how to ascertain weapon release actions the pilot has requested
21. weight on gear

- almost corresponds to hardware structure
    - exceptions are closely linked devices

Third-Level Decomposition: Function Driver Module

1. air data computer functions
2. audible signal functions
3. computer fail signal functions
4. Doppler radar functions
5. flight information display functions
6. forward looking radar functions
7. head-up display (HUD) functions
8. inertial measurement set (IMS/IMU) functions
9. panel functions
10. projected map display set (PMDS) functions

11. ships inertial navigation system (SINS) functions
12. visual indicator functions
13. weapon release functions
14. ground test functions

- input-only modules are missing here:
  - angle of attack sensor
  - radar altimeter
  - . . .
- each module can be divided further

Head-Up Display Functions

- primary secrets:
  - where the movable HUD symbols should be placed
  - whether a HUD symbol should be on, off, or blinking
  - what information should be displayed on the fixed-position displays

Inertial Measurement Set Functions

- primary secrets:
  - rules determining the scale to be used for the IMS velocity measurements
  - when to initialize the velocity measurements
  - how much to rotate the IMS for alignment

Panel Functions

- primary secrets:
  - what information should be displayed on panel window
  - when the enter light should be turned on

Third-Level Decomposition: Shared Services Module

1. mode determination module
2. stage director module
3. shared subroutine module
4. system value module
5. panel I/O support module
6. diagnostic I/O support module
7. event tailoring module

Mode Determination Module

- determines system modes
  (as defined in the requirements document)
- signals the occurence of mode transitions
- makes the identity of the current modes available
- primary secrets:
  the mode transition tables in the requirements document

System Value Module

- has a set of sub-modules
- each sub-module computes a set of values,

  some of which are used by more than one function driver
- primary secrets: the rules in the requirements that define the value that it computes
  - selection among several alternative sources
  - applying filters to values produced by other modules
  - imposing limits on a value calculated elsewhere

Third-Level Decomposition: Application Data Type Module

- examples:
  - angles (several versions)
  - distances
  - temperatures
  - local data types for device modules
  - STE (state transition event) variables

Third-Level Decomposition: Physical Model Module

1. earth model module
2. aircraft motion module
3. spatial relations module
4. human factors module
5. weapon behaviour module
6. target behaviour module
7. filter behaviour module

Earth Model Module

- primary secrets: models of the earth and its atmosphere
  - local gravity
  - curvature of the earth
  - pressure at sea level
  - magnetic variation
  - local terrain
  - rotation of the earth
  - coriolis force
  - atmospheric density

Aircraft Motion Module

- primary secrets: models of the aircraft's motion
- used to calculate aircraft position, velocity, attitude from observable inputs

Spatial Relations Module

- primary secrets: models of three-dimensional space
- used to perform coordinate transformations, angle calculations, distance calculations

Human Factors Module

- primary secrets: models of pilot reaction time and perception of simulated continuous motion
- determines the update frequency for symbols on a display

Weapon Behaviour Module

- primary secrets: models used to predict weapon behaviour after release

Third-Level Decomposition: Data Banker Module

- one for each real-time data item
- value always up-to-date
- secret: when to compute up-to-date value

Third-Level Decomposition: System Generation Module

- . . .
  - (these programs do not run on on-board computer)

Third-Level Decomposition: Software Utility Module

- resource monitor module
- other shared resources
  - square root
  - logarithm
  - ...

Results of the A-7E Module Guide

- module guide is $<$ 30 pages
  - every project member must and can read it
- experience:
  - important to organize the guide by secrets, not by interfaces or by roles
  - software requirements document was essential for disambiguating choices in the guide's structure

- implementation of several subsets on a flight simulator
- integration testing of the first "minimal useful subset":
  - ◦ took a week only
  - ◦ nine bugs found
    - ▷ each in a single module only
    - ▷ each quickly fixed

  Dave Weiss: "like a breeze!"

- guide often used as a *document template* for other projects applying the method

3.3 Hierarchical Software Structures

Text for Chapter 3.3

[Par74] Parnas, D. *On a 'buzzword': Hierarchical structure*. In "IFIP Congress 74", pp. 336–339. North-Holland (1974). Reprinted in [HoWe01].

[HoWe01] Hoffman, D. M. and Weiss, D. M., editors. *Software Fundamentals – Collected Papers by David L. Parnas*. Addison-Wesley (Mar. 2001).

Additional Background for Chapter 3.3

[Cou85] Courtois, P.-J. *On time an space decomposition of complex structures*. Commun. ACM **28**(6), 590–603 (June 1985).

"Courtois hierarchy" of structures which are complex in time and space.

Structure

- partial description of a system, showing
  - a division into *parts*
  - a *relation* between the parts

- graphs can describe a structure

Hierarchical Structure

- a structure with no loops in its relation's graph:
  - $P_0 = \{\alpha \in P \mid \neg\exists\, \beta \in P \,.\, R(\alpha, \beta)\}$
  - $P_i = \{\alpha \in P \mid \exists\, \beta \in P_{i-1} \,.\, R(\alpha, \beta) \wedge$
    $\neg\exists\, j \in \mathbb{N},\ \gamma \in P_j \,.\, R(\alpha, \gamma)\ \wedge\ j \geq i\}$

- note: hierarchy $\neq$ tree
- meaning of "hierarchical structure"?
  - meaning of parts?
  - meaning of relation?

Different Kinds of Software Hierarchies

- module decomposition hierarchy
- calls hierarchy
- uses hierarchy
- Courtois hierarchy

- gives-work-to hierarchy
- created hierarchy
- resource allocation hierarchy
- can-be-accessed-by hierarchy

Module Decomposition Hierarchy

- kind of structure:
  - parts: write-time modules
  - relation: part-of
- time: early design time
- this structure is always a hierarchy
  - never loop in “part-of”

Calls Hierarchy

- kind of structure:
  - parts: programs
  - relation: calls
- time: design time
- hierarchical relation forbids recursion
  - usually not a useful hierarchy

Uses Hierarchy

- kind of structure:
  - parts: programs
  - relation: uses (i.e., requires-the-presence-of)
- time: design time
- definition of "uses":
  Given a program A with specification S and a program B,
  A uses B iff
  A cannot satisfy S unless B is present and functioning correctly

- example: list insert routine
  - uses getNextElem, setNextElem routines
  - calls nullPointerException routine
  - does *not* “use” nullPointerException routine
- example: window manager with call-backs
  - application passes address of draw() program to window manager
  - application responsible for drawing sub-area when draw() called
  - window manager calls draw()
  - window manager does *not* “use” draw()
- example: layers of communication services
  - the higher layer uses the services of the lower layer
  - messages are passed in both directions
    (reqest, indication, response, confirm)

- if a structure is a uses hierarchy:
  levels define virtual machines
- useful for “ease of subsetting” (see later)

Courtois Hierarchy

- kind of structure:
  - parts: operations
  - relation: takes more time and occurs less frequently than
- time: run time

Courtois: Decomposition of Complex Structures

- domains with complex structures:
  - physics
  - social science
  - economy
  - computer science
- sometimes easily decomposable in time and space
  - concentrations in chemical reactions
    - ▷ differential equation suitable
    - ▷ large number of molecules allows to assume continuum

- hierarchical decomposition difficult when
  - time or size scales are not far apart
  - interesting behavioural properties are related to rare events caused by weak interactions within the system
  - events at many scales of time or size from each other nevertheless have a non-negligible influence on each other
- a hierarchical decomposition should ideally have:
  - *time and size scales far apart between levels*
  - . . .
- (Courtois describes how one can model structures even when they are not easily decomposable)

Some More Kinds of Software Hierarchies

- module decomposition hierarchy
- calls hierarchy
- uses hierarchy
- Courtois hierarchy

some more kinds:

- gives-work-to hierarchy
- created hierarchy
- resource allocation hierarchy
- can-be-accessed-by hierarchy

Gives-Work-To Hierarchy

- kind of structure:
  - parts: processes
  - relation: gives an assignment to
- time: run time
- found in T.H.E. operating system
  - organized as set of parallel sequential processes
  - processes exchange work assignments and information by message passing
  - processes are in hierarchical gives-work-to relation
- useful for guaranteeing termination, but neither necessary nor sufficient for this

Created Hierarchy

- kind of structure:
  - parts: processes
  - relation: created
- time: run time
- must be a hierarchy (parent is older than child)
- is a tree
  - why? (team work in creating progeny is accepted practice)
- sometimes implies unnecessary restrictions
  - example: parent cannot die until all progeny die

Resource Allocation Hierarchy

- kind of structure:
  - parts: processes
  - relation: allocate-a-resource-to or
    owns-the-resources-of
- time: run time
- applicable with dynamic resource administration only
- “allocate to” vs. “controls”: the question of pre-emption
- example: hierarchical money budgets for country, state, university, department, . . .

- advantages:
  - interference reduced or eliminated
  - deadlock possibilities reduced
- disadvantages:
  - poor utilization when load unbalanced
  - high overhead when resources are tight
    (especially with many levels)

Can-Be-Accessed-By Hierarchy

- kind of structure:
  - parts: programs
  - relation: can-be-accessed-by
- time: design time
- important to security and reliability
- example: the “rings” of Multics
  - generalization of supervisor/user level of CPU execution
  - is even complete ordering
- a hierarchy prevents some useful accessability patterns

Many Kinds of Software Hierarchies Possible

- not all of these relations must form a hierarchy!
- you may choose some of these relations to form a hierarchy
- *if you confuse these relations, you will mess up your design*
  - you then force a hierarchy on a relation that should not be a hierarchy
    - ▷ T.H.E.: uses hierarchy and gives-work-to hierarchy coincided
    - ▷ write-time module hierarchy and uses hierarchy of course should not coincide

- ▷ write-time module hierarchy and created hierarchy should not coincide if the latter imposes constraints (object creation in OO!)

Example: ISO OSI Basic Reference Model

- basic reference model for communication systems
  - 7 layers
- is a uses hierarchy
- should not be implemented as a gives-work-to hierarchy
  - then lots of message passing between layers
  - much too inefficient

Uses Hierarchy and Courtois Hierarchy

- in practice they usually coincide
  - programs that require few or no other programs to function run short and are executed often
  - programs that run long and only a few times require many other programs to function
- except: the handling of exceptions
  - interrupts
  - reboot (seldom, needed by all programs)
  - . . .
- *if the above is not the case then usually something is wrong!*

3.4 Designing Software for Ease of Extension and Contraction

Text for Chapter 3.4

[Par79] Parnas, D. L. *Designing software for ease of extension and contraction*. IEEE Trans. Softw. Eng. **SE-5**(2), 128–138 (Mar. 1979).

Additional Background for Chapter 3.4

[Par76] Parnas, D. L. *On the design and development of program families*. IEEE Trans. Softw. Eng. **2**(1), 1–9 (Mar. 1976).

Stepwise refinement vs. information hiding; families of programs.

Motivation

some common complaints about software systems:

- deliver early release with subset of functionality?
  $\longrightarrow$ the subset won't work until everything works
- add simple capability?
  $\longrightarrow$ rewrite most of the current code
- remove unneeded capability?
  $\longrightarrow$ rewrite much of the current code

A Family of Programs

- usually you don't write a single program, but a *family* of programs
- families of systems: Chapter 4
- here special case: families of programs where
  - some members are subsets of other members, or
  - several members share a large common subset

Alternatives for the Software Producer

- a “super” system
  - generality costs
    - ▷ memory, speed: still important for embedded systems
    - ▷ difference to mathematics
- a system for the “average” user
  - doesn’t really fit for anybody
- a set of independently developed systems
  - with subtle differences $\longrightarrow$ maintenance nightmare
- a subsettable “super” system
  - each family member offers a subset
    of services of the largest member

A Subsettable System

- individual installations only pay for what they need
  - computer resources
  - marketing
- incremental implementation possible
- allows for fail-soft subsets
- ability to contract by deleting whole programs, not by modifying programs
- ability to extend by adding programs, without changing programs

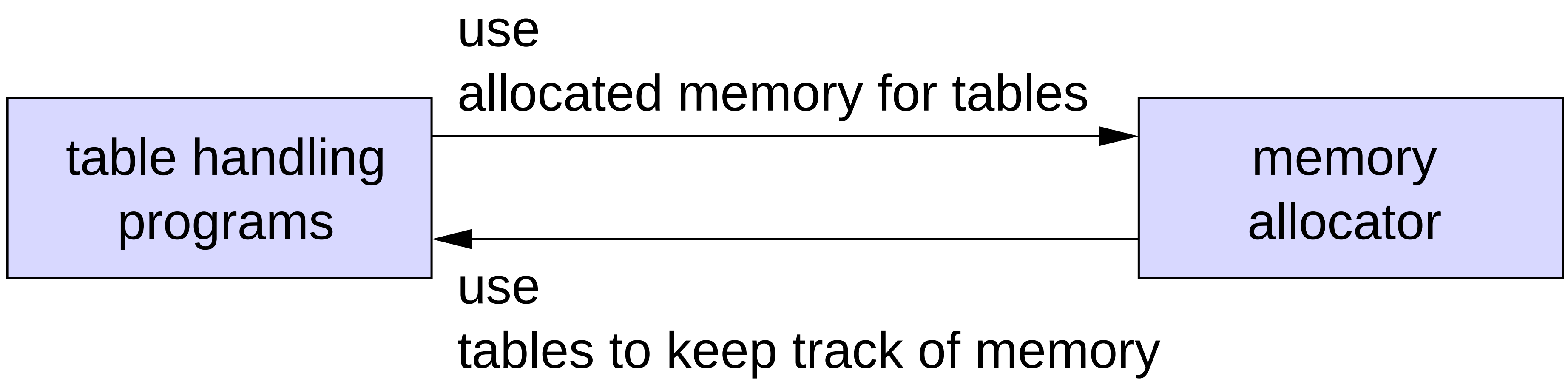
The Uses Hierarchy, Again

- is the key to subsets!

- kind of structure:
  - parts: programs (not modules)
  - relation: uses (i.e., requires-the-presence-of)
- time: design time
- definition of “uses”:
  Given a program A with specification S and a program B,
  A uses B iff
  A cannot satisfy S unless B is present and functioning correctly

Design Error: Loops in the Uses Relation

example:



- neither works until both work
- if either is removed, the other no longer works
- should memory allocator build own tables?
  - code duplication

example (from Multics):

- virtual memory uses file system
- file system uses virtual memory

Basic Steps in the Design of a Subsettable System

1. identify the subsets
2. make list of programs belonging to each module
3. decide on uses matrix for the programs
4. construct the uses hierarchy from the matrix

Identify the Subsets

- during requirements definition
- search for minimal useful subset
- search for minimal useful increments
  - even if it appears trivial now
- each increment later becomes a write-time module in the design

Make List of Programs Belonging to Each Module

- access programs
- internal programs
  - cannot be used directly by outside programs
  - can use other programs
- main programs
  - cannot be used (are top-level)
  - can use other programs

Basic Steps in the Design of a Subsettable System

1. identify the subsets
2. make list of programs belonging to each module
3. decide on uses matrix for the programs
4. construct the uses hierarchy from the matrix

Decide on Uses Matrix for the Programs

- three possibilities for each pair (A, B)
  - A may use B
  - B may use A
  - neither may use the other

Conditions for Allowing Program A to Use Program B

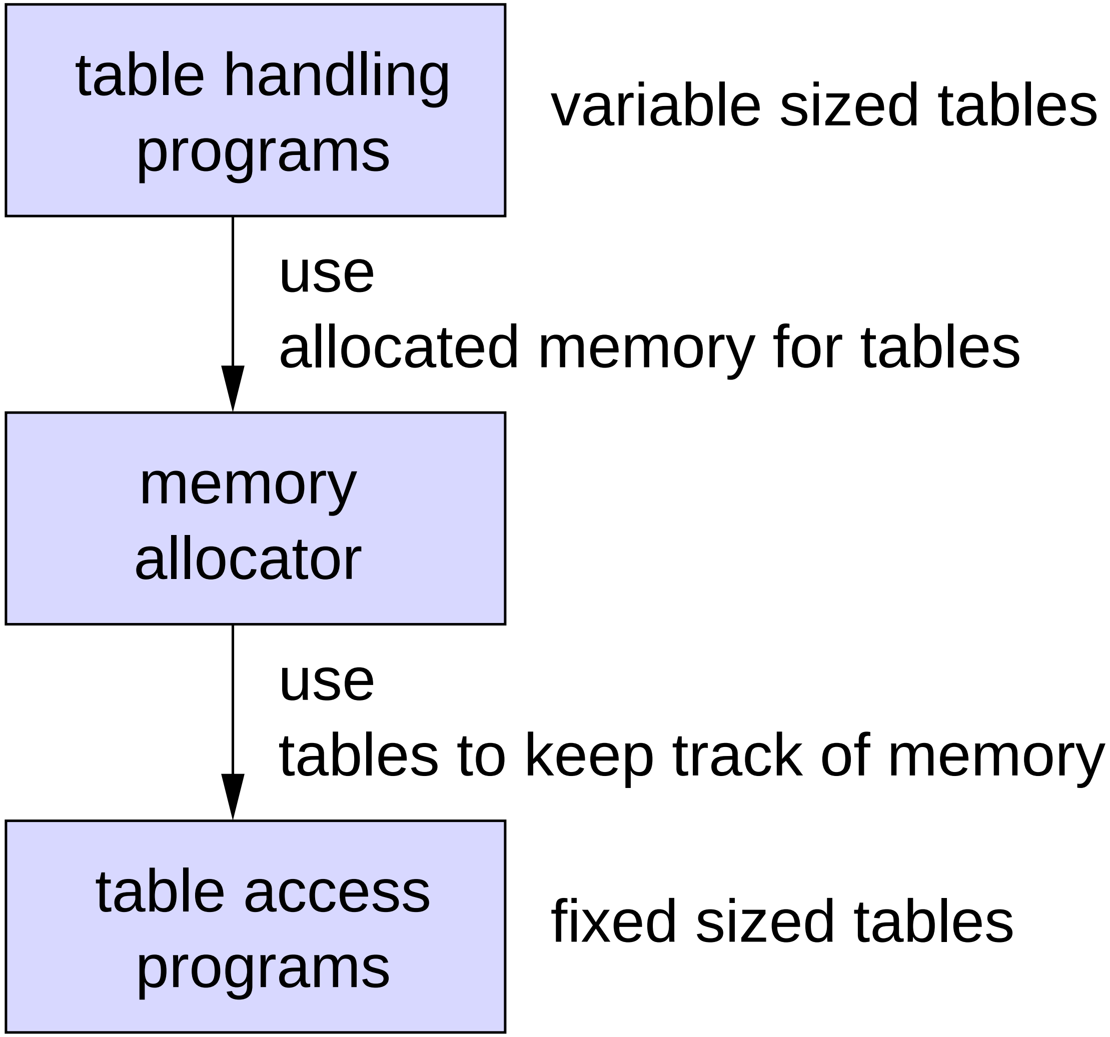
- A is simpler because it uses B
- B is not more complex because it is not allowed to use A
- there is a useful subset containing B and not A
- there are no useful subsets containing A and not B

- all conditions must be satisfied

Construct the Uses Hierarchy from the Matrix

- could be done by a tool
  - see Ada's "with" clause to make the uses relation explicit
- make list of programs at level 0
  - they don't use other programs
- work up from there
  - level 1 programs use only level 0 programs
  - level 2 programs . . .
- the uses matrix and hierarchy must be maintained, of course

