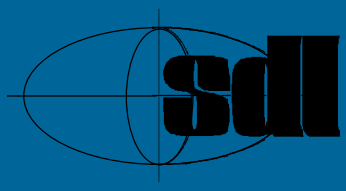
SDL-2000 Tutorial



Prof. J. Fischer
Dr. E. Holz

Humboldt-Universität zu Berlin

SAM 2000 Workshop Grenoble

SDL-2000

- is a major revision of SDL
  - removal of outdated concepts
  - alignment of existing concepts
  - introduction of new concepts
- is completely based on object-orientation
- has a new formal semantics
- is accompanied by a new standard Z.109 SDL-UML-Profil

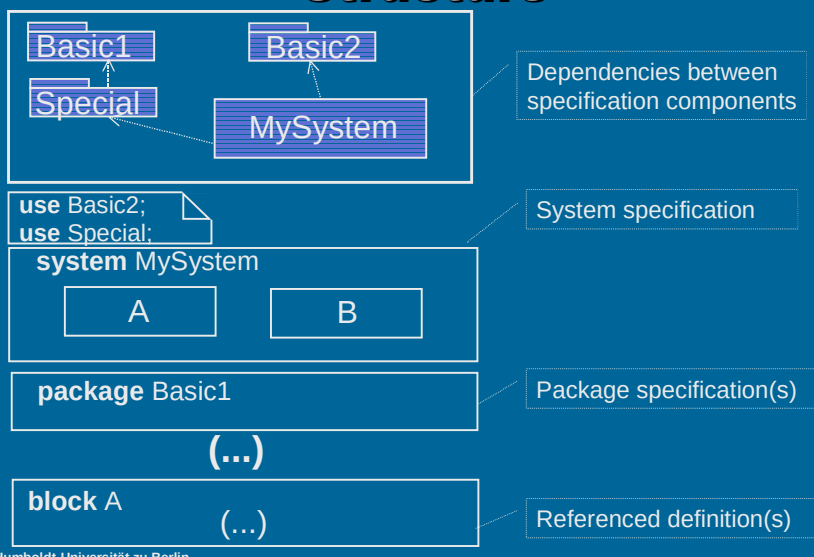
Differences to SDL-92

- document structure of standard has been reorganized
- SDL-2000 is case-sensitive
  - two spellings for keywords all uppercase or all lowercase
  - removed keywords
    all, axioms, constant, endgenerator,endnewtype, endrefinement, endservice, error, fpar, generator, imported, literal, map, newtype, noequal, ordering, refinement, returns, reveal, reverse, service, signalroute, view, viewed
  - new keywords in SDL-2000
    abstract, aggregation, association, break, choice, composition, continue, endexceptionhandler, endmethod, endobject, endvalue, exception, exceptionhandler, handle, method, object, onexception, ordered, private, protected, public, raise, value

- not available constructs in SDL-2000
  - signal routes -> replaced by non-delaying channels
  - view expression -> deleted (import concept, “global” variables)
  - generators -> generalized by parameterized types
  - block substructures -> replaced by nested agents
  - channel substructures -> deleted
  - signal refinement -> deleted
  - axiomatic definition of data -> deleted
  - macro diagrams -> deleted
  - services -> deleted (agents and state aggregations)

