Higher-Order Processes - A Specification Formalism for Interactive Systems

Anke Dittmar

University of Rostock

Our Perspective on Interactive Systems

- Systems are used to analyze and to establish behavior, e.g. ecosystems, economic systems, computing systems, cognitive systems...
- Systems consist of interacting sub-systems, which can exist independently and have similar properties as the whole system but in a simpler form,
- Higher-order processes to facilitate a unified behavioral and structural description of interactive systems.

Higher-Order Processes

Based on **operations** as smallest units of analysis and synthesis:

- no inner structure,
- without interruption,
- described by pre- and postconditions.

Processes are abstractions over operations:

- with inner structure: components and sub-processes,
- describe behavior by sets of alternative sequences of operations (interruption is inherent),
- **components** of a process P:
  - are processes themselves
  - P describes their interaction,
  - constitute the environment of P,
- **sub-processes** of a process P:
  - describe partial behavior of P,
  - introduce operations, which were not in the focus of P.

Process P = (C_b, C_{add}, Ops, Sub, Beh)

-- Part of the Intensional Description

C_b: set of basic components,

C_{add}: set of additional components,

Ops: set of operations known to P with $Ops = Ops_b \cup Ops_{add} \cup Ops_n$

Ops_b – set of operations of basic components (initial focus of P),

Ops_{add} – set of operations of additional components introduced by sub-processes of P,

Ops_n – set of operations defined by P,

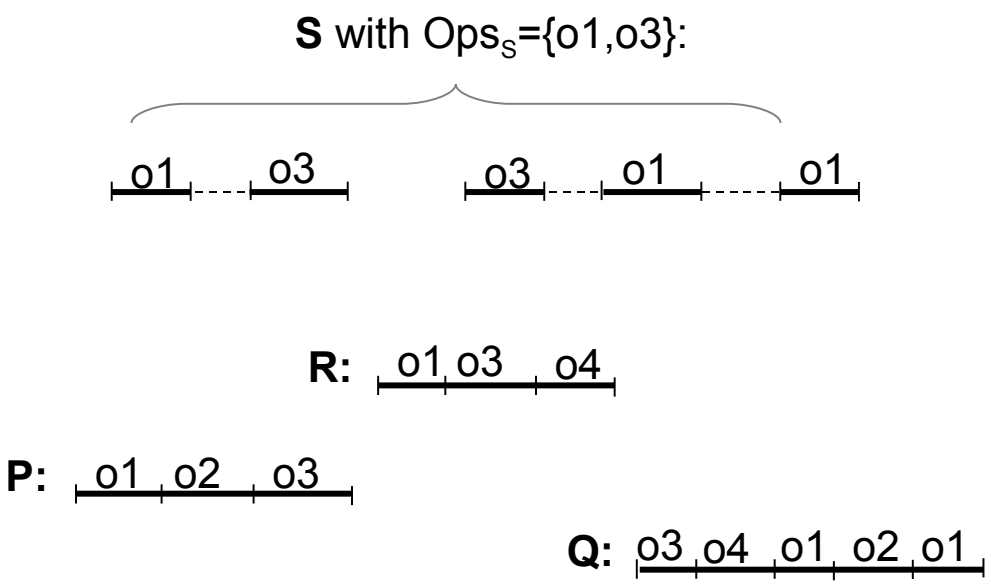
Sub: set of sub-processes of P,

Beh: set of alternative sequences of operations of P ($Beh \subseteq Ops^*$)

Higher-Order Process -- Extensional Description

Let P = (C_b, C_{add}, Ops, Sub, Beh) be a process.

The extensional description of P comprises each sequence seq of operations with seq $|^{\backslash}$ Ops $\in$ Beh.