Conflict Removal: Sandwiching



- message:

  a level (in the uses hierarchy)
  is not a module (in the write-time hierarchy)

  ◦ uses relationship is between programs, not modules

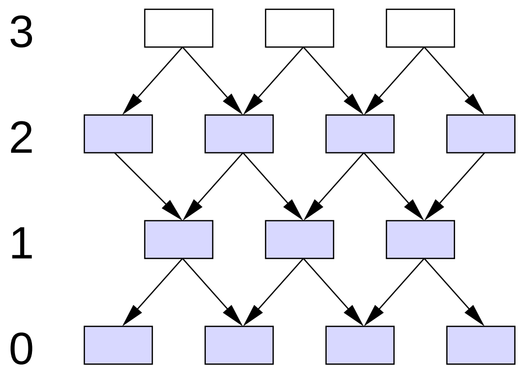
  ◦ there are no "layers of abstraction"

  ◦ in a subsetted system,
  there may be subsets of the programs in the modules

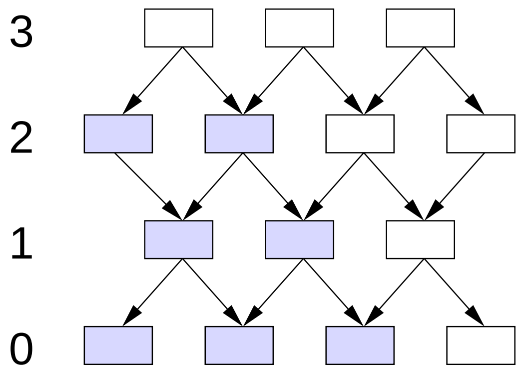
  ▷ the designer of $each$ module must identify the useful subsets

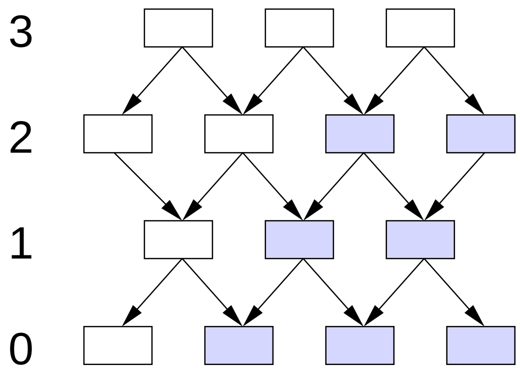
Deriving Subsets from the Uses Relation

- any level is a subset



- can also omit parts of levels





Levels and Virtual Machines

- def. virtual machine: a set of variables and operations, implemented in software
- each level is a virtual machine
  - applications programs are simpler: they use virtual machine programs
- upper level machines are *less powerful*
  - resources used to implement a VM must not be available to a program that uses the VM
  - upper level machines more specialized
- upper level machines are more convenient and safer

Evaluation Criteria for a Uses Hierarchy

1. all desirable subsets?
2. no duplicated or almost alike programs?
3. is it simple?

Getting All Desirable Subsets

- principle of minimal steps
  - start with minimal useful subset
  - minimal useful increments
- examples of violation:
  - RC4000 operating system combined synchronization and message passing
  - Hydra operating system combined parameter passing and run-time type checking

The One, Fixed, Variable Pattern

- a common, useful pattern for designing a uses hierarchy
- three levels of operations:
  - operations on *one* item
  - operations on a *fixed* number of similar items
  - operations on a *variable* number of similar items
- you might want to have three subsets
- language/library support
  for "fixed", "variable" supersets
  of "one" data element
  - C++, Java, . . .

Example: an Address Processing System

- read, store, and write out lists of addresses
- example taken from [Par79]

Information in an Address

- last name
- given names
- organization
- internal identifier
- street address or P.O. box
- city or mail unit identifier
- state
- Zip code
- title
- branch of service if military
- GS grade if civil service

- each field may be empty

Basic Assumptions

- the items on previous slide will be processed by all application programs
- the input formats are subject to change
- the output formats are subject to change
- choice of input/output format for different systems:
  - fixed format
  - run-time choice from a fixed set
  - user-specified format definition language

  } (one/fixed/variable)
- representation of addresses in main memory will vary

- most systems: only a subset of addresses in main memory at any one time
  - number needed may vary
  - some systems: number needed may vary at run-time

Proposed Design Decisions

- input and output programs will be table driven
  - table specifies format
  - secret of input and output modules:
    content and organization of format tables
- secret of address storage module (ASM):
  representation of addresses in main memory
  - changing a part of an address is cheaper than
    growing or shrinking the address table

- address file module (AFM):
  used if more addresses than main memory
  - interface compatible to ASM
  - provides additional operations for efficient sequential iteration
- implementation of AFM has ASM, BFM as submodule
  - block file module (BFM):
    stores data blocks (size of at least an address),
    does not look at content
  - the ASM within the AFM has two interfaces:
    - ▷ "normal" interface: addresses and their fields
    - ▷ interface for blocks of contiguous storage, input/output
  - BFM might be part of operating system

Access Programs of “Normal” Interface of ASM

addTit:	$asm \times integer \times string \rightarrow asm$	fetTit:	$asm \times integer \rightarrow string$
addGN:	$asm \times integer \times string \rightarrow asm$	fetGN:	$asm \times integer \rightarrow string$
addLN:	$asm \times integer \times string \rightarrow asm$	fetLN:	$asm \times integer \rightarrow string$
addServ:	$asm \times integer \times string \rightarrow asm$	fetServ:	$asm \times integer \rightarrow string$
addBOrC:	$asm \times integer \times string \rightarrow asm$	fetBOrC:	$asm \times integer \rightarrow string$
addCOrA:	$asm \times integer \times string \rightarrow asm$	fetCOrA:	$asm \times integer \rightarrow string$
addSOrP:	$asm \times integer \times string \rightarrow asm$	fetSOrP:	$asm \times integer \rightarrow string$
addCity:	$asm \times integer \times string \rightarrow asm$	fetCity:	$asm \times integer \rightarrow string$
addState:	$asm \times integer \times string \rightarrow asm$	fetState:	$asm \times integer \rightarrow string$
addZip:	$asm \times integer \times string \rightarrow asm$	fetZip:	$asm \times integer \rightarrow string$
addGsL:	$asm \times integer \times string \rightarrow asm$	fetGsL:	$asm \times integer \rightarrow string$
setNum:	$asm \times integer \rightarrow asm$	fetNum:	$asm \times integer$

Component Programs of Address Input Module

InAd:	Reads in an address in the currently selected format and calls ASM or AFM programs to store it.
InFSel:	Selects a format from an existing set of format tables for InAd. There is always a format selected.
InFCr:	Adds a new format to the tables used by InFSel. The format is specified in a "format language". Selection is not changed.
InTabExt:	Adds a blank table to the set of input format tables.
InTabChg:	Rewrites a table in the input format tables. Selection is not changed.
InFDel:	Deletes a table from the set of format tables. The selected format cannot be deleted.
InAdSel:	Reads in an address using one of a set of formats. Choice is specified by an integer parameter.
InAdFo:	Reads in an address in a format specified as one of its parameters (a string in the format definition language). The format is selected and added to the tables and subsequent addresses could be read in using InAd.

Component Programs of Address Output Module

- **OutAd:** Prints out an address in the currently selected format. The information is in an ASM and identfied by its position there.
- **OutFSel:** Selects a format from an existing set of format tables for OutAd. There is always a format selected.
- **OutFCr:** Adds a new format to the tables used by OutFSel. The format is specified in a "format language". Selection is not changed.
- **OutTabExt:** Adds a blank table to the set of output format tables.
- **OutTabChg:** Rewrites a table in the output format tables. Selection is not changed.
- **OutFDel:** Deletes a table from the set of format tables. The selected format cannot be deleted.
- **OutAdSel:** Prints out an address using one of a set of formats. Choice is specified by an integer parameter.
- **OutAdFo:** Prints out an address in a format specified as one of its parameters (a string in the format definition language). The format is selected and added to the tables and subsequent addresses could be printed using OutAd.

Component Programs of Address Storage Module

Fet<CompName>:	Read information from an address store. (See Slide 333.)
Add<CompName>:	Write information in an address store. (See Slide 333.)
GetBlock:	Takes an integer parameter, returns a storage block.
SetBlock:	Takes a storage block and an integer. Changes the contents of an address store – reflected by the Fet<CN> programs.
AsmExt:	Extends an address store by appending a new address with empty components at the end of the address store.
AsmShr:	"Shrinks" the address store.
AsmCr:	Creates a new address store. The parameter specifies the number of components. All components are initially empty.
AsmDel:	Deletes an existing address store.

Component Programs of Block File Module

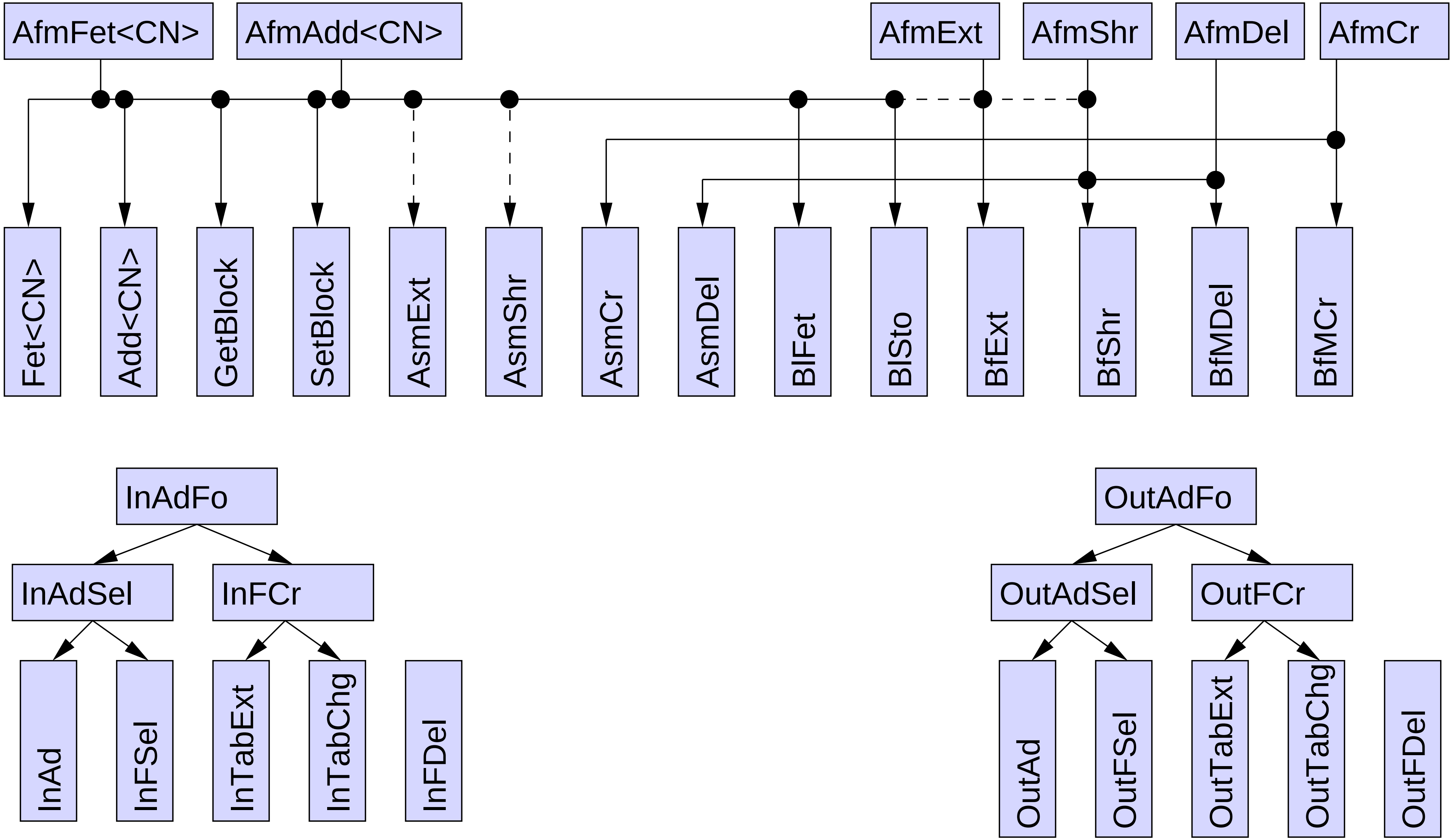
BlFet:	Takes an integer and returns a “block”.
BlSto:	Takes a block and an integer and stores the block.
BfExt:	Extends BFM by adding additional blocks to its capacity.
BfShr:	Reduces the size of the BFM by removing some blocks.
BfMCr:	Creates a file of blocks.
BfMDel:	Deletes an existing file of blocks.

Component Programs of Address File Module

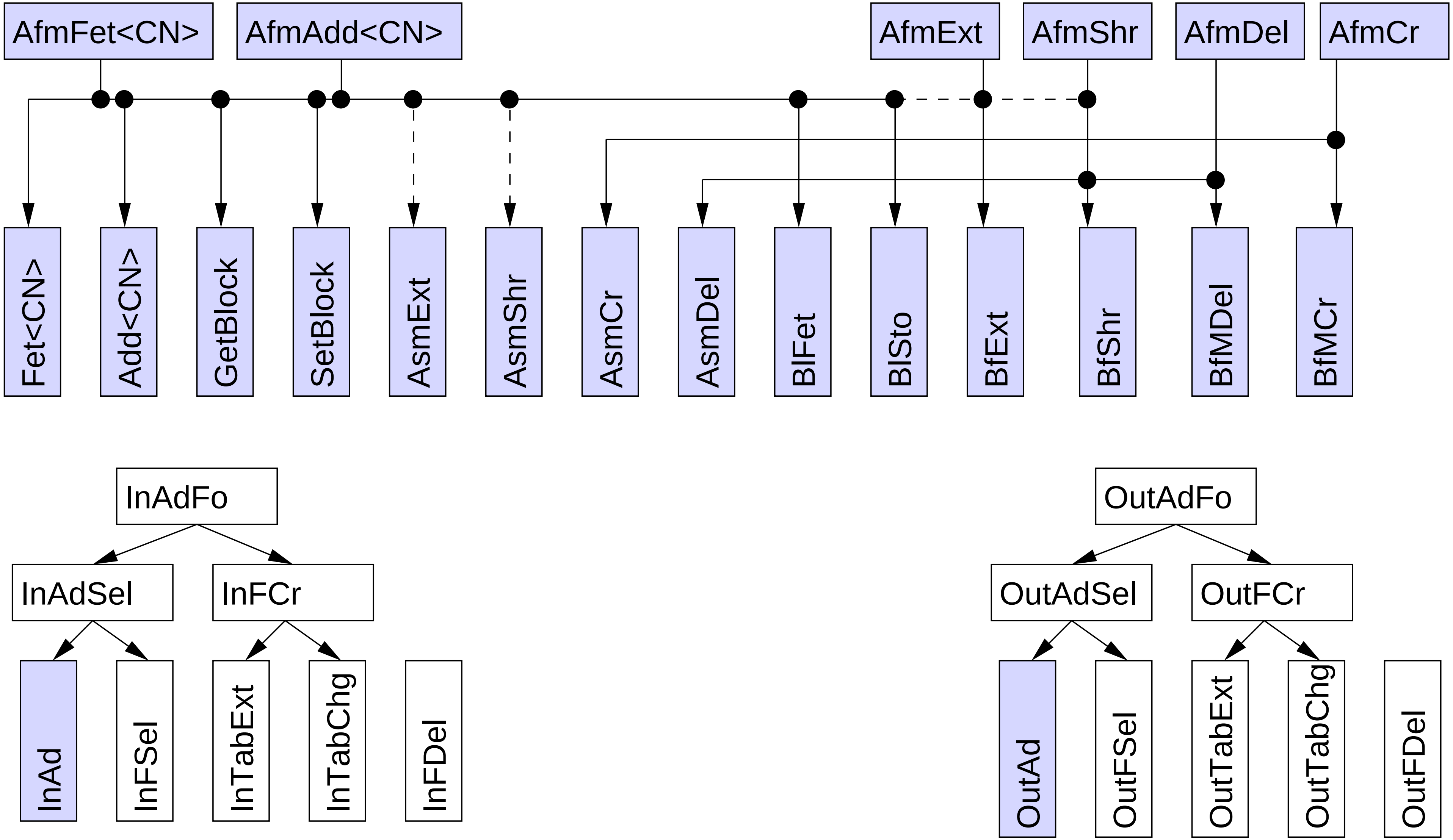
- provides all ASM programs except GetBlock and SetBlock.
- the programs are renamed as follows:

AfmFet<CompName>:	As in ASM.
AfmAdd<CompName>:	As in ASM.
AfmExt:	As in BFM.
AfmShr:	As in BFM.
AfmCr:	As in BFM.
AfmDel:	As in BFM.