SAM 2000, Grenoble

SDL-2000 Specification Structure



5

SAM 2000, Grenoble

Agents

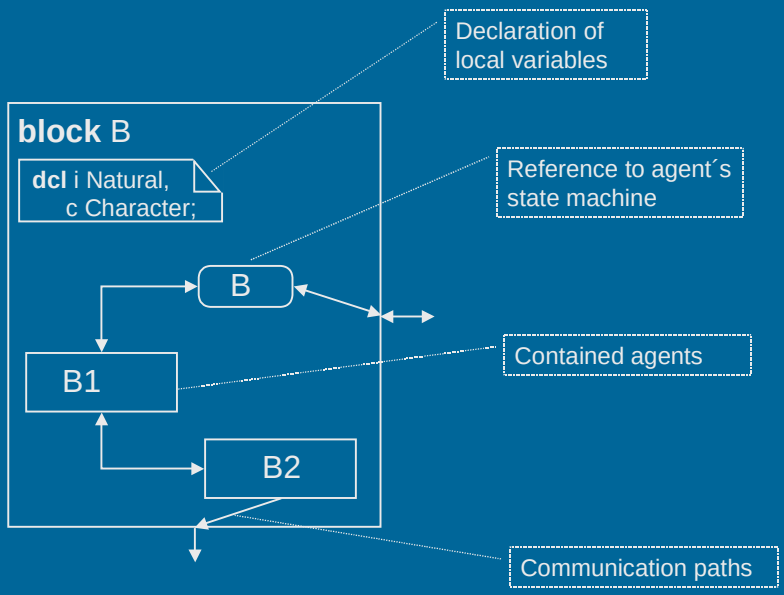
- subsumes the concepts system, block, process
- basic specification concept
- model active components of a system
- an agent instance is an extended finite communicating state machine that has
  - its own identity
  - its own signal input queue
  - its own life cycle
  - a reactive behaviour specification

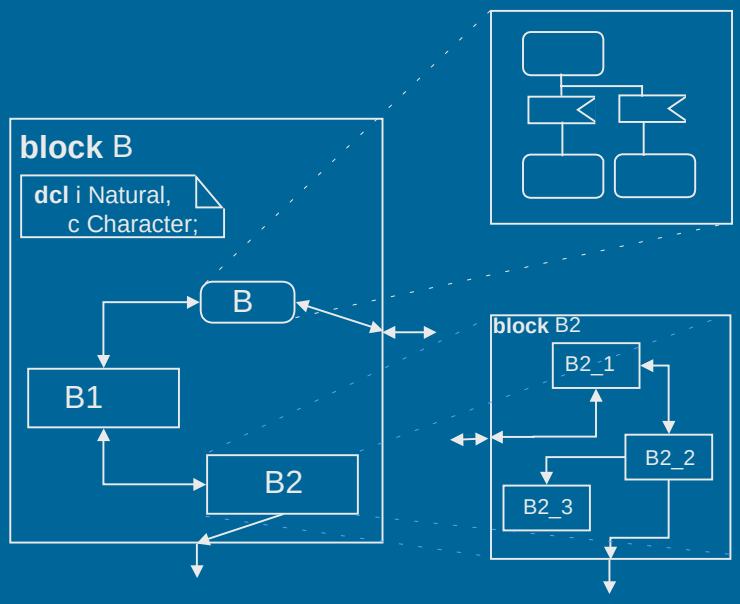
6

Agent Declaration

- three main parts
  - attributes
    parameters, variables, procedures
  - behaviour
    implicit or explicit state machine
  - internal structure
    contained agents and communication paths