HOPS: A Prototypical Specification Language for Higher-Order Processes

Example specifications of low-level processes:

```
1 PROCESS Bool
2 OPS
3    true,
4    false
5 SUB PROCESSES
6    Bool = T XOR F,
7       T = false; F,
8       F = true; T
9 END PROCESS
```

```
1 PROCESS Engine
2 OPS    start,
3        stop
4 SUB PROCESSES
5    Engine = (Off ; On)*,
6    On = stop,
7    Off = start
8 END PROCESS
```

Bool = (∅, ∅,
{true,false},
{Bool,T,F},
{ ⟨false,true,false,...⟩,
⟨true,false,true,...⟩ })

Engine = (∅, ∅,
{start,stop},
{Engine,On,Off},
{ ⟨⟩, ⟨start,stop⟩,
⟨start,stop,start,stop⟩,...})

Initial Behavior of Higher-Level Processes

...implicitely determined by the concurrent composition of basic components

```
1 PROCESS Vehicle_1
2 BASIC COMPS
3     running: Bool,
4     doors: Doors,
5     motor:  Engine
6 ADDITIONAL COMPS
7     has_power: Bool
8 END PROCESS
```

Example:

Vehicle_1 = running ||| doors ||| motor

with

$Beh_{running}$ = { ⟨false,true,false,...⟩, ⟨true,false,true,...⟩ }

Beh_{doors} = { ⟨open,close,open,...⟩, ⟨close,open,close,...⟩ }

Beh_{motor} = { ⟨⟩, ⟨start,stop⟩, ⟨start,stop,start,stop⟩,...}

An animation run: ⟨doors.close, running.true, doors.open,...⟩

?- **animate('Vehicle_1').**

enabled operations:
1: Stop
2: motor.start
3: doors.close
4: doors.open
5: running.false
6: running.true

1 > **3**

enabled operations:
1: Stop
2: motor.start
3: running.false
4: running.true
5: doors.open

2 > **4**

enabled operations:
1: Stop
2: motor.start
3: doors.open
4: running.false

3 > **3**

enabled operations:
1: Stop
2: motor.start
3: running.false
4: doors.close

4 > ...

Partial Equations

- Support the description of structures of sub-processes, e.g. hierarchies,
- restrict the initial behavior of the process.

```
// Specification Template
PROCESS PId
BASIC COMPS Comp*
ADDITIONAL COMPS Comp*
OPS Op*
SUB PROCESSES
   PartialEquation*
END PROCESS
```

Example:

```
1  PROCESS Vehicle
2  //...
3  SUB PROCESSES
4      Vehicle = Tram XOR Bus,
5      GeneralBeh = ((start ; stop)* ||| (open_doors ; close_doors)*); park,
6      Tram = GeneralBeh AND TramBeh,
7      Bus = GeneralBeh AND BusBeh,
8      TramBeh = (collector_up ; ring* ; collector_down)*,
9      BusBeh = (motor.start ; motor.stop)* ||| honk*
10 END PROCESS
```

;	sequence
|||	concurrency
*	iteration

Predefined Operators

restrict the behavior in two ways:

- behavioral operators **compose** valid sequences,
- structural operators **combine** different sets of valid sequences.

Behavioral operators:

;	sequence
|||	concurrency
[]	alternative
*	iteration
[...]	option

Structural operators:

AND: $Beh_{(E1\ AND\ E2)} = \{ s \mid s \in (Ops_{E1} \cup Ops_{E2})^* \wedge s \downarrow Ops_{E1} \in Beh_{E1} \wedge s \downarrow Ops_{E2} \in Beh_{E2} \}$

OR, XOR, NOT

Operations Defined by a Process

...restrict the initial behavior

```
1  PROCESS Vehicle
2  BASIC COMPS
3      running: Bool,
4      doors: Doors,
5      motor:  Engine
6  ADDITIONAL COMPS
7      has_power: Bool
8  OPS
9      ring,
10     collector_up:
11        <{has_power: F(Bool)},
12         {has_power: T(Bool)}>  =  <<has_power.true ; motor.start>>,
13     //...
14
15 SUB PROCESSES
16     Vehicle = Tram XOR Bus,
17     GeneralBeh = ((start ; stop)* ||| (open_doors ; close_doors)*); park,
18     Tram = GeneralBeh AND TramBeh,
19     Bus = GeneralBeh AND BusBeh,
20     TramBeh = (collector_up ; ring* ; collector_down)*,
21     BusBeh = (motor.start ; motor.stop)* ||| honk*
22 END PROCESS
```

precondition
(reject some formerly
valid sequences)

post-condition

operationalization part

Sub-processes are used
...for describing states and their manipulation by operations