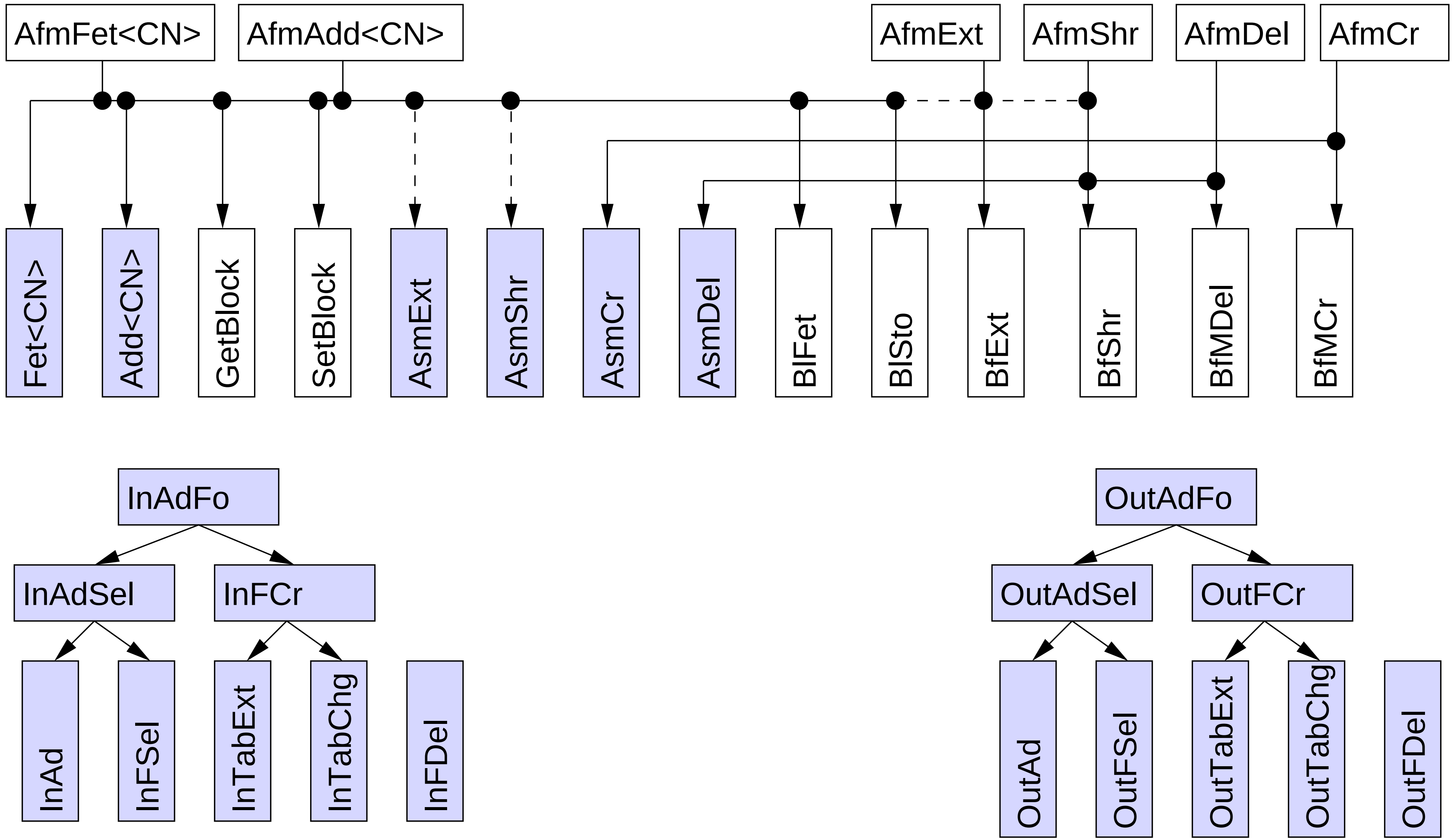
Uses Relation of the System



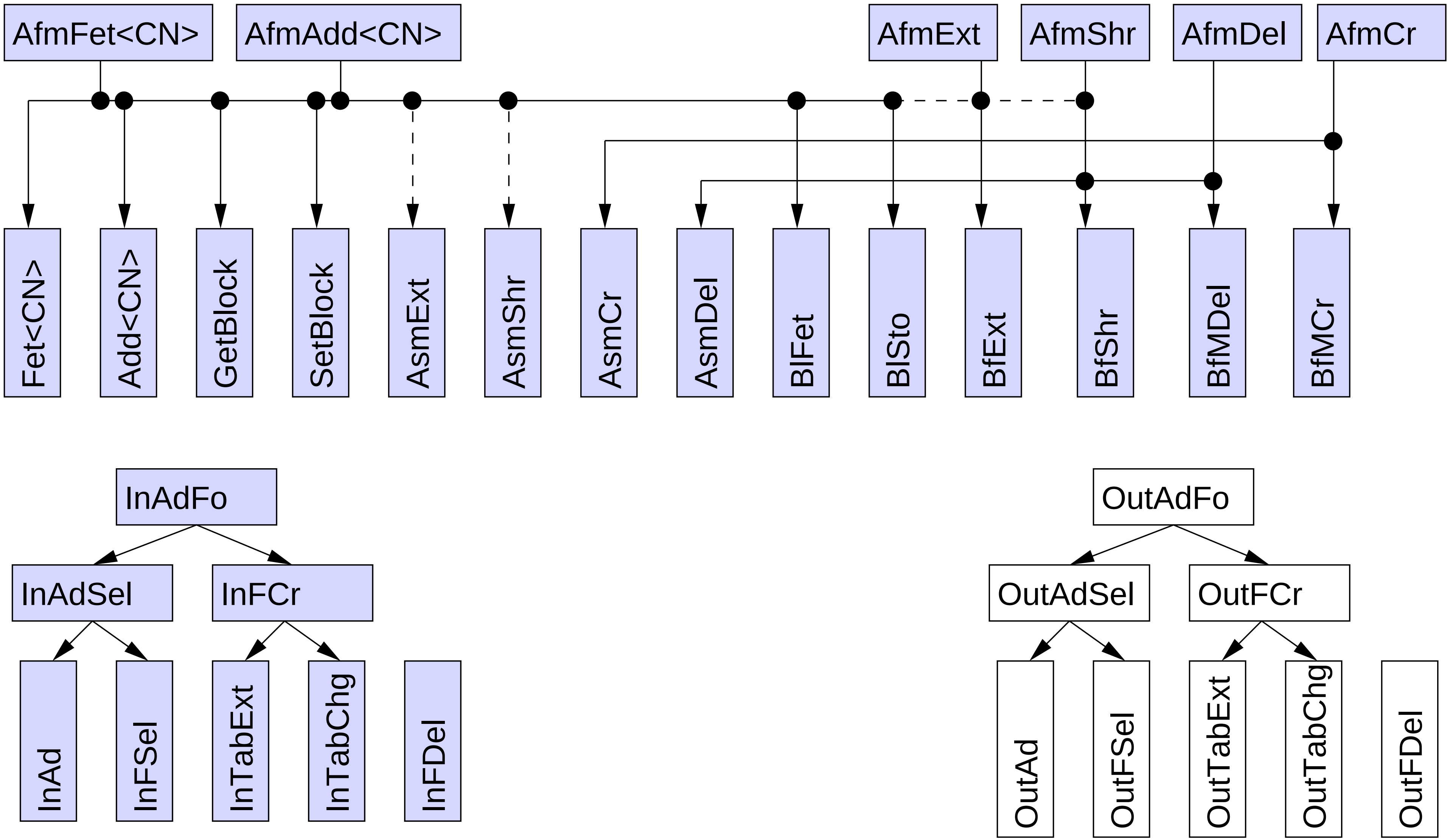
Subset: Addresses in a Single Format



Subset: Small Set of Addresses



Subset: Query-Only System



3.5 Design of Abstract Interfaces

Text for Chapter 3.5

[HBPP81] Heninger Britton, K., Parker, R. A., and Parnas, D. L. *A procedure for designing abstract interfaces for device interface modules*. In “Proc. of the 5th Int’l. Conf. on Software Engineering – ICSE 5”, pp. 195–204 (Mar. 1981).

Additional Background for Chapter 3.5

[Par77] Parnas, D. L. *Use of abstract interfaces in the development of software for embedded computer systems*. NRL Report 8047, Naval Research Lab., Washington DC, USA (3 June 1977). Reprinted in Infotech State of the Art Report, Structured System Development, Infotech International, 1979.

A predecessor report of [HBPP81] with more examples.

[PaWe85] Parnas, D. L. and Weiss, D. M. *Active design reviews: Principles and practices*. In “Proc. of the 8th Int'l Conf. on Software Engineering – ICSE 8”, London (Aug. 1985).

How to organize the review of documentation.

Applying Information Hiding to Embedded Systems

- the external interface is what is likely to change
- use an abstract interface to hide the actual interface

Motivation for Abstract Interface Design Rules

- much of the complexity of embedded real-time software: special-purpose hardware devices
  - example A-7 avionics:
    - ▷ 21 devices, arbitrary interfaces (value encodings, timing quirks)
    - ▷ changes during and after development
    - ▷ device “adequate” but does not meet specification exactly
    - ▷ device replaced by better one
    - ▷ new connections between devices
- hide details inside device interface modules
- but *which* details?

Device Interface Modules

- software module structure:
  1. hardware-hiding module
     1.1 extended computer module
     1.2 device interface module
        1.2.1 air data computer
        1.2.2 angle of attack sensor
        ...
  2. behaviour-hiding module
  3. software decision module
- provide virtual devices
  - example: virtual altimeter
    - ▷ provides value of type `range` instead of bit string
    - ▷ raw data is read, scaled, corrected, and filtered

Design Goals for Device Interface Modules

- confine changes
- simplify the rest of the software
- enforce disciplined use of resources
- code sharing
- efficient use of devices

Definitions

for:

- interface
- abstraction
- abstract interface
- device interface module
- secret of a device interface module

Definition: Interface

Definition 15 (Interface)
The interface between two programs consists of the set of assumptions *that* each *programmer needs to make about the other program in order to demonstrate the correctness of his own program.*

- more than syntax
- analogous definition for the interface program–device

Definition: Abstraction

Definition 16 (Abstraction)
An abstraction of a set of objects is a description that applies equally well to any one of them.

- each object is an instance of the abstraction
- an abstraction models some aspects, but not all
- example: differential equations
  (electrical circuits, collections of springs and weights, . . .)

Appropriateness of an Abstraction

- appropriate for a given purpose:
  easier to study the abstraction than the actual system
  - example: map *map*

Definition: Abstract Interface

Definition 17 (Abstract interface)
An abstract interface is
an abstraction that represents more than one interface.

- exactly the assumptions included in all of the interfaces that it represents

Definition: Device Interface Module

Definition 18 (Device interface module)
A device interface module is a set of programs that translate between the abstract interface and the actual hardware interface.

- implementation possible only if all assumptions in abstract interface are true of actual interface

Definition: Secret of a Device Interface Module

Definition 19 (Secret of a device interface module)
A secret of a device interface module is an assumption about the actual device that user programs is not allowed to make.

- secret is an information about the current device which needs not be true for others

Undesired Event Assumptions

- interface between programs A, B includes assumptions of A about B and of B about A
- B': does not make any assumptions about A
  - extra error checking and reporting in B'; more expensive
- development version of A-7:
  device interface modules that assume
  undesired events by user programs can occur
- production version of A-7: checking omitted
  - compiler switch
- error checks in the requirements: never omitted

Design Approach

- two partially redundant descriptions of the interface:
  1. assumption list characterizing the virtual device
  2. programming constructs embodying the assumptions
- review and iterate

Description 1: Assumption List Characterizing the Virtual Device

- study devices available or under development
  - advertisements of vendors
  - journals
  - . . .
- make list of common characteristics
  - device capabilities
  - modes
  - information requirements
  - behaviour
  - proper use

- these are the assumptions
- example:
  "The device provides information from which barometric altitude can be determined."
  - only devices satisfying this assumption
    will replace the current barometric altitude sensor
  - no common assumption on the format of the information
- many assuptions appear inocuous
  - record anyway
  - review might prove them false

Description 2: Programming Constructs Embodying the Assumptions

- access programs
  - name, parameter types, value returned
  - limitations
  - effect on the device
- signalling events

The Descriptions are Partially Redundant

- specifications for the programming constructs imply the assumptions
- access program specifications additionally provide form of data exchange
  - example:
    altimeter device interface module
    might not provide barometric altitude directly,
    but two or three quantities from which it can be computed
  - a design change would change the access program specification
    but not the assumption list

Different Purposes of the Two Descriptions

1. assumption list: state assumptions explicitly
   - explicit: invalid assumptions are easier to detect
   - prose: easier to review for non-programmers
   - review by programmers, users, hardware engineers
     - ▷ valid?
     - ▷ general enough?
2. programming constructs: direct use in user programs
   - review by programmers
     who have worked with similar programs
     - ▷ typical user programs supported well?
     - ▷ efficient implementation possible?

- consistency is essential
  - assumptions clearly embodied in the programming construct specifications
  - programming construct specifications should not imply additional capabilities

Reviews

- ask the expert *why* something *cannot* change
  - “active design review”
  - for details see [PaWe85]

Iterative Process for the A-7

- tried to list assumptions first
- many subtle assumptions became apparent only when designing programming constructs
- review of assumptions revealed errors in programming constructs
- several cycles of review
  - internally at NRL (several times)
  - by A-7 maintenance team (informal, then formal)

Example: Development of the Air Data Computer (ADC)

- a sensor that measures
  - barometric altitude
  - true airspeed
  - the mach number representation of airspeed

Excerpt of an Early Draft

assumption list

1. The ADC provides a measure of barometric altitude, mach number, and true airspeed.
2. The above measurements are based on a common set of sensors. Therefore an inaccuracy in one ADC sensor may affect any of these outputs.
3. The ADC provides an indication if any of its sensors are not functioning properly.
4. The measurements are made assuming a sea level pressure of 29.92 inches of mercury.

access program table

program name	parameter type	parameter information
G_ADC_ALTITUDE	p1:distance;O	altitude assuming 29.92 inches sea level pressure
G_ADC_MACH_INDEX	p1:mach;O	mach
G_ADC_TRUE_AIRSPEED	p1:speed;O	true airspeed
G_ADC_FAIL_INDICATOR	p1:logical;O	true if ADC failed

Problems with This Early Draft

- current ADC hardware and most replacement devices have built-in test capability – no access
- when ADC is in failed state, no values specified for access functions
- ranges of measured values not specified
- user programs must poll to detect changes in validity
- not clear whether module performs device-dependent corrections to the raw sensor values

Excerpt of Draft for Formal Review

assumption list

1. The ADC provides measurements of the barometric altitude, true airspeed, and the mach number representation of the airspeed of the aircraft. Any known measurement errors are compensated for within the module. Altitude measurements are made assuming that the air pressure at sea level is 29.92 inches of mercury.
2. All of these measurements are based on a common set of sensors; therefore an inaccuracy in one ADC sensor will affect all measurements.
3. User programs are notified by means of an event when the ADC hardware fails. If the access programs for barometric altitude, true airspeed, and mach number are called during an ADC failure, the last valid measurements (stale values) are provided.
4. The ADC is capable of performing a self-test upon command from the software. The result of this test is returned to the software.
5. The minimum measureable value for mach number and true airspeed is zero. The minimum barometric altitude measureable is fixed after system generation time, as are the maximum value and resolution for all measurements.

access program table

program name	parameter type	parameter information
G_ADC_BARO_ALTITUDE	p1:distance;O	corrected altitude assuming sea level pressure = 29.92 inches mercury
G_ADC_MACH_INDEX	p1:mach;O	corrected mach
G_ADC_RELIABILITY	p1:logical;O	true if ADC reliable
G_ADC_TRUE_AIRSPEED	p1:speed;O	corrected true airspeed
TEST_ADC	p1:logical;O	true if ADC passed self test

event table

event	when signalled
@T(ADC unreliable)	When “ADC reliable” changes from true to false

Problems with the Later Draft

- correction for actual sea level pressure is device-dependent
  - therefore better do inside DIM
  - future hardware may do this automatically
- only one reliability indicator for three values
  - current hardware: only one indicator; OK
  - future hardware: might have independent sensors
- some devices might not be able to measure speeds as low as zero

Excerpt of Published Version

assumption list

1. The ADC provides measurements of the barometric altitude, true airspeed, and the mach number representation of the airspeed of the aircraft (mach index). Any known measurement errors are compensated for within the module. <DELETED>

<DELETED>

2. User programs are notified by means of events when one or more of the outputs are unavailable. A user program can also inquire about the reliability of individual outputs. If the access programs for barometric altitude, true airspeed, and mach number are called while the values are unreliable, the last valid measurements (stale values) are provided.
3. The ADC is capable of performing a self-test upon command from a user program. The result of this test is returned to the user program.
4. The minimum, maximum, and resolution of all ADC measurements are fixed after system generation time.
5. The ADC will compute its outputs on the basis of a value for Sea Level Pressure

(SLP) supplied to it by a user program. If no value is provided, an SLP of 29.92 will be assumed.

access program table

program name	parameter type	parameter information
G_ADC_ALTITUDE	p1:distance;O	corrected altitude assuming SLP=29.92 or user supplied SLP
	p2:logical;O	true if altitude valid
G_ADC_MACH_INDEX	p1:mach;O	corrected mach
	p2:logical;O	true if mach valid
G_ADC_TRUE_AIRSPEED	p1:speed;O	corrected true airspeed
	p2:logical;O	true if true airspeed valid
S_ADC_SLP	p1:pressure;I	sea level pressure
TEST_ADC	p1:logical;O	true if ADC passed self test

event table

event	when signalled
@T(altitude invalid)	When “altitude valid” changes from true to false
@T(airspeed invalid)	When “true airspeed valid” changes from true to false
@T(mach invalid)	When “mach valid” changes from true to false

Design Problems – Tradeoffs and Compromises

- design goals in conflict:
  - small device interface modules
  - device-independent user programs
  - efficiency

- ultimate goal:
  minimize expected cost of the software
  over its entire period of use

Major Variations Among Available Devices

- sometimes differences are more than skin deep
  - example: Inertial Measurement Set (IMS)
- full simulation does not separate concerns
- solution: two modules

Devices with Characteristics that Change Independently

- failure to fully separate
  - example: Projected Map Display Set (PMDS)
- solution: module within module

Virtual Device Characteristics that are Likely to Change

- they cannot be hidden:
  user programs *must* behave differently if these characteristics change
  - examples:
    - ▷ measurement resolutions
    - ▷ number of positions on switches
    - ▷ max. displayable value
- a solution: symbolic constants
  - are system generation parameters

- problem:
  initial assumption wrong that
  all values known at system generation time
- solutions:

cost for variability	likelihood of change	solution
low	*	run-time variable (+ access prgs.)
high	low	system generation parameter
high	high	run-time variable with option to bind earlier
		conservative value for all devices, bind early

Device Dependent Data to/from Other Modules

- device dependent characteristics that vary at run-time
  - example: enter drift rate of IMS at run-time through panel
- reporting and displaying device dependent errors
- solution: restricted interface
  - mark these assumptions and and access programs as “restricted”
  - append to normal interface

Removable Interconnections Between Devices

- device interdependences for hardware convenience
  - example: Doppler and Ship Inertial Navigation Set share a data path
    - ▷ someone assumed the software never needs both simultaneously
  - can hide nature but not existence of connection
- hardware connection might be removed later
- similar: concurrent access to capabilities restricted within a single module
- solution: upward compatible interface
  - show interdependence now
  - maybe remove later

Interconnections Through Possible Failures?