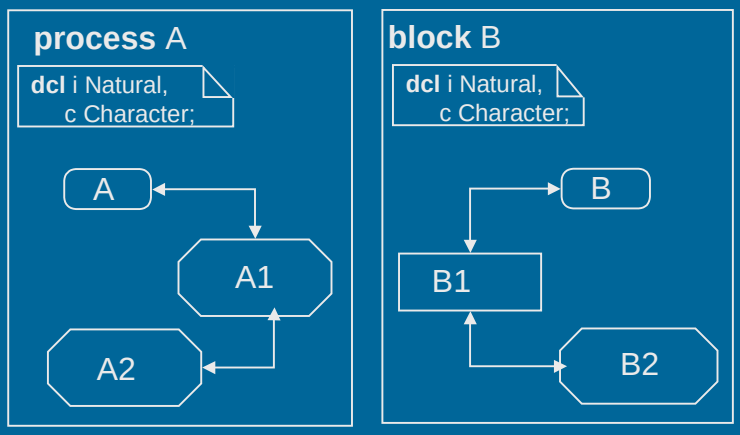
  each of the parts may be optional





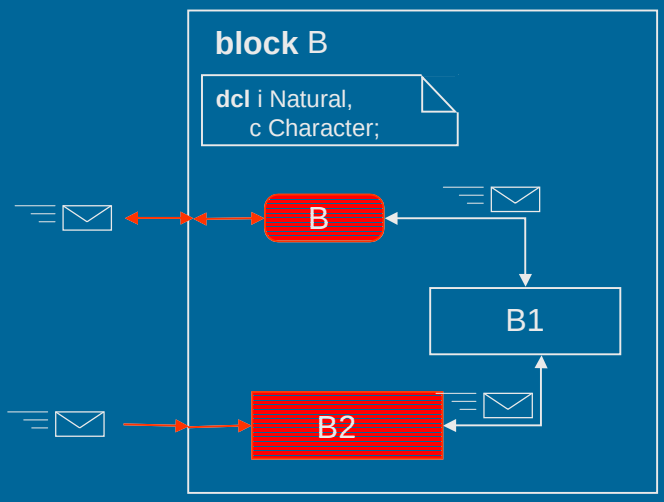
De-Composition of Agents

- structural decomposition into internal agents implies also decomposition of behaviour
- container of an agent determines scheduling semantics of its contents
  - concurrent agents: ***block***
  - alternating agents: ***process***



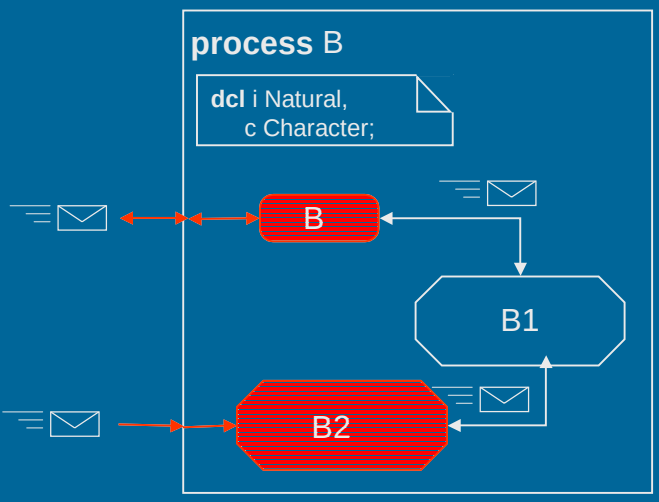
Block Agent

- all contained agents execute concurrently with each other and with the agents state machine
  - multiple threads of control
  - concurrent execution of multiple transitions
  - transitions execute with run-to-completion
- contained agents may be
  - blocks or processes



Process Agent

- all contained agents execute alternating with each other and with the agents state machine
  - at most one transition is executed at any point in time
  - selection is non-determined
  - transitions execute in run-to-completion
- contained agents
  - may be of kind process only