In the example:

```
1 PROCESS Bool
2 OPS
3    true,
4    false
5 SUB PROCESSES
6    Bool = T XOR F,
7       T = false; F,
8       F = true; T
9 END PROCESS
```

```
...
ADDITIONAL COMPS
  has_power: Bool
OPS
  collector_up:
    <{has_power: F(Bool)},
     {has_power: T(Bool)}> =  <<has_power.true ; motor.start>>,
...
```

Sub-processes are used
...for embedding a set of processes in a common context

In the example:

```
PROCESS Vehicle
...
SUB PROCESSES
   Vehicle = Tram XOR Bus,
   ...
```

```
PROCESS MotorPool
BASIC COMPS
   t_1: Tram(Vehicle),
   t_2: Tram(Vehicle),
   b_1: Bus(Vehicle),...
```

Influences on the Approach

Vygotski: two ways of analysising phenomena

- decomposition into **elements**
- properties of the system explained by associative links between elements

- **atoms** as units of analysis
- can exist alone and can be combined
- own all properties of the whole but at a lower level of organization

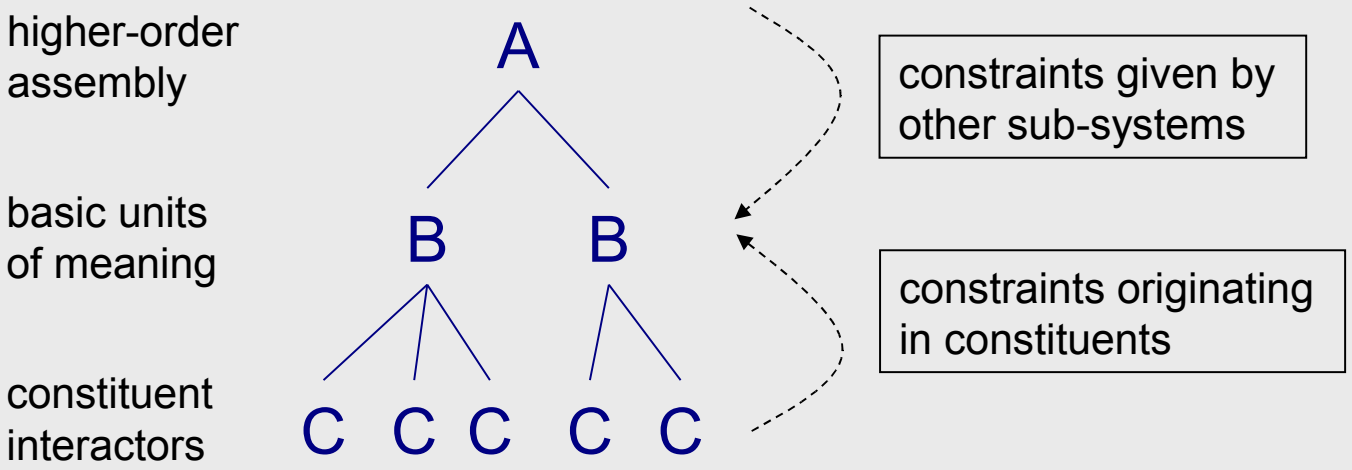
Both approaches can be found in HCI:

e.g. Seeheim model

e.g. Interactors

Interactors (Barnard et al., 2000)