- device A provides information, device B uses it
- device A can fail, invalidating the data of B
- if computer can detect failure of A:
  - device interface module of B can and should hide interconnection by simulating the detection of a failure of B
- if computer cannot detect failure of A:
  - users of B must expect undetectable failures
  - the interconnection itself can and should be hidden

Reporting Changes in Device State

- by signalling events or by access programs?
  - problem: depends on the (changing) requirements of user programs
- solution:
  specify always both,
  implement only what is used

Devices That Need Software Supplied Information

- information from outside device interface module
  - example: current IMS device needs to know whether aircraft is above $70°$ latitude
    - ▷ latitude not calculated within IMS module
- how to get information?
  - (a) device interface module provides access program
  - (b) device interface module programs call other programs
- solution: depends on whether information requirement is common to the replacement devices
  - if yes: provide access program

Virtual Devices that Do not Correspond to Hardware Devices

- a 1-to-1 relationship not always gives clear interfaces
  - some related capabilities scattered among several hardware devices
    - ▷ example: weapons-related capabilities of A-7
  - some unrelated capabilities occur in the same device for physical convenience
    - ▷ example: weapons release device fills two roles
  - some groupings explained by history only
- solution:
  - one virtual device for weapons release
  - one virtual device for weapon data

Bottom Line

- the basic definition of abstraction gives good guidelines even in hard design problems
- we can do a better job with a systematic procedure and a principle

When Won't It Work?

success depends on:

- the oracle assumption
  - our ability to predict change
- existence of commonality between actual interfaces
  - interface programs smaller than applications programs
- the Big "Big-Box" Assumption
  - the application is big enough to justify the effort for an abstract interface

Abstract Interface Design as an Application of Fundamental Principles

- being explicit about assumptions and design decisions
- encapsulation of likely change

- abstract interface module can solve
  the embedded computer system problem
  by hiding the embedding from the computer
- external interface modules are just a special case
  - use same method for other information hiding modules

4. Families of Systems

Overview of SCS4, Again

1. *rigorous description* of requirements
2. *what information* should be provided
   in computer system documentation?
3. *decomposition* into modules
4. *families* of systems

Overview of Chapter 4: Families of Systems

4.1 motivation:
maintenance problems in telephone switching

4.2 families of programs

4.3 families of requirements

4.1 Motivation: Maintenance Problems in Telephone Switching

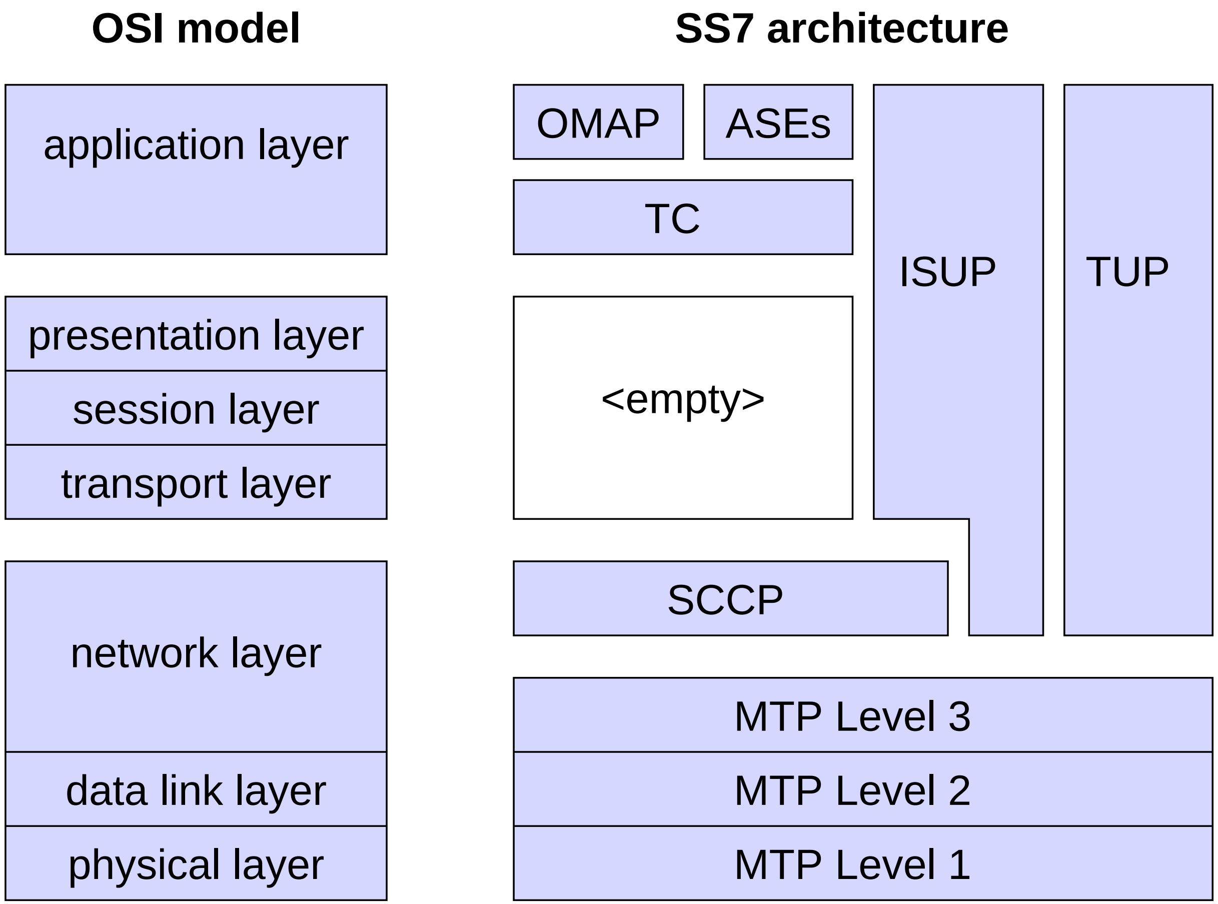
Overview of Chapter 4.1

- background
  on telephone switching
- feature interaction problems
  in telephone switching

History of Telephone Switching Systems

...	...
1950s	direct distance dialling (DDD) No. 5 Crossbar
early 1960s	stored program control switches
1976	Signalling System No. 6
1980	*Signalling System No. 7*
1984	*ISDN*
currently	IP telephony

Signalling System No. 7



OMAP:	Operations, Maintenance and Administration Part
ASE:	Application Service Element
TC:	Transactions Capabilities
ISUP:	ISDN User Part
SCCP:	Signalling Connection Control Part
MTP:	Message Transfer Part
TUP:	Telephone User Part

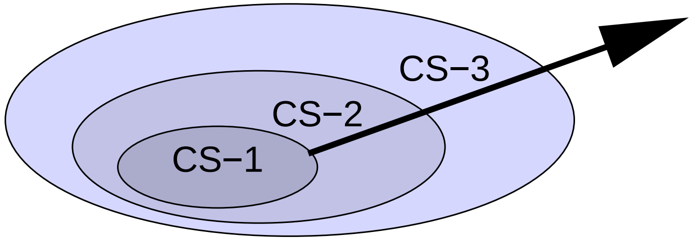
ISDN/DSS1

- Integrated Services Digital Network
- basic service:
  - two B-channels (64 kbit/s, transparent)
  - one D-channel (16 kbit/s, for signalling, e.g., call setup)
    - ▷ protocol: Digital Subscriber Signalling 1 (DSS1)
- supplementary services:
  - Calling Line Identification Presentation
  - Call Forwarding
  - Closed User Group
  - User-to-User Signalling
  - ...

- fixed set of supplementary services

Intelligent Network (IN)

- extension of telephone switching systems
- general goals:
  - rapid introduction of new services
  - broaden range of services
  - multi-vendor environment
  - evolve from (all) existing networks
- standardized by ITU-T
- approach: base service & additional services/features
- new services step by step:

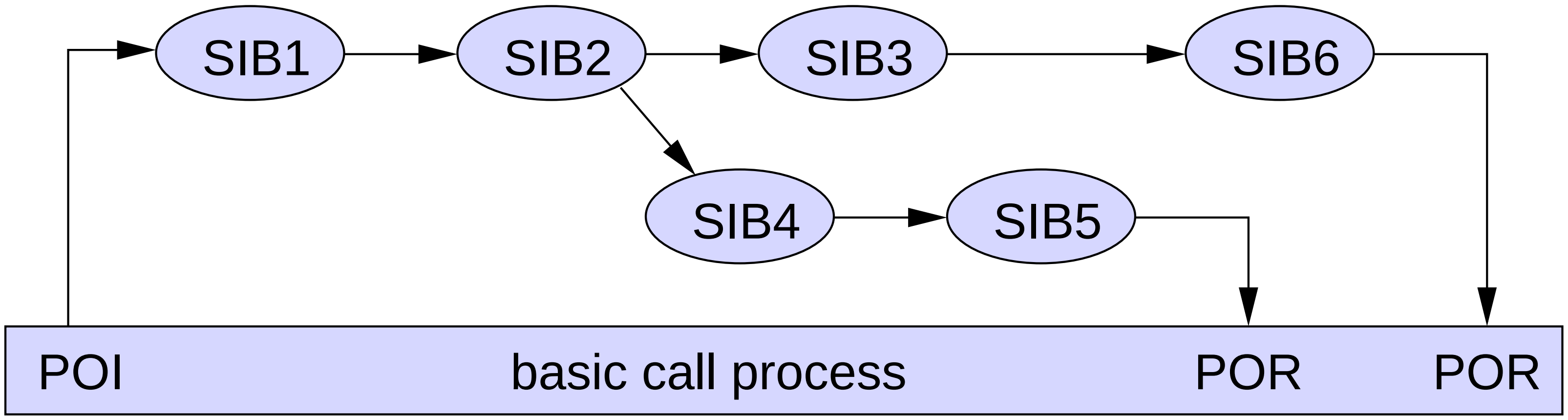


Intelligent Network Conceptual Model (INCM)

- four "levels":
  - service plane
  - global functional plane
  - distributed functional plane
  - physical plane

Global Functional Plane

- service independent building blocks (SIBs)
- service logic ("glue" for SIBs)
- basic call process
  - is special SIB
  - POI: point of initiation (of service)
  - POR: point of return



Services in IN CS-1

- Abbreviated dialling
- Account card calling
- Automatic alternative billing
- Call distribution
- Call forwarding
- Call rerouting distribution
- Completion of call to busy subscriber
- Conference calling
- Credit card calling
- Destination call routing
- Follow-me diversion
- Freephone
- Malicious call identification
- Mass calling
- Originating call screening
- Premium rate
- Security screening
- Selective call forward on busy / don't answer
- Split charging
- Televoting
- Terminating call screening
- Universal access number
- Universal personal telecommunications
- User-defined routing
- Virtual private network

- 25 services
- kind of services limited:
  - mainly for call setup and call tear down
  - 1 customer and 1 call leg only, mostly
- set is "political":
  - some services very similar
    - ▷ taken from different sources, without proper merge
    - ▷ example: Televoting / Mass Calling

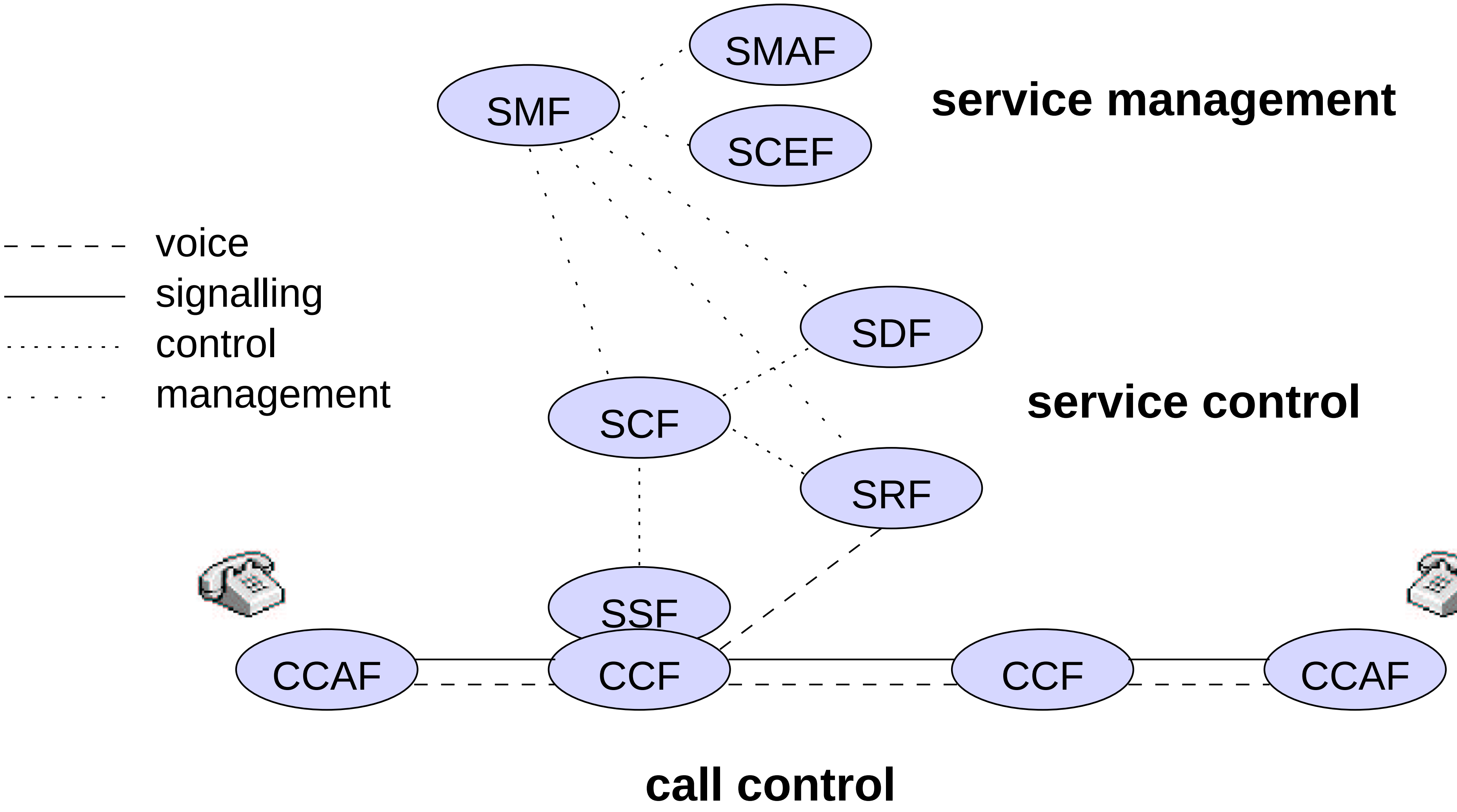
Features in IN CS-1

- Abbreviated dialling
- Attendant
- Authentication
- Authorization code
- Automatic call back
- Call distribution
- Call forwarding
- Call forwarding on BY/DA
- Call gapping
- Call hold with announcement
- Call limiter
- Call logging
- Call queueing
- Call transfer
- Call waiting
- Closed user group
- Consulation calling
- Customer profile management
- Customized recorded announcement
- Customized ringing
- Destinating user prompter
- Follow-me diversion
- Mass calling
- Meet-me conference
- Multi-way calling
- Off net access
- Off net calling
- One number

- Origin dependent routing
- Originating call screening
- Originating user prompter
- Personal numbering
- Premium charging
- Private numbering plan
- Reverse charging
- Split charging
- Terminating call screening
- Time dependent routing

- 38 features

Architecture of Distributed Functional Plane



Basic Call State Model

- originating BCSM *automaton*
- terminating BCSM *automaton*

Feature Interaction Problems in Telephone Switching

- *features work separately, but not together*
  - hundreds of (proprietary) features
  - combinations cannot be checked anymore
- telephone switching
  - users' expectation high
- feature
  - about any increment of functionality

Calling Card & Voice Mail

- #-button
  - (Bell) calling card:
    start new call without re-authorization
  - (Meridian) voice mail:
    end of mailbox number, end of password, . . .
- call voice mailbox using calling card??
  - either early disconnect, or
  - calling card feature crippled
- resolution by Bell
  - introduce new signal:
    "#-button pressed at least 2 sec."

Call Waiting & Call Forward on Busy

- both activated simultaneously
  - in busy state
  - when another call arrives
- only one can get control
  - no resolution, except restrictions on features

Originating Call Screening & Area Number Calling

- OCS
  - aborts calls to numbers in list
  - query Service Data Point (SDP) for list
- ANC
  - dialled number + area(calling number) $\rightarrow$ called number
  - example: Domino's Pizza
  - query SDP for called number

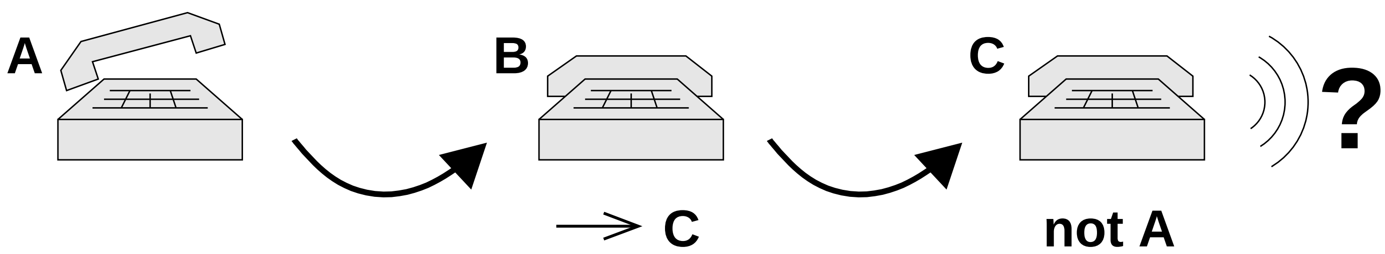
- switch may restrict no. of queries
  - protection against infinite loops
  - e.g., one query per call
  - $\longrightarrow$ OCS subscription prevents orders for pizza
- solution: one more query??

Calling Number Delivery & Unlisted Number

- conflict of goals
  - CND reveals caller
  - UN prevents revealing caller
- resolution
  - weaken one feature
  - e.g.: CND delivers only 1-111-1111-1111 for unlisted number

Call Forwarding & Terminating Call Screening

- CF
  - B forwards all calls to C
- TCS
  - when A is caller, C blocks him
- A calls B: can/should A reach C?



- notion of "caller" is crucial

Informal Feature Interaction Definition in Literature

- *FI:*
  the behaviour of a feature is changed by another feature
- not precisely clear what a feature actually is
- not all interactions are undesired

Categorization of Causes

according to [Cameron et. al.]:

- violation of feature assumptions
  - naming
  - data availability
  - administrative domain
  - call control
  - signalling protocol
- limitations on network support
  - limited CPE signalling capabilities
  - limited functionalities for communications among network components

- intrinsic problems in distributed systems
  - resource contention
  - personalized instantiation
  - timing and race conditions
  - distributed support of features
  - non-atomic operations

Approaches for Tackling FI

- ignore
- informal
  - filtering
  - heuristics
  - . . .
- formal methods
  - validation of:
    - ▷ specified properties of the features
    - ▷ general properties of the system (free of non-determinism, . . .)