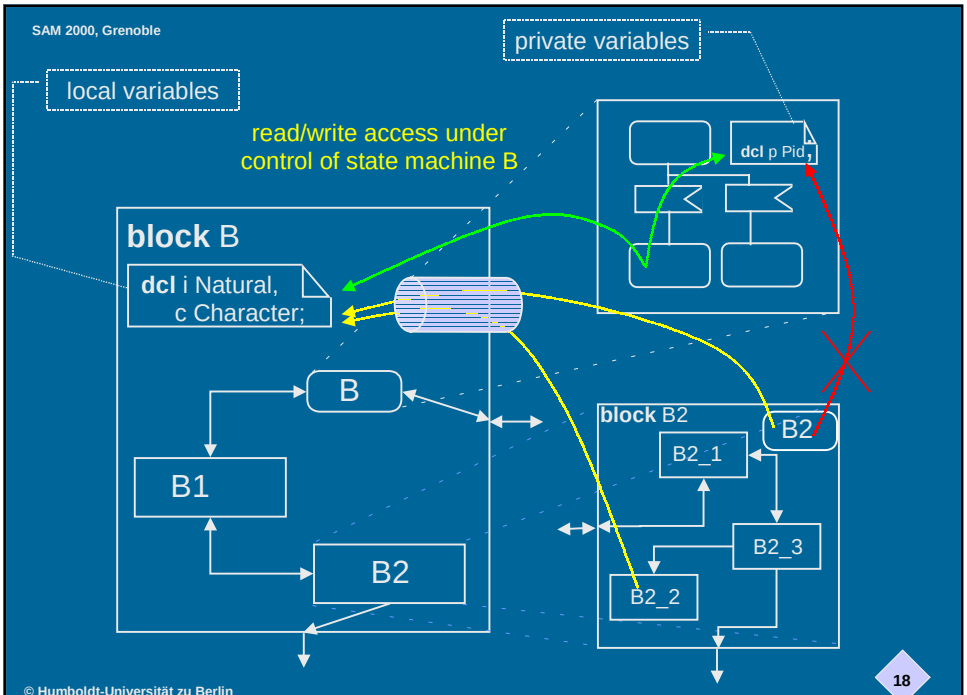


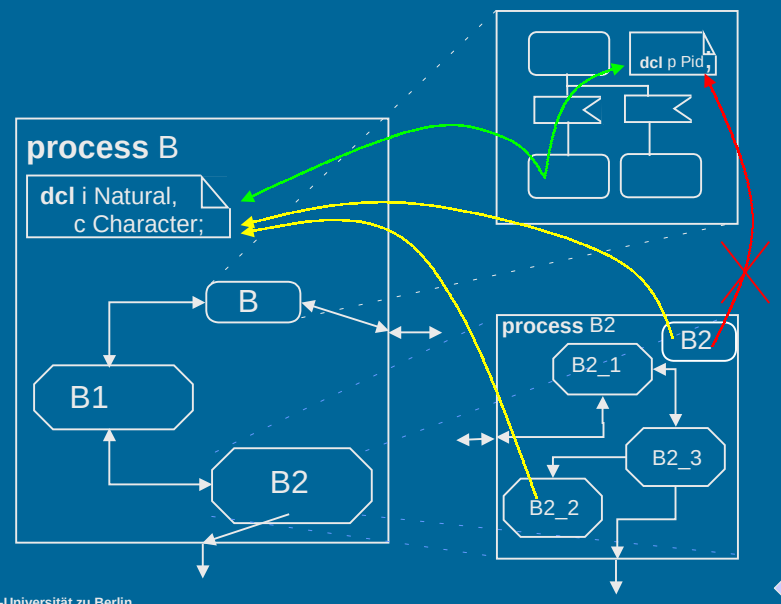
System Agent

- a system is one special (block) agent
  - must be the outermost agent
  - defines the border to the environment
  - can define communication primitives exchanged with the environment

Variables in Agents

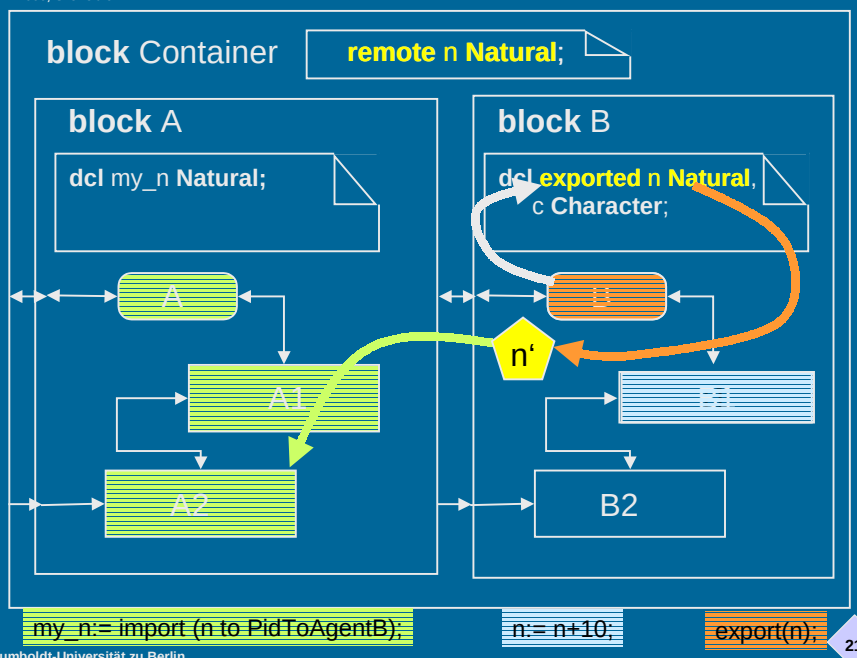
- variables
  - store data local to an agent
  - owned by agents state machine
- private variables, visible only to
  - agents state machine
- local variables, visible to
  - agents state machine
  - contained agents
- public (exported) variables
  - visibility controlled by *remote* declaration