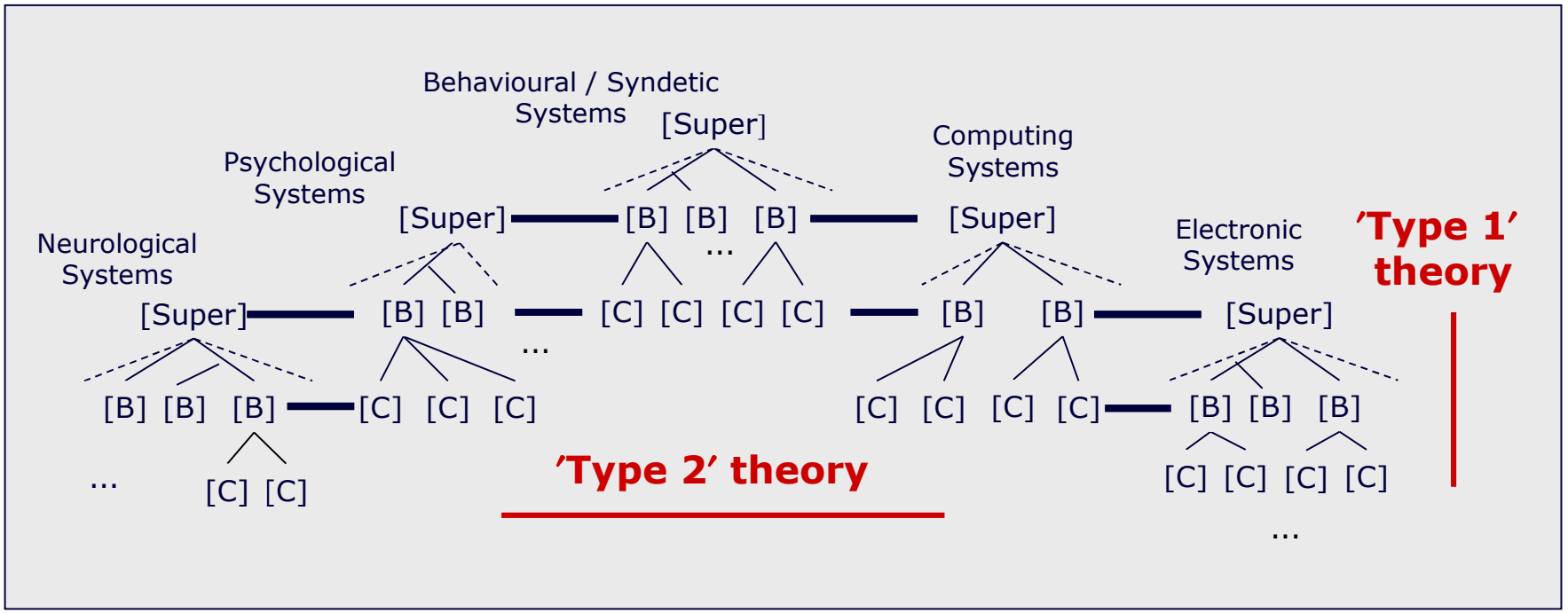
- refer to things that interact
- behave over time
- behavior can be described in terms of lower order interactors of which they are composed



Systems are described as hierarchically organized interactors.

Generic Representations of Systems of Interactors

(Barnard et al., 2000)



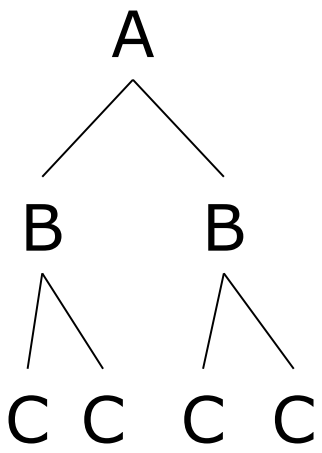
Stronger integration of HCI theories of different sub-domains.