- new architectures
  - IN
  - Tina, Race, Acts
  - DFC, agents
- better software engineering processes

- in practice: ignore / informal / processes / (architectures)
- formal analysis?
  yes, but. . .
  - formalization is huge task
  - complexity not amenable to tools
    - ▷ "spaghetti code" dependences

Feature Interactions in the Requirements

- if requirements complete,
  all FI are (inherently) present in the requirements

Requirements Structuring Problems

- monolithic requirements or single layer of extension
  - ISDN: monolithic
  - IN: no features on top of features
  - CF & TCS: resolution needs extended, common notion of caller
  - CF & OCS: resolution needs extended, common notion of called user

- new services depend implicitly on new concepts
  - some new concepts:
    - ▷ conditional call setup blocking
    - ▷ dialled number translation
    - ▷ multi-party call/session
      - required for CF & TCS and for CF & OCS
    - ▷ service session without communication session
    - ▷ distinction user – terminal device
    - ▷ distinction user – subscriber
    - ▷ mobility of users and of terminals
      - difficult to specify with network of distributed switches
    - ▷ multiple service providers, billing separately

- concerns of the users' interface are spread out
  - several features assume exclusive access to the user's terminal device (12 buttons + hook)
  - example: calling card & voice mail

Needed: a More Modular Requirements Structure

- centralize responsibility for the users' interface
- a layered architecture
  - like in computer communication systems

New Architectures

- current: IN
  - currently largest impact on implementations
    - ▷ see above
  - Jain
    - ▷ enhanced IN-like architecture
    - ▷ developed currently
    - ▷ in Java
    - ▷ allows multi-party, multi-media calls
    - ▷ Java Call Control (JCC):
      call state machine similar to that of the IN
    - ▷ JCC does not handle feature interactions

- future: Tina, Race, and Acts
  - Tina
    - ▷ radical approach: entirely new architecture
    - ▷ strongly based on Open Distributed Processing (ODP) and Corba
    - ▷ migration difficult
  - Race project
    - ▷ Cassiopeia
      - developed open services architectural framework (Osa)
      - many commonalities with Tina
      - focuses on requirements engineering of services
      - tries to take legacy services into account
    - ▷ Score
      - concerned with the methodological aspects of service creation
      - detection of undesired service interactions:
        formal methods, exhaustive simulation
        applied to small example

- Acts project
  - ▷ followed Race project
  - ▷ application and on evaluation of service architectures
  - ▷ result: a modified architecture

- research: the DFC and the agent architecture
  - Distributed Feature Composition (DFC)
    - ▷ compose features in a pipe-and-filter network
    - ▷ designed to be implementable on a conventional switch
    - ▷ some new concepts supported, others not
    - ▷ no layered architecture
    - ▷ implemented in AT&T's Eclipse project, which additionally incorporates Voice Over IP
  - Zibman et. al.'s agent architecture
    - ▷ separates several concerns explicitly
    - ▷ restricts itself to narrow-band telephony over a fixed network
    - ▷ Plain Old Telephone Service is represented by a single service agent

Discussion of New Architectures

- IN important step, but not sufficient
- Tina, Race, Acts have most of the interesting concepts, but transition is very expensive
- feature interaction detection is still research

- some undesired service interactions still possible in new architectures
  - a paper checked the FI benchmark for Tina
  - still possible:
    - ▷ forwarding loop
    - ▷ automatic callback & automatic re-call
    - ▷ calling number delivery & calling number delivery blocking
    - ▷ billing problems for video conference
    - ▷ . . .
  - causes: violated assumptions or conflicting goals
- *how to prepare for unanticipated changes??*
  - at least encapsulate as much as possible

4.2 Families of Programs

Overview of Chapter 4.2

- basic idea of families of programs
- . . . and what to do if the first version is due yesterday

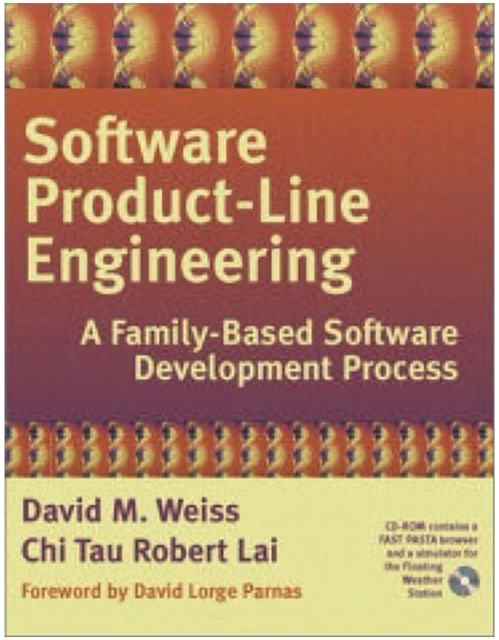
Text for Chapter 4.2

[Par76] Parnas, D. L. *On the design and development of program families*. IEEE Trans. Softw. Eng. **2**(1), 1–9 (Mar. 1976).

First paper to introduce families of programs explicitly. Presents the essentials very clearly.

[WeLa99] Weiss, D. M. and Lai, C. T. R. *Software Product Line Engineering – a Family-Based Software Development Process*. Addison Wesley Longman (1999).

Best current book on how to do software product line engineering (families of programs) in practice.

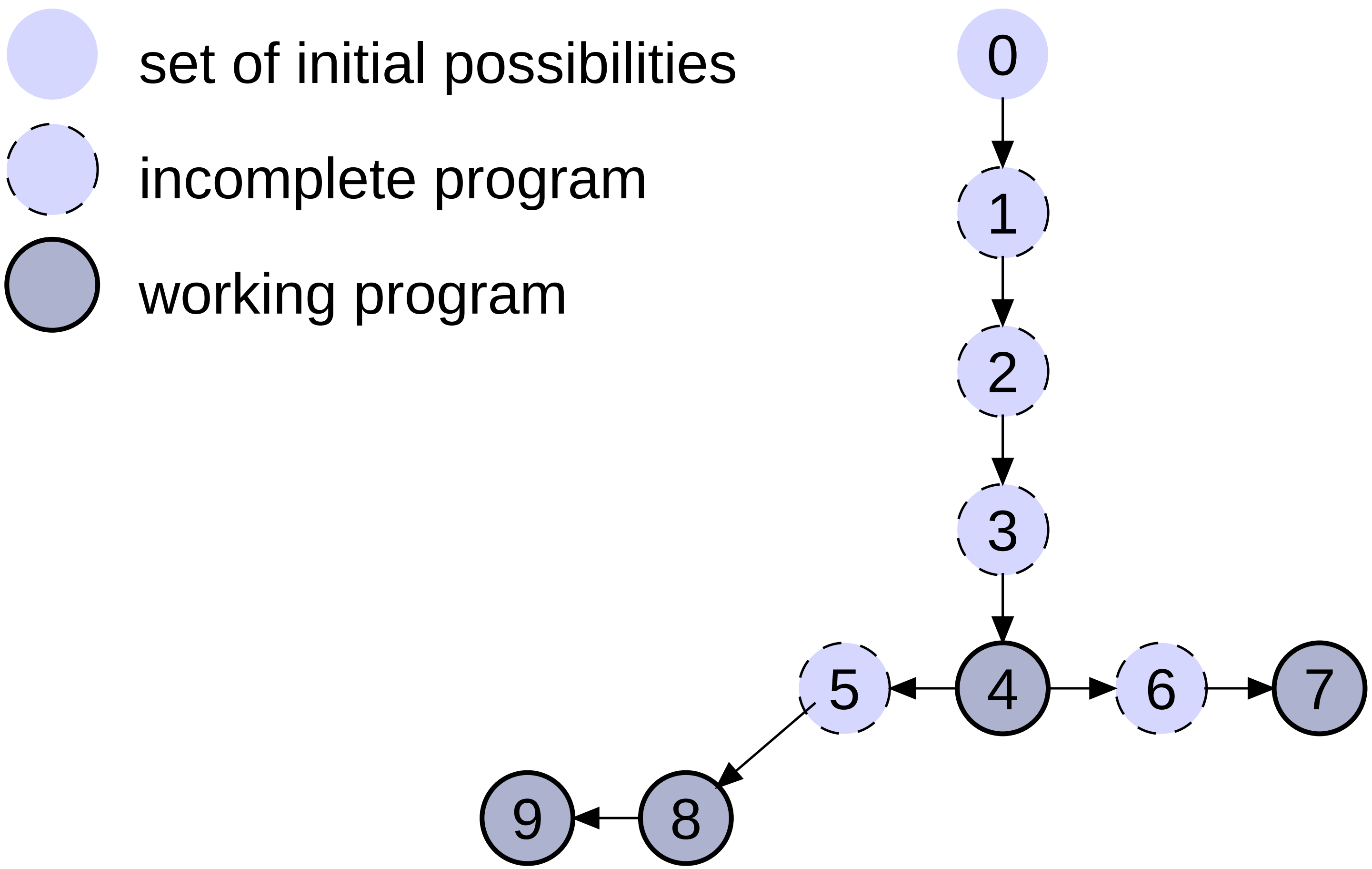


Definition of Program Family

Definition 20 (Program family)
A set of programs constitutes a family whenever it is worthwile *to study programs from the set by* first *studying the common properties of the set and* then *determining the special properties of the individual family members.*

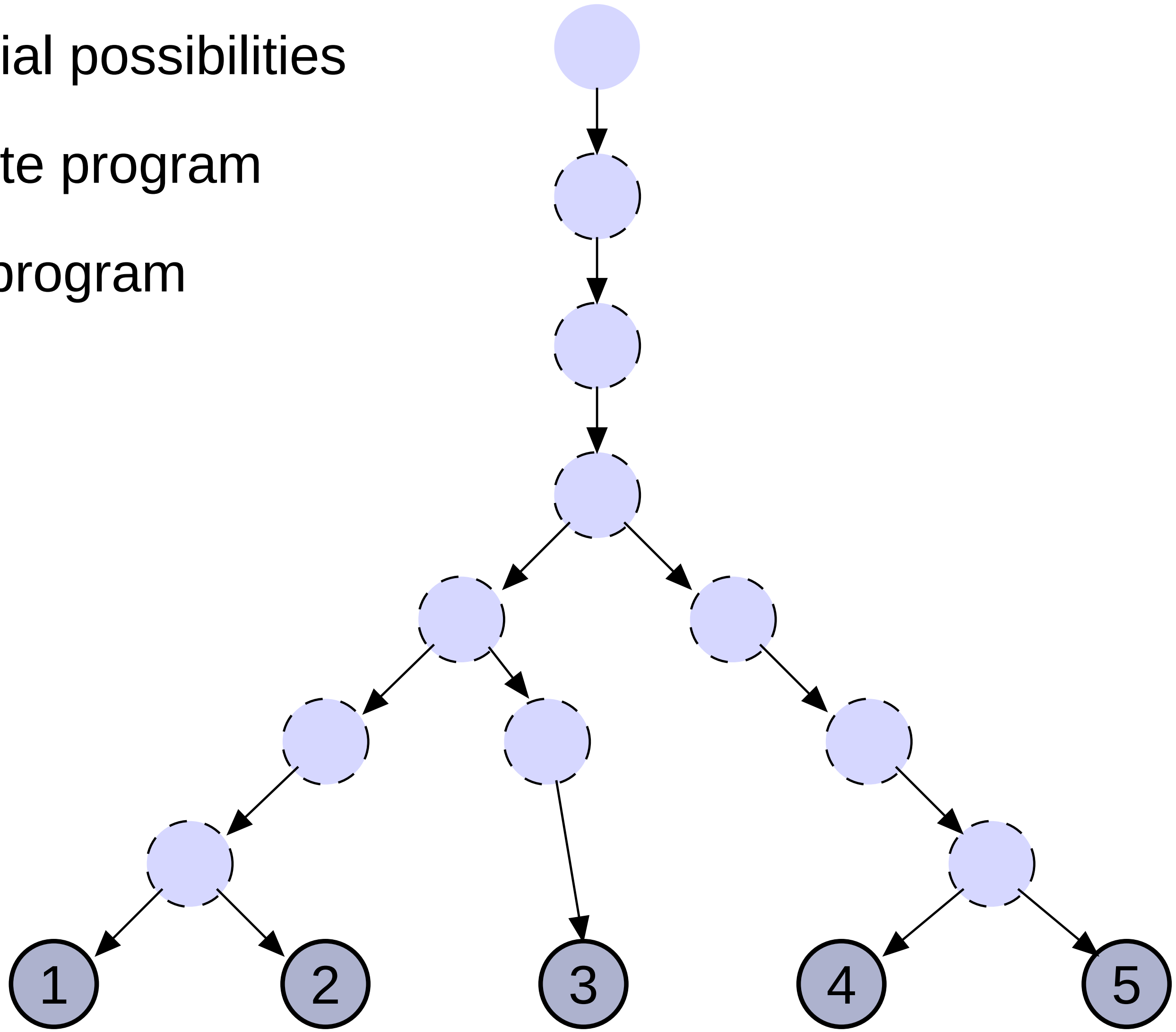
- examples:
  - the set of versions of an embedded software for different environments
  - the set of versions of a software over time

The “Classical” Method of Producing Program Families



Newer Techniques

- set of initial possibilities
- incomplete program
- working program



Stepwise Refinement

- intermediate stages:
  - complete programs
  - except: certain operators and operand types only specified, not yet implemented
- next step: provide some more implementation, using more, newly introduced specifications as necessary
- linear sequence of steps towards one program
  - if a step must be taken back, all subsequent steps are lost

Module Specification

- intermediate stages:
  - black-box specifications of modules
  - *not* complete programs
- next step: add design decisions for a module, using newly introduced sub-modules as necesary
- steps taken in different modules are independent
  - any step taken back affects its sub-modules only
  - order of steps: more important
  - independent further development of modules

Discussion of Both Development Approaches

- both based on same basic ideas:
  - represent intermediate stages precisely
  - postpone certain decisions
- extra effort to design first family member:
  - stepwise refinement: none
  - module specification: significant
- effort to design next family members:
  - stepwise refinement: high, if early step taken back
  - module specification: low, as long as low uses-level modules affected

Dilemma: Careful Engineering vs. Rapid Production

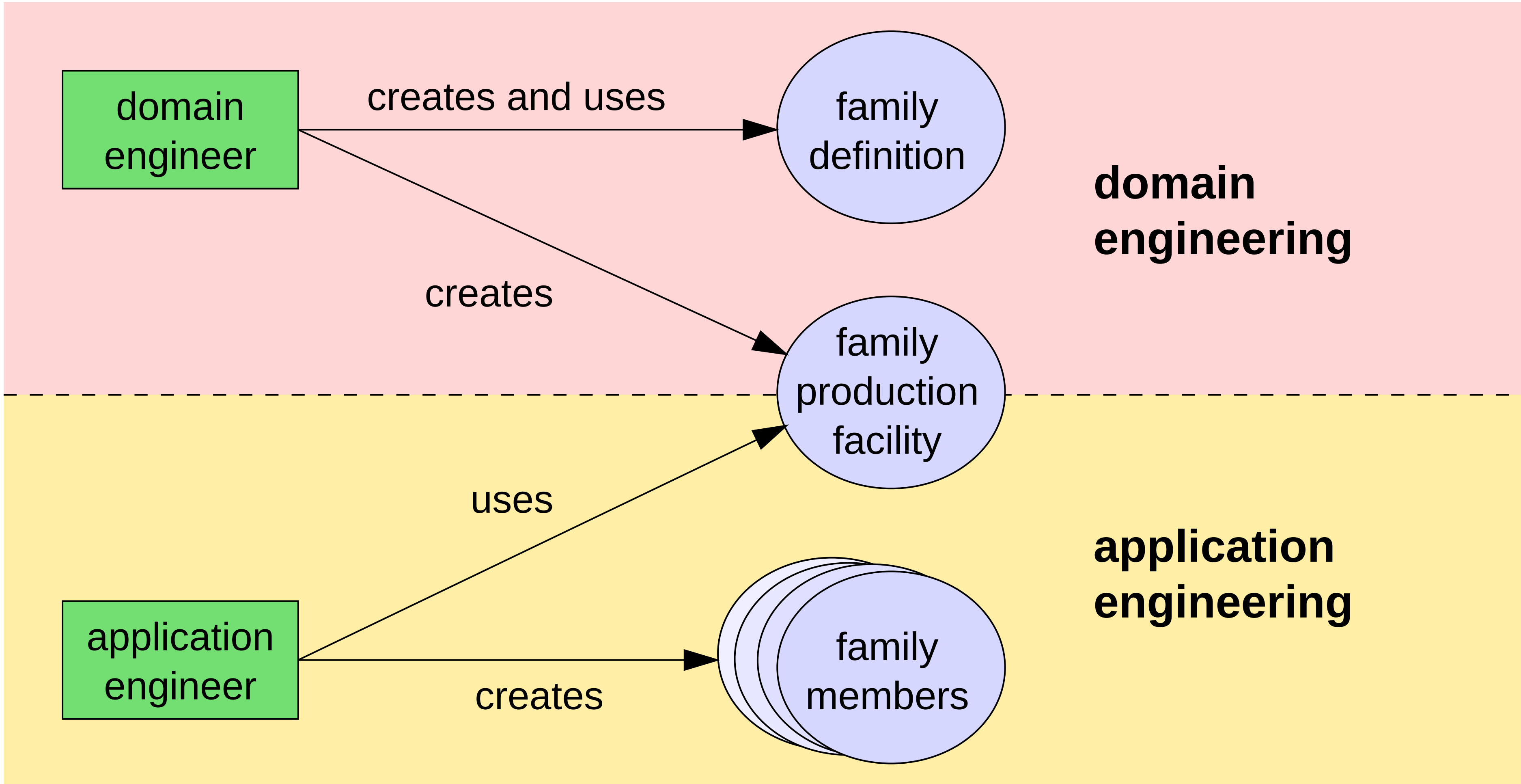
- careful engineering:
  - attractive functionality
  - ease of use
  - reliability
  - easy to enhance
- rapid production:
  - market it ahead of competition

A Solution in Other Fields

- fields:
  - aerospace
  - automotive
  - computer hardware
  - . . .
- idea: *a family of products produced with a single production facility*

- family: set of items
  - common aspects (e.g., chassis)
  - predicted variabilities (e.g., engine)
- def. *product line*: a family of products

Family-Oriented Abstraction, Specification, and Translation (FAST)



Applications of FAST

- developed and in use within Lucent Technologies (development: Bell Labs)
- many product lines already created there

Basic Assumptions

- redevelopment hypothesis
- oracle hypothesis
- organizational hypothesis

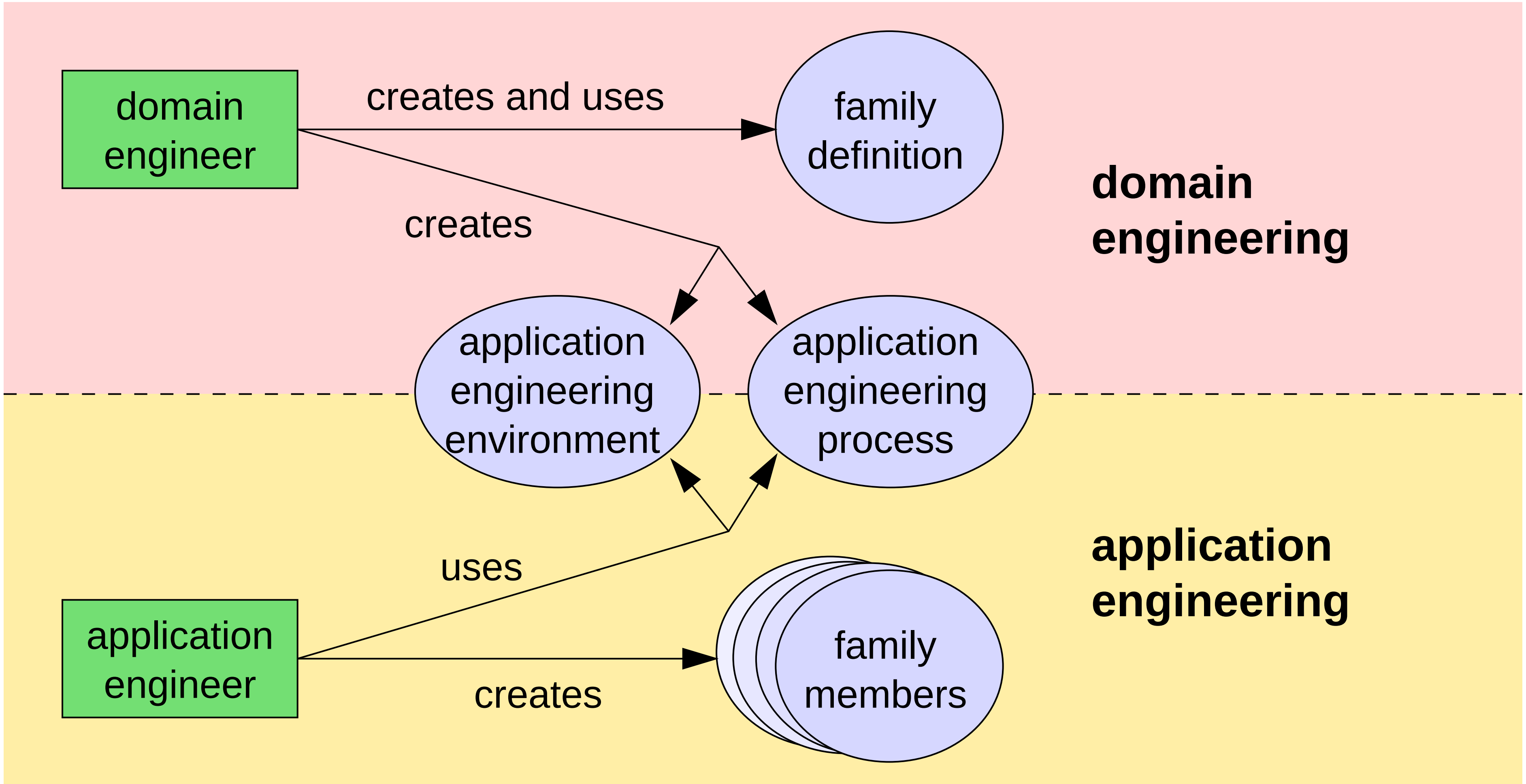
Stages Towards an Engineered Family

1. potential family
   - one suspects sufficient commonality
2. semifamily
   - common and variable aspects identified
3. defined family
   - semifamily + economic analysis of exploiting it
4. engineered family
   - defined family + investment in processes, tools, resources