Remote Variables

- read-access to variables owned by an other agent
  - short-hand notation for signal interchanges
  - provision by *exported*-declaration
  - visible by *remote*-dedclaration
  - no *imported* declaration
  - access with *import*-operation
  - update (by owner) with *export*-operation



General Communication

- communication is based on signal exchange
- a signal carries
  - kind of signal (signal name)
  - user data
  - implicit sender identification (Pid value)

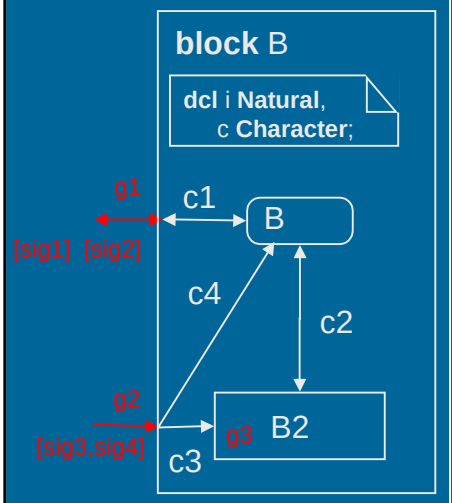
signal aSignal (Natural, Character);

- communication requires a complete path from sender to receiver consisting of
  - gates
  - channels
  - connections
- path may be defined
  - explicitly
  - or implicitly derived

- channel
  - uni- or bi-directional communication path between two endpoints
    - gate, agent, connection, state machine
  - safe and reliable (no loss, no re-ordering,...)
  - delaying or non-delaying transmission
  - name and signallists are optional
  - supersedes also signalroute concept

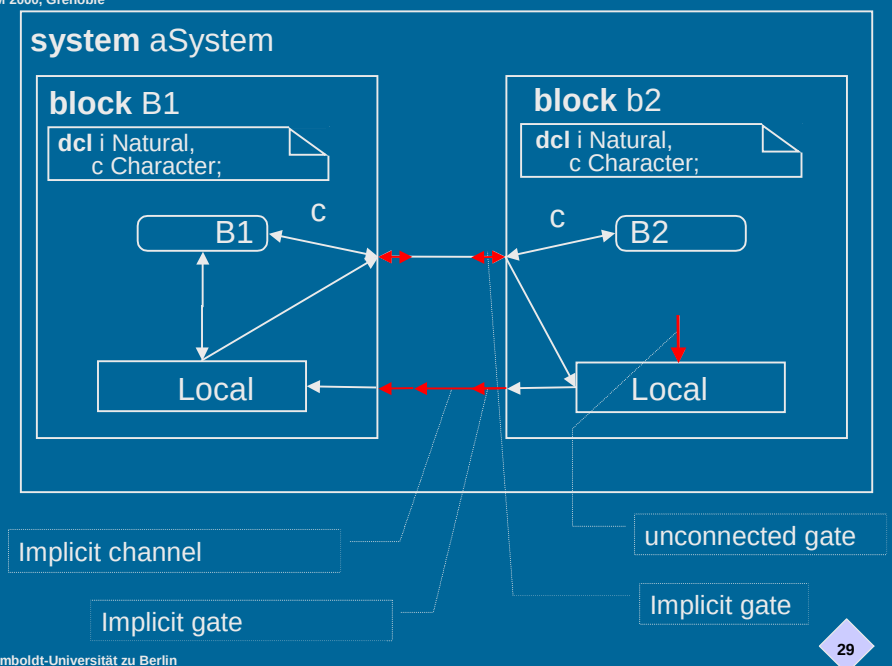
```
channel aChannel nodelay
        from block1 to block2
                with sig1, sig2;
        from block2 to block1
                with sig1, sig3;
endchannel;
channel nodelay
        from block1 to block2;
        from block2 to block1;
endchannel;
```