interactor

higher-order assembly

basic units of meaning

constituent interactors



higher-order process

process

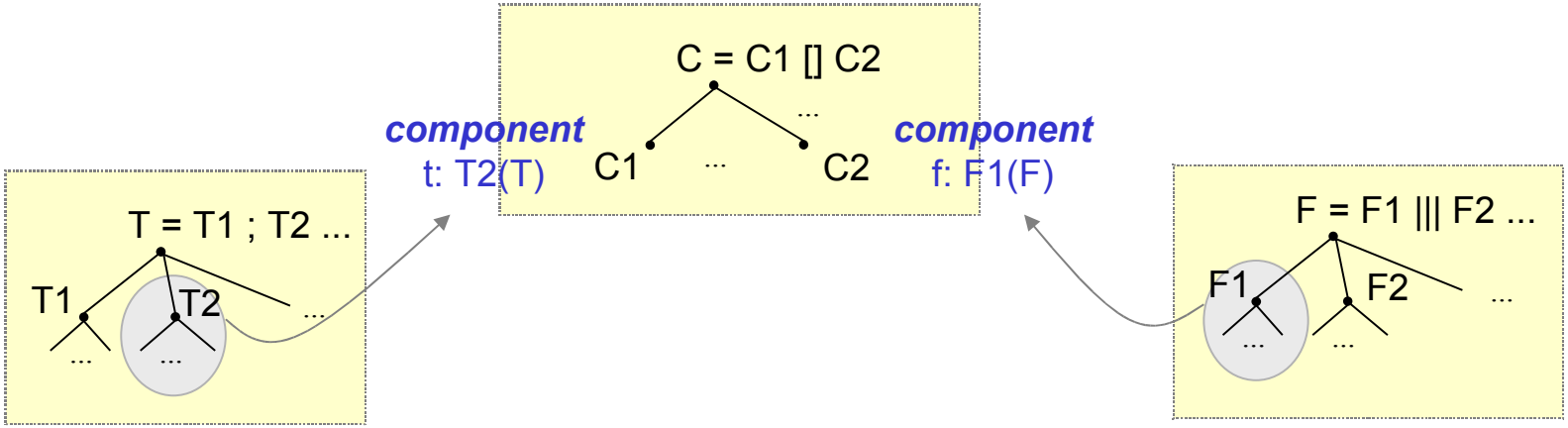
purpose in the context of other processes

sub-processes

patterns for components of other processes

operations

known to the process



HOPS: A Second Example Specification

```
PROCESS Entry_1
OPS
   init,
   edit,
   finish
SUB PROCESSES
   Entry_1 = init ; edit* ; finish
END PROCESS
```

```
PROCESS Space_1

BASIC COMPS
   visible: Bool,
      icon: Bool
OPS
   init  =  << visible.false ; icon.false >>,
   add,
   remove,
   finish,
   show: <{visible: F(Bool)}, {}>   =  visible.true,
   hide: <{visible: T(Bool)}, {}>   =  visible.false,
   iconify: <{visible: T(Bool), icon: F(Bool)}, {}>   =  icon.true,
   deiconify: <{icon: T(Bool), visible: T(Bool)}, {}> =  icon.false,
   action: <{visible: T(Bool), icon: F(Bool)}, {}>

SUB PROCESSES
   Space_1 = init
        ; (add [] remove [] show [] hide [] iconify [] deiconify [] action)*
        ; finish
END PROCESS
```

Addition of „Foreign Code“ to the Specification

- mapping of HOPS-processes to Java-objects
- space_init, space_show, space_hide,space_iconify,... are associated with appropriate methods

```
1  PROCESS Space
2  USES "widgets_java"
3
4  BASIC COMPS
5     visible: Bool,
6        icon: Bool
7  OPS
8     init
9        = <<visible.false ; icon.false ; fCall([this],space_init(this))>>,
10    add,
11    remove,
12    finish  =  <<fCall([this],space_finish(this))>>,
13    show: <{visible: F(Bool)}, {}>
14       = <<visible.true ; fCall([this],space_show(this))>>,
15    hide: <{visible: T(Bool)}, {}>
16       = <<visible.false ; fCall([this],space_hide(this))>>,
17    iconify: <{visible: T(Bool), icon: F(Bool)}, {}>
18       = <<icon.true ; fCall([this],space_iconify(this))>>,
19    deiconify: <{icon: T(Bool), visible: T(Bool)}, {}>
20       = <<icon.false ; fCall([this],space_deiconify(this))>>,
21    action: <{visible: T(Bool), icon: F(Bool)}, {}>
22
23 SUB PROCESSES
24    Space = init ;
25        (add [] remove [] show [] hide [] iconify [] deiconify [] action)*
26        ; finish
27 END PROCESS
```

An Interactive Animation of Process „Space“:

⟨init, add, show, iconify, hide, finish⟩

?- **animate('Space').**
enabled operations:
1: init

1 > **1**
enabled operations:
1: finish
2: add
3: remove
4: show

2 > **2**
enabled operations:
1: finish
2: add
3: remove
4: show

3 > **4**

enabled operations:
1: finish
2: add
3: remove
4: hide
5: iconify
6: action

4 > **5**

enabled operations:
1: finish
2: add
3: remove
4: hide
5: deiconify

5 > **4**
enabled operations:
1: finish
2: add
3: remove
4: show

6 > **1**
enabled operations:
1: Stop

7 > **h**
history:
1 - init
2 - add
3 - show
4 - iconify
5 - hide
6 - finish

Interaction between two Spaces and one Entry

```
1  PROCESS Pres
2  USES "widgets_java"
3
4  BASIC COMPS
5         e: Entry,
6        c1: Space,
7        c2: Space,
8     in_c1: Bool
9
10 OPS
11    init  =  <<e.init ; c1.init ; c2.init>>,
12    add_to_c1
13       = <<c1.add ; in_c1.true ; fCall([c1,e],space_add(c1,e))>>,
14    move_to_c1: <{in_c1:F(Bool)}, {in_c1:T(Bool)}>
15       = <<c2.remove;c1.add;in_c1.true; fCall([c2,e],space_remove(c2,e))
16         ; fCall([c1,e],space_add(c1,e))>>,
17    move_to_c2: <{in_c1:T(Bool)}, {in_c1:F(Bool)}>
18       = <<c1.remove; c2.add;in_c1.false; fCall([c1,e],space_remove(c1,e))
19         ; fCall([c2,e],space_add(c2,e))>>,
20    finish  =  <<e.finish ; c1.finish ; c2.finish>>,
21    edit_in_c1: <{in_c1:T(Bool)}, {}>   =  <<e.edit ; c1.action>>,
22    edit_in_c2: <{in_c1:F(Bool)}, {}>   =  <<e.edit ; c2.action>>
23
24 SUB PROCESSES
25    Pres = init ; add_to_c1
26           ; (move_to_c1 [] move_to_c2 [] edit_in_c1 [] edit_in_c2)*
27           ; finish
28 END PROCESS
```

Process „Pres“ knows

- all operations from e: **init**, **edit**, **finish**,
- from c1, c2: **init**, **finish**, **add**, **remove**, **action**
  ...but not: **show**, **hide**, **iconify**, **deiconify**

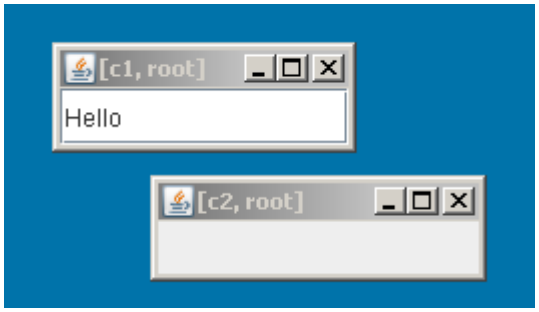
Animation of process „Pres“

enabled operations:
1: init
1 > 1

enabled operations :
1: add_to_c1
2: c1.show
3: c2.show
2 > 1

enabled operations :
1: finish
2: move_to_c2
3: c2.show
4: c1.show
3 > 4

enabled operations :
1: finish
2: move_to_c2
3: edit_in_c1
4: c2.show
5: c1.hide
6: c1.iconify
4 > 4



enabled operations :
1: finish
2: move_to_c2
3: edit_in_c1
4: c1.hide
5: c1.iconify
6: c2.hide
7: c2.iconify
5 > 4

enabled operations :
1: finish
2: move_to_c2
3: c2.hide
4: c2.iconify
5: c1.show
6 > 2

enabled operations :
1: finish
2: move_to_c1
3: edit_in_c2
4: c1.show
5: c2.hide
6: c2.iconify

Higher-Order Processes: Summary

- P. Wegner: „Interactive systems interact with an external environment they cannot control.“
  - Processes do not fully control the specific behavior of their environment (components),
- Interaction paradigm supports nonmonotonicity:
  - decomposition may create interactive unpredictable systems,
  - composition may produce noninteractive algorithms,
- Interaction paradigm supports openness
  - fluid boundary between operations and processes
- Unified behavioral and structural description of interactive systems