FAST Strategies

- identify collections of programs
  that can be considered families
- design the family for producibility
- invest in family-specific tools
- create a family-specific way to model family members
  - for validating the requirements by exploring the behaviour
  - for generating code and documentation

Outputs from Domain and Application Engineering



Predicting Change

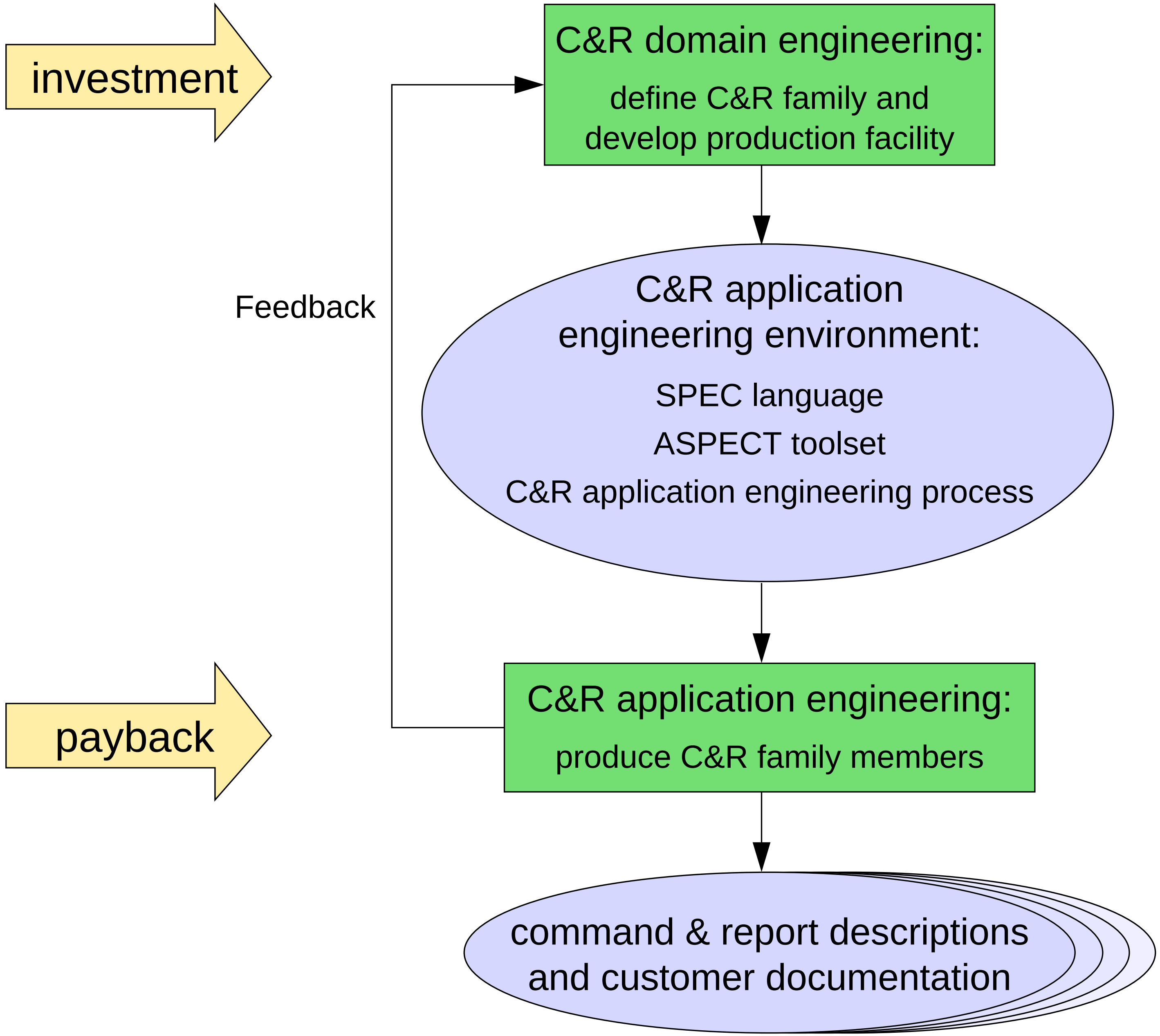
- is critical
  - but is not all-or-nothing
- confidence should rule size of investment
- FAST: explicitly bounds change (oracle hypothesis)
  - allows for common abstractions
- good guides for future change:
  - past change
  - your marketing organization
  - early adopters
  - experienced developers

Organizational Considerations

- reorientation of software development around domains may need change in organization of development
  - e.g., one sub-organization for each domain
  - e.g., a product line composed out ouf several sub-domains
    - ▷ example: protocol stack

Example: FAST Applied to Commands and Reports

- C&R: part of Lucent's 5ESS telephone switch
- technicians monitor and maintain running switch
  - issue commands
  - receive status reports
- voluminous documentation
- command set: thousands of commands and report types


investment
payback
C&R domain engineering:
define C&R family and
develop production facility
Feedback
C&R application
engineering environment:
SPEC language
ASPECT toolset
C&R application engineering process
C&R application engineering:
produce C&R family members
command & report descriptions
and customer documentation

Defining the C&R Family

- identify potential family members, characterize commonalities and differences
- 5ESS command:
  - always command code followed by parameters
    - ▷ command code: action and an object
      - example: report status of a line connected to the switch
  - the particular actions, objects, parameters vary
    - ▷ over reasonably well-defined sets
    - ▷ certain combinations not included in family
      - example: remove clock is not included, but set clock is included

C&R Commonality Analysis Document

- introduction
  - purpose of the commonality analysis
- overview
  - brief overview of C&R domain
- dictionary
  - defines technical terms for the C&R domain used
- commonalities
  - assumptions true for every member
- variabilities
  - assumptions about how members may vary

- parameters of variation
  - the value space for each variability
  - the time at which the value must be fixed
- issues
  - issues that arose during analysis / how resolved

Excerpts from Dictionary Section

Command code	Unique identifier of an *input command*, consisting of a *verb* and an *object*.
Input command	A command entered by an office technician that acts as a stimulus to the 5ESS to perform tasks. Such tasks include changing the state or reporting the state of the 5ESS.
Input command definition	A specification of all the information needed to identify and produce an *input command* or a set of *input commands* with common structure and contents.
Input command manual page	Documentation of an *input command* for the customer's use.
Output report	An information message that is printed on an output device.
Output report definition	A specification of all the information needed to identify and produce an *output report* or a set of *output reports* with common structure and contents.

Purpose	*Customer documentation* that describes the use of an *input command*.
Verb	The name of the action indicated by an *input command*.

Excerpts from Commonality Section

COMMONALITIES

The following are basic assumptions about the domain of *input commands*, *output reports*, and *customer documentation*.

INPUT COMMANDS

C1. Each *input command* is uniquely determined by its *command code*. When an *input command definition* is used to define more than one *input command*, it defines multiple *command codes*, all of which share the same set of *input parameters*.

C2. Each *input command* is described on exactly one *input manual page*.

C3. The following *administrative data* are required in an *input command definition*: *msgid, process, ostype, schedule*, and *auth*. Each *input command* has exactly one value for each of these fields.

C4. A *verb* is an *alpha-string* with a maximum length.

C5. There is a fixed maximum number of *input parameters* permitted for *input commands*.

C6. An *input parameter description* consists of a *parameter name* and a *value specification*. The *value specification* defines the range of values that an office technician may use for the *input parameter*.

OUTPUT REPORTS

C7. *Output reports* appear in three different contexts as follows.

a. Runtime: At runtime an *output report* may appear on an output device, such as the printer.

b. Report definition: The set of *output reports* that a 5ESS switch may produce at runtime, and the meaning of each possible *output report*, must be defined before building the software for the switch.

c. Output report documentation: Each *output report* must be documented for customer use. The documentation of *output reports* must include all the information that the office technician needs to know to understand the report and determine the reason for its appearance at runtime.

C8. An *output report* contains the report type – spontaneous or solicited – and the text of the report.

C9. There is a fixed maximum number of characters in a line of an *output report*.

C10. Each *output report* is described on exactly one *output manual page*; however, an *output manual page* may describe more than one *output report*.

C11. An *output report definition* is a sequence of *text block definitions*.

DOCUMENTATION

C12. An (*input command* or *output report*) *manual page* consists of several fixed sections. It may also reference an *appendix*.

C13. An (*input command* or *output report*) *manual page* documents one or more *input commands* or *output reports*.

SHARED COMMONALITIES

C14. All the information needed to define an *input command*, the associated *solicited output report*, and the associated *manual pages* must be describable as one specification. It must be possible to generate from such a specification all the files and data needed to process *input commands* and produce *output reports* at runtime and to generate either (1) the *input command* and *output manual pages* or (2) files and data that can be used to generate the *input command manual pages* and *output manual pages*.

Excerpts from Variabilities Section

The following statements describe how *input commands*, *output reports*, and *customer documentation* may vary.

VARIABILITIES

INPUT COMMANDS

V1. The maximum length of a *verb*, *object*, *parameter name*, or *enumeration* value.

V2. The domain for *verbs*.

V3. The maximum number of *input parameters*.

V4. The *Csymbol* used to designate a *msgid*.

OUTPUT REPORTS

V5. The maximum number of characters in a line of an *output report*.

DOCUMENTATION

V6. The representation of an *input command* on an *input manual page*, particularly the following in the syntactic template for the *input command*:

a. The separators used between the *command code* and the list of *input parameters*

b. The terminator for the representation of the *input command*

c. The separator used between the *verb* and the *object*

Typical *input command* representations appear as follows:

<command code rep><separator1><input parameter rep><input terminator>

<command code rep><input terminator>

<verb><separator2><object>

V7. Typographic distinguishers for command templates.

Sample Command Template, Written in SPEC

```
COMMAND {
  TEMPLATE {
    abt-task:tlws;
    purpose: "Aborts an active trunk and line workstation
             (TLWS) maintenance task.";
    warning: "Once this command is entered, the
             consistency of all hardware states and data
             in use by the task is questionable.";
    }
}
```

Formatted Generated Documentation

ABT-TASK:TLWS=a;

Warning: **Once this command is entered, the consistency of all hardware states and data in use by the task is questionable.**

- Purpose

  Aborts an active trunk and line workstation (TLWS) maintenance task.

- Explanation of Parameters

  a = Task identifier given to active TLWS maintenance tasks by the OP-JOBST command.

- Responses

  Only standard system responses apply.

Sample Parameter Definition

```
COMMAND {
  ..........

  PARAM tlws {
     TYPE {
          domain: num;
          min: 0;
          max: 15;
          default: 0;
          }
     desc: "Task identifier given to active TLWS
            maintenance tasks by the OP-JOBST command.";
     csymbol: task_id;
     }
}
```

A Parameterized Version of TLWS

```
PARAM lib_tlws( x ) {
   TYPE {
        domain: num;
        min: 0;
        max: 15;
        default: 0;
        }
   desc: "Task identifier given to active TLWS
          maintenance tasks by the OP-JOBST command.";
   csymbol: x;
   }
```

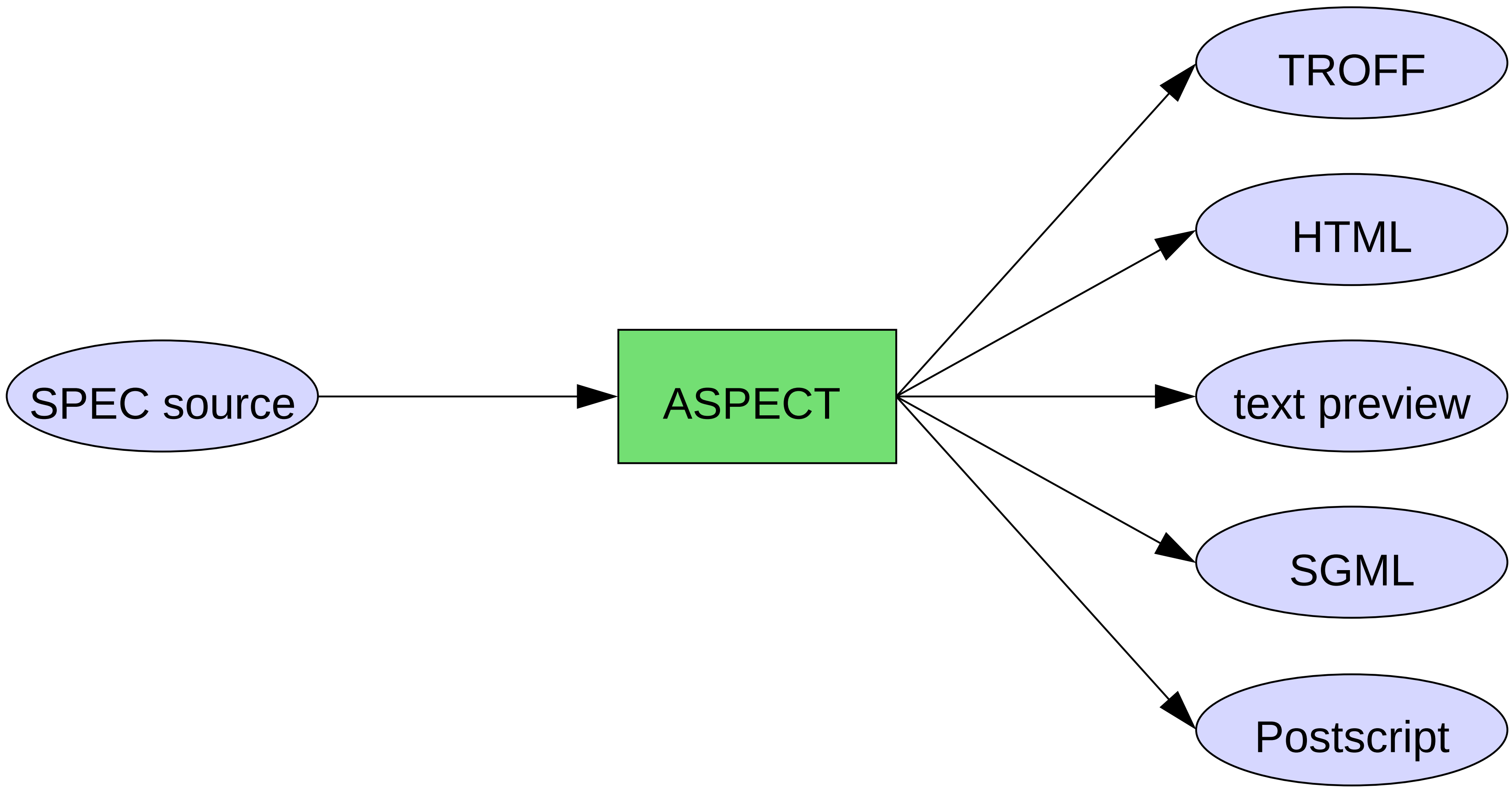
Reuse of TLWS

```
COMMAND {
  TEMPLATE {
    abt-task:tlws;
    purpose: "Aborts an active trunk and line workstation
             (TLWS) maintenance task.";
    warning: "Once this command is entered, the
             consistency of all hardware states and data
             in use by the task is questionable.";
    }

  PARAM tlws use lib_tlws( task_id )

}
```

Producing Multiple Documentation Formats



Designing the Translators

- existing parser generator tools used
- principles of software family development applied
- combined with SCR design process
- minimal toolset:
  - command translator
  - report translator
  - command documentation generator
  - report documentation generator
- much overlap between translators expected (commonalities)

Using the SCR Design Process

- information hiding hierarchy
  - module guide
  - uses relation
- ASPECT:
  - external interface module
    - ▷ . . .
  - behaviour hiding module
    - ▷ . . .
  - software decisions module
    - ▷ . . .
- result: substantial code reuse

ASPECT External Interface Module

- output drivers module
  - command format module
  - report format module
  - documentation format module
- library reference module
- device drivers module
  - text module
  - HTML module
  - formatter macros (TROFF) module
  - Postscript module
  - SGML module

ASPECT Behaviour Hiding Module

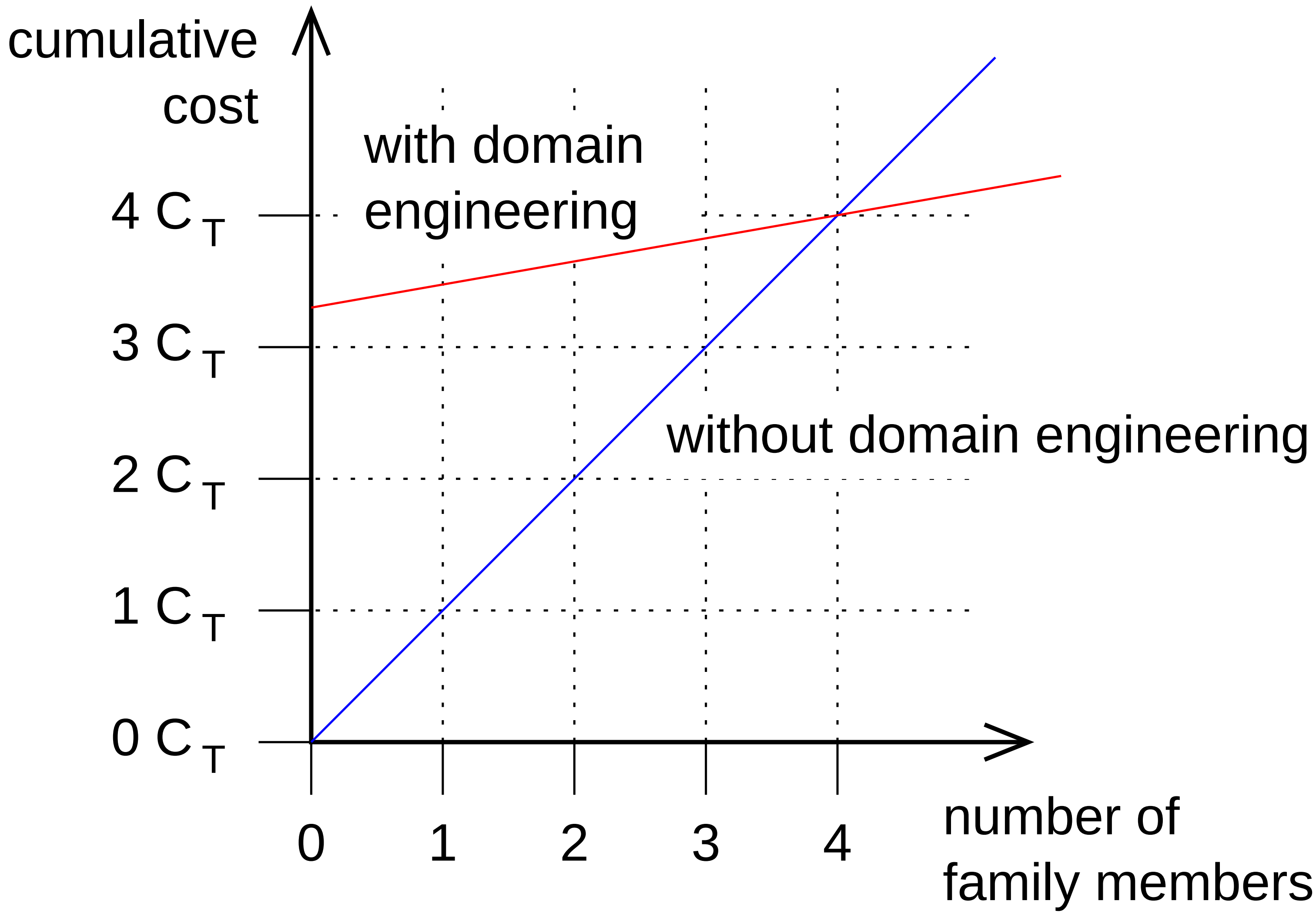
- tool builder module
- input command traversal module
- output report traversal module
- command documentation traversal module
- report documentation traversal module
- shared services module

ASPECT Software Decisions Module

- cross reference module
- database module
- domain translator module
- error recorder module
- global context module
- preprocessors module
  - alter structure module
  - alter syntax module
  - random access module

- semantic verification module
  - completeness module
  - consistency module
  - placement module
- specification expander module
- symbol reference module
- text function module
- text translation module
- global language data module
- system interface module
- transversal module

The Economics of FAST



Modelling the FAST Process

- there is a precise model for the FAST process
  - see [WeLa99]
- description of process models: PASTA approach (Process and Artifact State Transition Abstraction)
  - see [WeLa99]

Finding Domains where FAST is Worth Applying

- usually apply to legacy systems
- look for domain with
  - frequent, continuing change
  - change at high cost
  - predictable change
  - (quick change needed)
- do an informal or formal economic analysis

Applying FAST Incrementally

- early activities of FAST:
  better understanding of market, customers, requirements
  - facilitates communication, staff training, member design
  - modest cost
- later activities of FAST:
  make effective use of information and understanding
- apply FAST iteratively, e.g.:
  1. commonality analysis only, to make design more flexible
  2. introduce a rudimentary language
     to generate data structures changing most often
  3. expand language to generate majority of code

Transitioning to a FAST Process

- FAST process allows for gradual introduction into company
- early: staff learns to think in terms of families
  - test: can they predict future changes?
- later: use this thinking
- one way to start:
  - pick a few, high-leverage, well-understood domains
  - apply a simple version of FAST
  - several iterations
  - if you understand it and if it works,
    spread to more domains and more parts of company
- you might have to reengineer your organization later

4.3 Families of Requirements

Text for Chapter 4.3

[Bre01b] Bredereke, J. *A tool for generating specifications from a family of formal requirements*. In Kim, M., Chin, B., Kang, S., and Lee, D., editors, "Formal Techniques for Networked and Distributed Systems", pp. 319–334. Kluwer Academic Publishers (Aug. 2001).

A tool for families of CSP-OZ specifications.

Additional Background for Chapter 4.3

[Bre02] Bredereke, J. *Maintaining telephone switching software requirements*. IEEE Commun. Mag. **40**(11), 104–109 (Nov. 2002).

Telephone switching system structure problems and solutions.

[Zav01] Zave, P. *Requirements for evolving systems: A telecommunications perspective*. In "5th IEEE Int'l Symposium on Requirements Engineering", pp. 2–9. IEEE Computer Society Press (2001).

Feature-oriented descriptions and "feature engineering" in telephone switching.

[Mil98] Miller, S. P. *Specifying the mode logic of a flight guidance system in CoRE and SCR*. In “Second Workshop on Formal Methods in Software Practice”, Clearwater Beach, Florida, USA (4–5 Mar. 1998).

Application of the CoRE approach to an auto-pilot.

[Bre00b] Bredereke, J. *genFamMem 2.0 Manual – a Specification Generator and Type Checker for Families of Formal Requirements*. University of Bremen (Oct. 2000). URL http://www.tzi.de/~brederek/genFamMem/.

Definition of CSP-OZ language extension and manual for the genFamMem tool.

[Bre00a] Bredereke, J. *Families of formal requirements in telephone switching*. In Calder, M. and Magill, E., editors, "Feature Interactions in Telecommunications and Software Systems VI", pp. 257–273, Amsterdam (May 2000). IOS Press.

Families of CSP-OZ specifications.

[Bre00d] Bredereke, J. *Specifying features in requirements using CSP-OZ*. In Gilmore, S. and Ryan, M., editors, "Proc. of Workshop on Language Constructs for Describing Features", pp. 87–88, Glasgow, Scotland (15–16 May 2000). ESPRIT Working Group 23531 – Feature Integration in Requirements Engineering.

Families of CSP-OZ specifications.

[Bre00c] Bredereke, J. *Hierarchische Familien formaler Anforderungen*. In Grabowski, J. and Heymer, S., editors, "Formale Beschreibungstechniken für verteilte Systeme – 10. GI/ITG-Fachgespräch", pp. 31–40, Lübeck, Germany (June 2000). Shaker Verlag, Aachen, Germany.

Families of CSP-OZ specifications, ordered hierarchically.

[Bre01a] Bredereke, J. *Ein Werkzeug zum Generieren von Spezifikationen aus einer Familie formaler Anforderungen*. In Fischer, S. and Jung, H. W., editors, “Formale Beschreibungstechniken – 11. GI/ITG-Fachgespräch”, Bruchsal, Germany (June 2001). URL http://www.i-u.de/fbt2001/.

A tool for families of CSP-OZ specifications.

[Bre99] Bredereke, J. *Modular, changeable requirements for telephone switching in CSP-OZ*. Tech. Rep. IBS-99-1, University of Oldenburg, Oldenburg, Germany (Oct. 1999).

Case study with families of CSP-OZ specifications.

[Bre98] Bredereke, J. *Requirements specification and design of a simplified telephone network by Functional Documentation*. CRL Report 367, McMaster University, Hamilton, Ontario, Canada (Dec. 1998).

Case study with families of Parnas tables.

[Kat93] Katz, S. *A superimposition control construct for distributed systems*. ACM Trans. Prog. Lang. Syst. **15**(2), 337–356 (Apr. 1993).

Seminal paper on superimposition.

[BrSc02] Bredereke, J. and Schlingloff, B.-H. *An automated, flexible testing environment for UMTS*. In Schieferdecker, I., König, H., and Wolisz, A., editors, “Testing of Communicating Systems XIV – Application to Internet Technologies and Services”, pp. 79–94. Kluwer Academic Publishers (Mar. 2002).

Families of CSP test specifications.

Overview of Chapter 4.3

- feature-oriented description
- the CoRE method
- families of CSP-OZ specifications
- families of CSP test specifications

Focus on Requirements

- motivation:
  - all feature interaction problems
    already (implicitly) present in requirements
  - many “formal methods” support single product only
    - ▷ how to integrate family support into method?

Feature-Oriented Description in Telephone Switching

- base description plus separate feature descriptions
- attraction: behavioural "modularity"
  - easy change of system behaviour
  - make *any* change by just adding a new feature description
  - never change existing descriptions
- emphasizes individual features
  - makes them explicit
- de-emphasizes feature interactions
  - makes them implicit in the feature composition operator

- not all feature interactions are bad
  - feature-oriented description relies on the good ones
- example: busy treatments
  - B_1 and B_2 both enabled, B_2 higher priority
  - B_1 not applied, despite its stand-alone description
  - behavioural "modularity":
    add new busy treatments without changing existing ones
- most feature-oriented descriptions still informal
  - behavioural "modularity" and formality do not combine easily
    - ▷ behavioural "modularity": don't answer some questions now
    - ▷ formality: answer all questions now
  - proposed composition operators / approaches often do not scale

- IP telephony:
  - highly complex new services
  - services still viewed as stand-alone
  - undesired feature interactions will haunt us soon

Feature-Oriented Descriptions and Common Abstractions

- modules need common abstractions/assumptions
  - module: now in the sense of this lecture
  - common abstraction/assumption: true for all family members
- rapid innovation, legacy systems, too many players: hard to limit the domain
- *without domain limits: no common abstractions*

Performing Incremental Specification Formally

- standard means:
  stepwise refinement
- step:
  1. extend behaviour *or* 2. impose constraints
  - example 1.: add another potential event to a state
  - example 2.: specify the order of two events
- interesting properties preserved by step
  - example 1.: all old events remain possible
    - ▷ no deadlock in this state
  - example 2.: no harmful event added
    - ▷ all safety properties preserved

Non-Monotonous Changes

- telephone switching:
  new features change the behaviour
  - of base system, or
  - of other features
- example: call forwarding
  - stops to connect to dialled number
    - ▷ restricts base system behaviour

    and
  - starts connecting to forwarded-to number
    - ▷ extends base system behaviour

Formal Support for Feature Specification

- considerable research effort
  on feature composition operators
- FIREworks project
  (Feature Interactions in Requirements Engineering)
  - various feature operators proposed and investigated
- "feature-oriented programming"
- based on the superimposition idea by Katz
- analytical complexity:
  too big for tools for real systems

Superimposition

- by Katz [Kat93]
- approach:
  - base system
  - textual increments
  - composition operator
- problem:
  - increments have defined interface,
    base system has not
  - increment can invalidate arbitrary assumptions about base system

The CoRE Method

- based on four-variable model and SCR
- groups the variables into classes
- developed during the early 1990's
- no explicit family support, but maybe a good base for it
- no formal syntax and semantics
- no tool support

Families of CSP-OZ Specifications

key ideas:

- *maintain all variants together*
  - generate specific member automatically as necessary
- *document information needed for changes*
  - dependence of requirements
  - what is the core of a feature

Constraint-Oriented Specification

- features closely interrelated
  - most refer to mode of connection
  - user interface: few, shared lexical events
    - ▷ system cannot be sliced by controlled events
- incrementally impose partial, self-contained constraints
- composition by logical conjunction

The Formalism CSP-OZ

- CSP-OZ demo: one very simple telephone *demo*
- CSP-OZ class inheritance for incremental constraints

Case Study on Telephone Switching Requirements

- black box specification of telephone switching
- attempt to incorporate new concepts
- details: see [Bre99]
  papers: see [Bre01b, Bre01a, Bre00c, Bre00a, Bre00a]

Grouping Classes into Features

the chapters of the requirements document:

1. Introduction
2. `feature` UserSpace
3. `feature` BasicConnection
4. `feature` VoiceChannel
5. `familymember` SpecificationA
6. `feature` ScreeningBase
7. `feature` BlackListOfDevices
8. `familymember` SpecificationB
9. `feature` BlackListOfUsers
10. `feature` FollowHumanConnectionForwarding
11. `familymember` SpecificationC
12. `feature` TransferUserRoleToAnotherHuman
13. `familymember` SpecificationD

⋮ ⋮

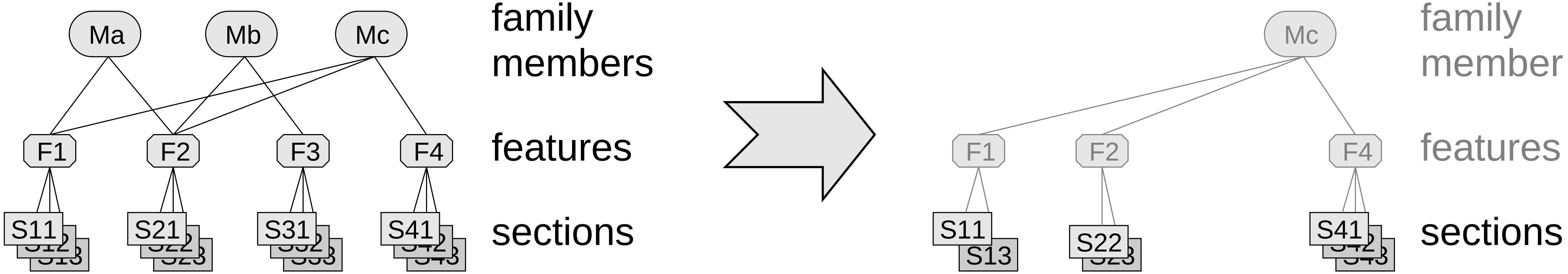
Indices / Bibliography

The Feature Construct

- feature UserSpace *spec*
- feature BasicConnection
- familymember SpecificationB

Generating Family Members From a Family Document

family of requirements **requirements specification**



extension of CSP−OZ

plain CSP−OZ

Result of Family Member Generation

1. Introduction
2. `feature` UserSpace
3. `feature` BasicConnection
4. `feature` VoiceChannel
5. `feature` ScreeningBase
6. `feature` BlackListOfDevices
7. `familymember` SpecificationB
Indices / Bibliography

- family member composition chapter:
  part replaced *spec*

Controlled Non-Monotonous Changes

- feature ScreeningBase *spec*
- feature BlackListOfUsers
- feature FollowHumanConnectionForwarding
- familymember SpecificationC

Avoiding Feature Interactions

- introduced three notions explicitly
  - “telephone device”
  - “human”
  - “user role”
- consequences:
  - black list above:
    screens user roles, not devices
  - another black list feature:
    screens devices, not user roles
  - also two kinds of call forwarding
- no feature interaction screening–forwarding anymore

Detecting Feature Interactions by Type Checks

- *type rules*: part of the family extension of CSP-OZ
- syntactic rules → *syntactic errors*:
  - “remove” an “essential” class
  - feature of needed class not included
  - feature of “removed” class not included
  - another class still needs “removed” class
- heuristic syntactic rules → *syntactic warnings*:
  - class is marked both essential and changeable
  - class is “removed” twice

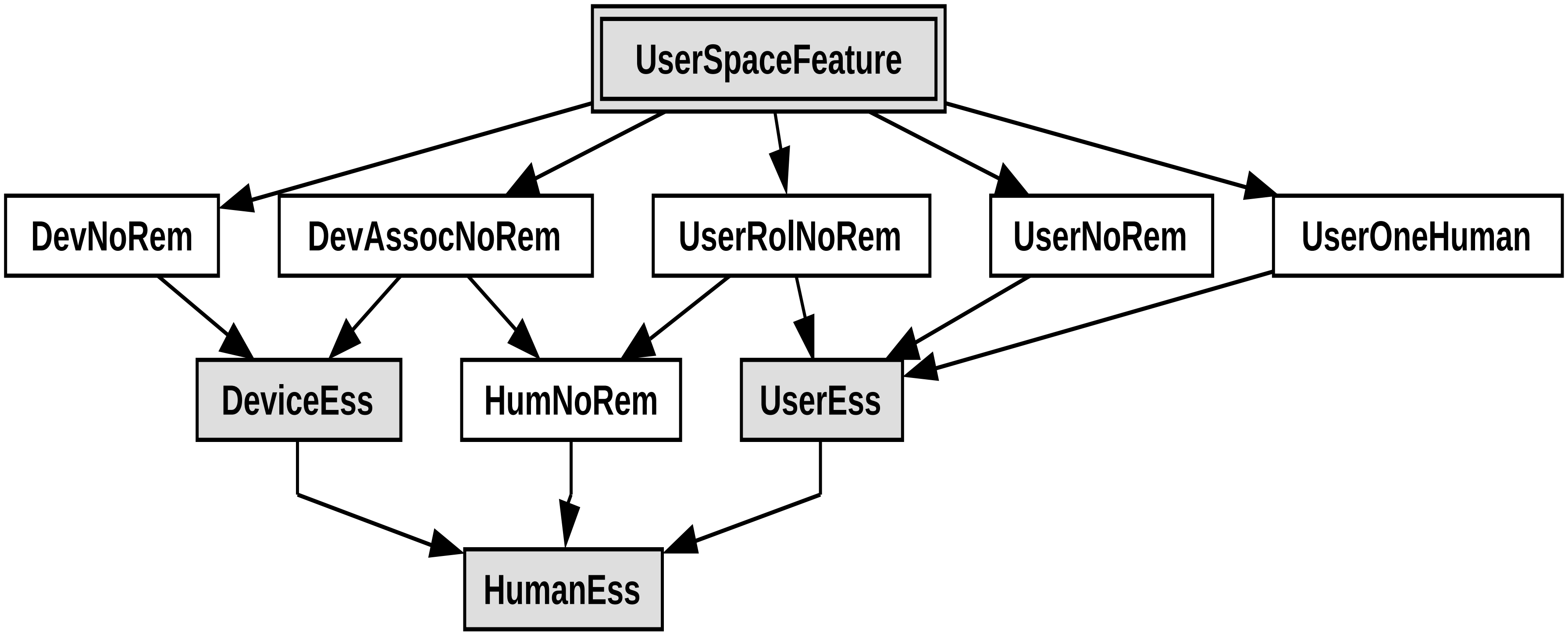
Feature Interactions Detected in Case Study

- no interactions between TCS and CF
  - no type errors detectable
- but other problems problems present:
  - both screening features “remove” the same section
  - type rules: warning!
  - manual inspection: contradiction
- resolution: another feature

Documenting Dependences

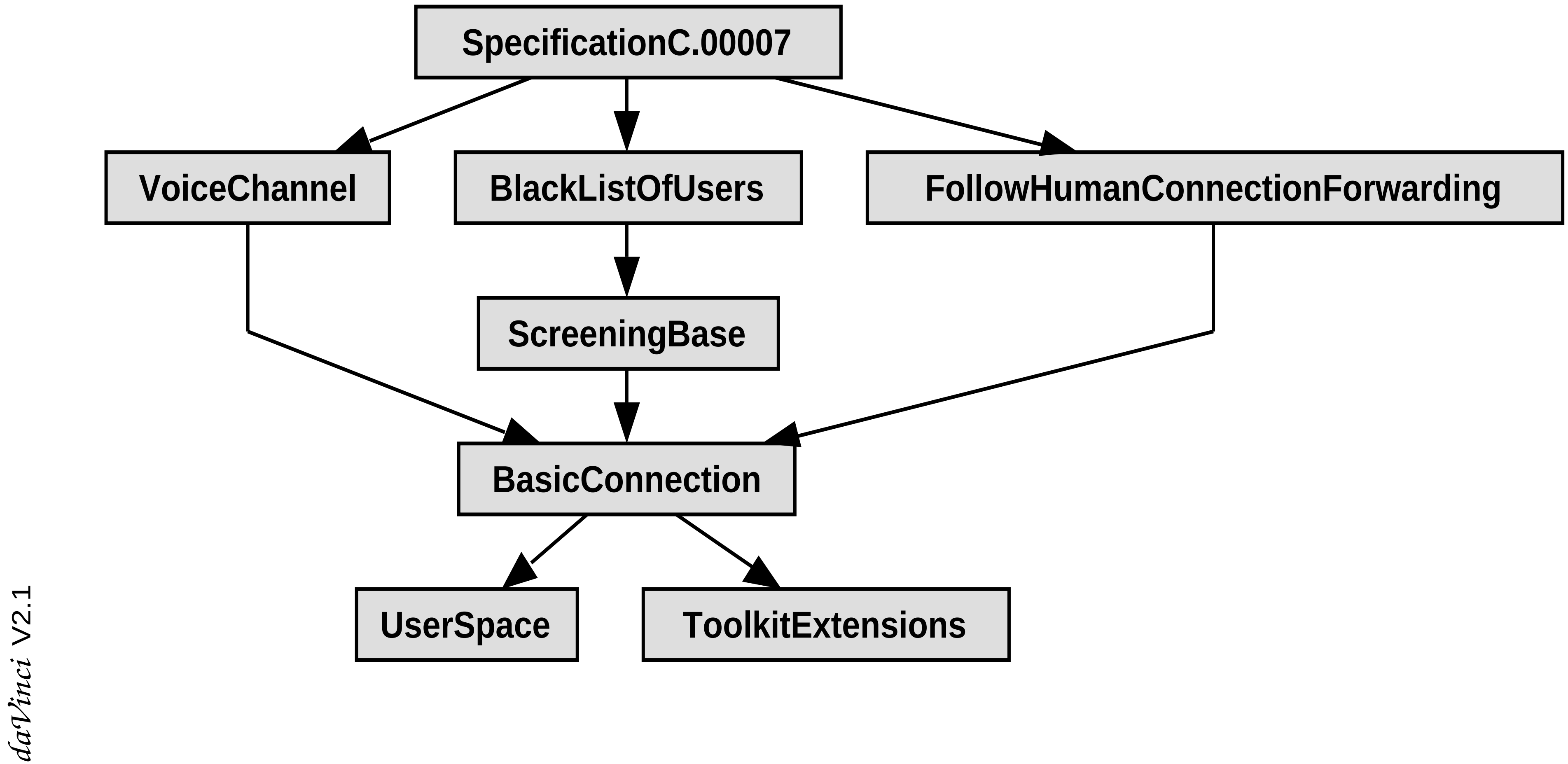
- uses-relation for requirements:
  - use of previous definition
  - reliance on previous constraint
- documented by:
  - Z's section "parents" construct
  - class inheritance (mapped to Z sections)
- if no relationship: identifiers out of scope

Sections of Feature UserSpace



daVinci V2.1

Hierarchy of Features of SpecificationC



Hierarchical Requirements Specification

- a feature can build on other features
- in contrast to the Intelligent Network
- possible to have feature providing a common base

The Tool genFamMem 2.0

- extracts specifications in plain CSP-OZ from a family document,
- detects feature interactions by
  - additional type checks for families
  - heuristic warnings
- helps avoiding feature interactions by generating documentation on the structure of the family.

- available freely

Further Tools

- cspozTC
  - type checker for CSP-OZ
- daVinci
  - visualizes uses hierarchy graphs

Semantics of CSP-OZ Extension

- formal definition of language extension in [Bre00b]
  - understand details: need to know Object-Z and CSP

What Is Still To Do?

- more experience – extend case study further
- apply to other formalisms than CSP-OZ
  - necessary:
    constraint-oriented specification style
    and incremental refinement
  - already supported: CSP_Z and plain Z
- investigate relationship:
  families of requirements – families of programs

Families of CSP Test Specifications

- testing of embedded systems with RT-Tester tool
- RLC layer in UMTS protocol stack
- project with Bosch/Siemens Salzgitter
- requirements specification in CSP
- see [BrSc02]

- light-weight application of previous ideas
  - no consistency checks
  - no documentation generation
  - simple preprocessor for CSP plus method

Flexible Maintenance of Test Specification

- late changes to requirements
- variants of test suites:
  (a) adjust test coverage
    - ▷ selected signal parameters
    - ▷ stimuli: random → increased probabilities → deterministic

  (b) component / integration tests
    - ▷ different protocol layers
    - ▷ parallel instances of same layer

  (c) active / passive tests

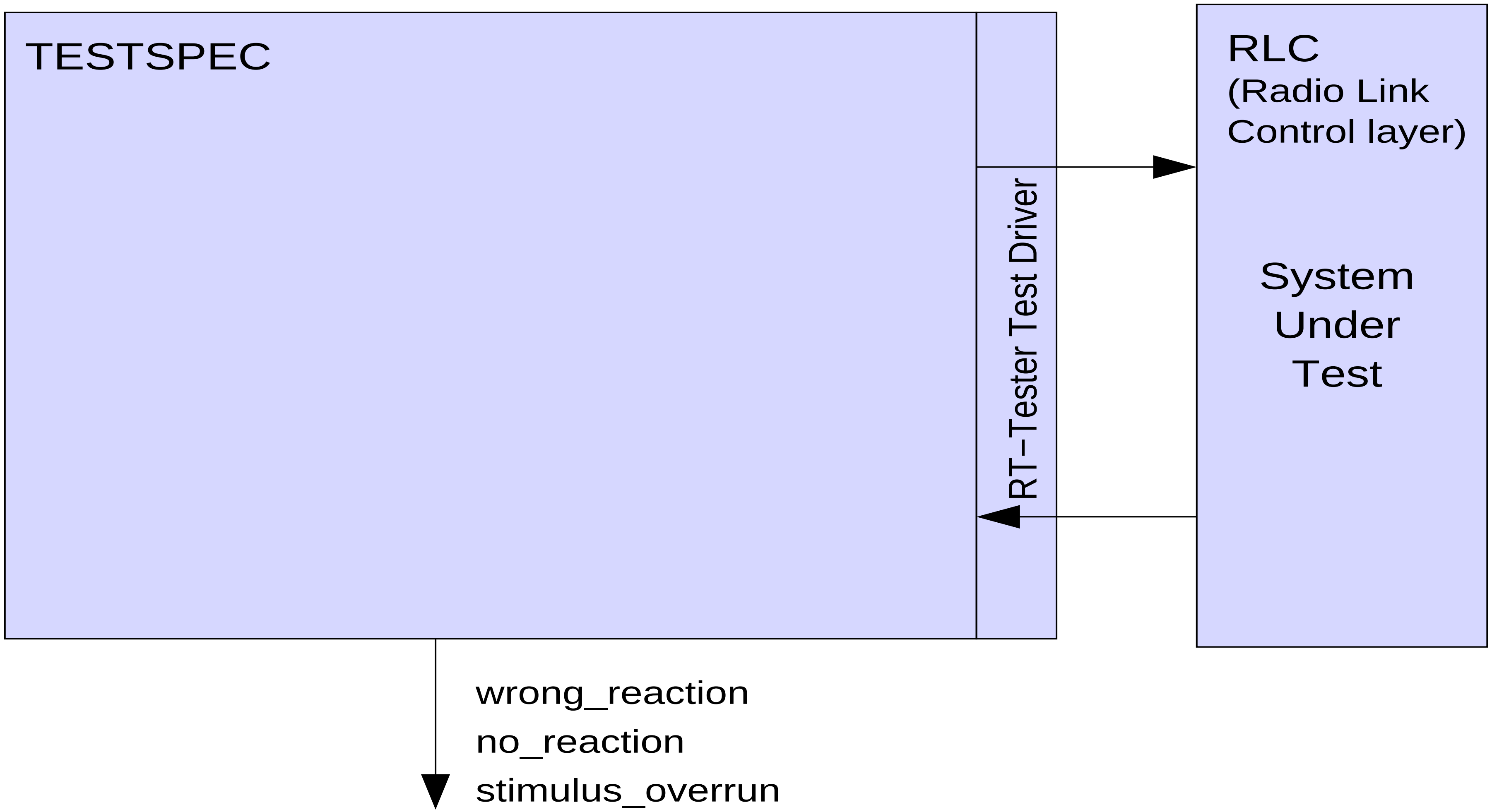
  ⇒ a family of test suites

Rules for Modularizing Requirements

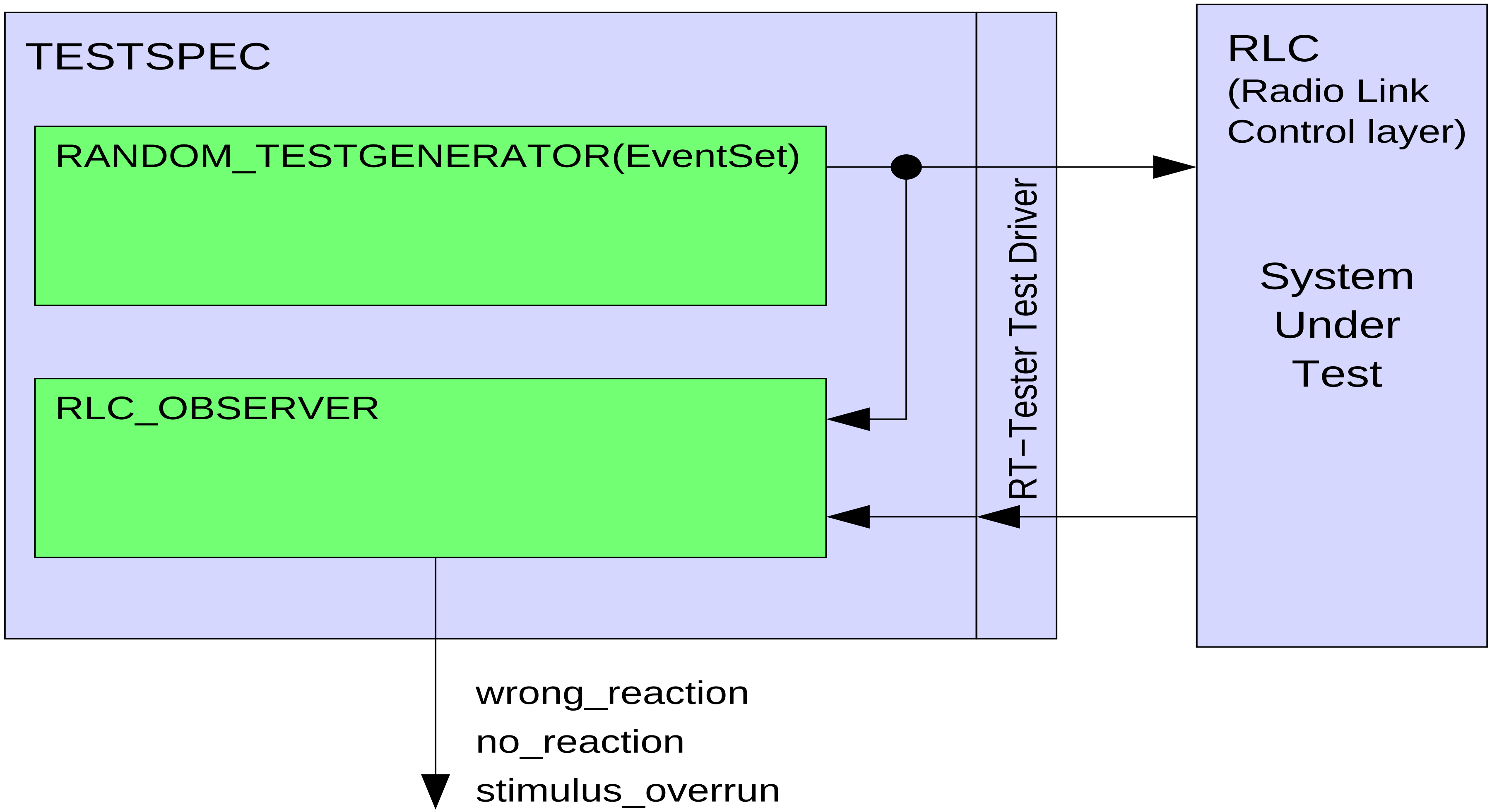
- separate: signature / behaviour of module
- identify requirements that will change together, put into one module

  specifically, separate:
  - tester specific issues / application
  - timer handling / application
  - protocol layers
  - stimulus generation / test observation

Separate:
Test Stimulus Generation / Test Observation



Cont.: Separate: Test Stimulus Generation / Test Observation



Summary of Lecture

- safety-critical systems
  - quality does matter
- professional engineering
  - "blueprint before build"
    - ▷ Chapter 2: what information in computer system documentation?
- embedded software systems
  - "ugly", strict interface constraints
    - ▷ Chapter 1: rigorous description of requirements
  - interface changes all the time
    - ▷ Chapter 3: decomposition into modules
    - ▷ Chapter 4: families of systems

5. Appendix

References

[Bre98] Bredereke, J. *Requirements specification and design of a simplified telephone network by Functional Documentation*. CRL Report 367, McMaster University, Hamilton, Ontario, Canada (Dec. 1998).

[Bre99] Bredereke, J. *Modular, changeable requirements for telephone switching in CSP-OZ*. Tech. Rep. IBS-99-1, University of Oldenburg, Oldenburg, Germany (Oct. 1999).

[Bre00a] Bredereke, J. *Families of formal requirements in telephone switching*. In Calder, M. and Magill, E., editors, “Feature Interactions in Telecommunications and Software Systems VI”, pp. 257–273, Amsterdam (May 2000). IOS Press.

[Bre00b] Bredereke, J. *genFamMem 2.0 Manual – a Specification Generator and Type Checker for Families of Formal Requirements*. University of Bremen (Oct. 2000). URL http://www.tzi.de/~brederek/genFamMem/.

[Bre00c] Bredereke, J. *Hierarchische Familien formaler Anforderungen*. In Grabowski, J. and Heymer, S., editors, “Formale Beschreibungstechniken für verteilte Systeme – 10. GI/ITG-Fachgespräch”, pp. 31–40, Lübeck, Germany (June 2000). Shaker Verlag, Aachen, Germany.

[Bre00d] Bredereke, J. *Specifying features in requirements using CSP-OZ*. In Gilmore, S. and Ryan, M., editors, “Proc. of Workshop on Language Constructs for Describing Features”, pp. 87–88, Glasgow, Scotland (15–16 May 2000). ESPRIT Working Group 23531 – Feature Integration in Requirements Engineering.

[Bre01a] Bredereke, J. *Ein Werkzeug zum Generieren von Spezifikationen aus einer Familie formaler Anforderungen*. In Fischer, S. and Jung, H. W., editors, "Formale Beschreibungstechniken – 11. GI/ITG-Fachgespräch", Bruchsal, Germany (June 2001). URL http://www.i-u.de/fbt2001/.

[Bre01b] Bredereke, J. *A tool for generating specifications from a family of formal requirements*. In Kim, M., Chin, B., Kang, S., and Lee, D., editors, "Formal Techniques for Networked and Distributed Systems", pp. 319–334. Kluwer Academic Publishers (Aug. 2001).

[Bre02] Bredereke, J. *Maintaining telephone switching software requirements*. IEEE Commun. Mag. **40**(11), 104–109 (Nov. 2002).

[BrSc02] Bredereke, J. and Schlingloff, B.-H. *An automated, flexible testing environment for UMTS*. In Schieferdecker, I., König, H., and Wolisz, A., editors, "Testing of Communicating Systems XIV – Application to Internet Technologies and Services", pp. 79–94. Kluwer Academic Publishers (Mar. 2002).

[Cou85] Courtois, P.-J. *On time an space decomposition of complex structures*. Commun. ACM **28**(6), 590–603 (June 1985).

[HBPP81] Heninger Britton, K., Parker, R. A., and Parnas, D. L. *A procedure for designing abstract interfaces for device interface modules*. In "Proc. of the 5th Int'l. Conf. on Software Engineering – ICSE 5", pp. 195–204 (Mar. 1981).

[HoWe01] Hoffman, D. M. and Weiss, D. M., editors. *Software Fundamentals – Collected Papers by David L. Parnas*. Addison-Wesley (Mar. 2001).

[JaKh99] Janicki, R. and Khedri, R. *On a formal semantics of tabular expressions*. CRL Report 379, McMaster University, Hamilton, Ontario, Canada (Sept. 1999).

[Kat93] Katz, S. *A superimposition control construct for distributed systems*. ACM Trans. Prog. Lang. Syst. **15**(2), 337–356 (Apr. 1993).

[Lam88] Lamb, D. A. *Software Engineering: Planning for Change*. Prentice-Hall (1988).

[LaRö01] Lankenau, A. and Röfer, T. *The Bremen Autonomous Wheelchair – a versatile and safe mobility assistant*. IEEE Robotics and Automation Magazine, "Reinventing the Wheelchair" **7**(1), 29–37 (Mar. 2001).

[Mil98] Miller, S. P. *Specifying the mode logic of a flight guidance system in CoRE and SCR*. In "Second Workshop on Formal Methods in Software Practice", Clearwater Beach, Florida, USA (4–5 Mar. 1998).

[PaCl86] Parnas, D. L. and Clements, P. C. *A rational design process: how and why to fake it*. IEEE Trans. Softw. Eng. **12**(2), 251–257 (Feb. 1986).

[PaMa95] Parnas, D. L. and Madey, J. *Functional documents for computer systems*. Sci. Comput. Programming **25**(1), 41–61 (Oct. 1995).

[Par72] Parnas, D. L. *On the criteria to be used in decomposing systems into modules*. Commun. ACM **15**(12), 1053–1058 (1972).

[Par74] Parnas, D. *On a 'buzzword': Hierarchical structure*. In "IFIP Congress 74", pp. 336–339. North-Holland (1974). Reprinted in [HoWe01].

[Par76] Parnas, D. L. *On the design and development of program families*. IEEE Trans. Softw. Eng. **2**(1), 1–9 (Mar. 1976).

[Par77] Parnas, D. L. *Use of abstract interfaces in the development of software for embedded computer systems*. NRL Report 8047, Naval Research Lab., Washington DC, USA (3 June 1977). Reprinted in Infotech State of the Art Report, Structured System Development, Infotech International, 1979.

[Par79] Parnas, D. L. *Designing software for ease of extension and contraction*. IEEE Trans. Softw. Eng. **SE-5**(2), 128–138 (Mar. 1979).

[PaWe85] Parnas, D. L. and Weiss, D. M. *Active design reviews: Principles and practices*. In "Proc. of the 8th Int'l Conf. on Software Engineering – ICSE 8", London (Aug. 1985).

[PCW85] Parnas, D. L., Clements, P. C., and Weiss, D. M. *The modular structure of complex systems*. IEEE Trans. Softw. Eng. **11**(3), 259–266 (Mar. 1985).

[Pet00] Peters, D. K. *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster Univ., Hamilton, Canada (Jan. 2000).

[vSPM93] van Schouwen, A. J., Parnas, D. L., and Madey, J. *Documentation of requirements for computer systems*. In "IEEE Int'l. Symposium on Requirements Engineering – RE'93", pp. 198–207, San Diego, Calif., USA (4–6 Jan. 1993). IEEE Comp. Soc. Press.

[WeLa99] Weiss, D. M. and Lai, C. T. R. *Software Product Line Engineering – a Family-Based Software Development Process*. Addison Wesley Longman (1999).

[Zav01] Zave, P. *Requirements for evolving systems: A telecommunications perspective*. In “5th IEEE Int'l Symposium on Requirements Engineering”, pp. 2–9. IEEE Computer Society Press (2001).