- gate
  - potential named endpoint for a channel at an agent, agent type or a state machine
  - uni- or bi-directional
  - possibly constrained by set of signals or by interface
- connection
  - joining/splitting of channels at implicit gate




```
block B;
gate g1 in with sig2,
        out with sig1;
gate g2  in with sig3,sig4;

state B referenced;
block B2 referenced;

channel c1      from env via g1 to this;
                from this to env via g1;
endchannel;
channel c2      from this to B2;
                from B2 to this;
endchannel;
channel c3      from env via g2
                to B2 via g3;
endchannel;
channel c4      from env via g2 to this;
endchannel;
endblock B;
```


- implicit signal lists on gates and channels
- implicit gates
  - introduced for connections
  - introduced for unconnected channel endpoints
  - signal lists derived from channel signal lists
- implicit channels
  - introduced for unconnected gates
  - gates have to have matching constraints



Advanced Communication

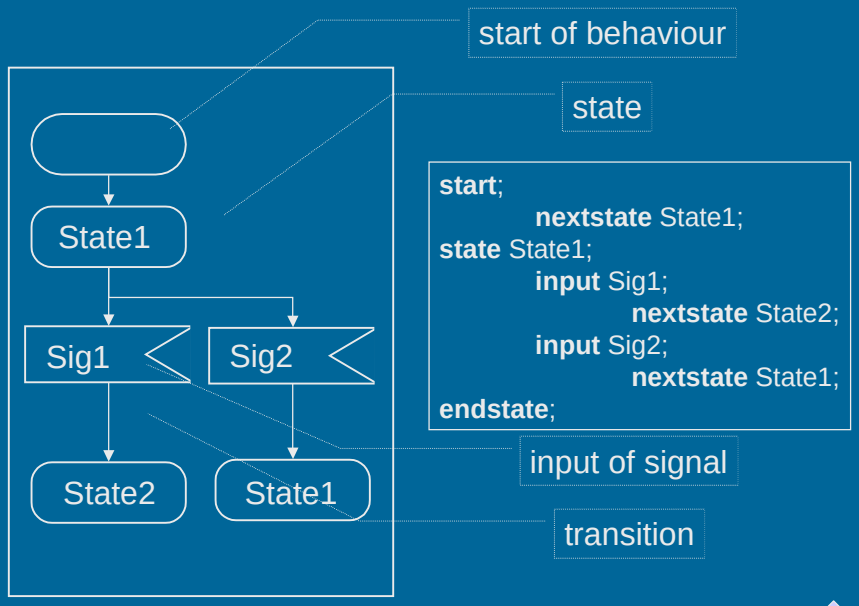
- two-way communication
  - implicitly mapped onto signal exchange
  - remote variables
    - read access to variables of other agents
    - no containment relation required
  - remote procedures
    - execution of a procedure by a different agent
  - remote procedure and and remote variables can be mentioned in signalists

Simple State Machines

- behaviour of an agent
  - is specified by a state machine
- two main constituents:
  - states
    - particular condition in which an agent may consume a signal
  - transitions
    - sequence of activities triggered by the consumption of a signal

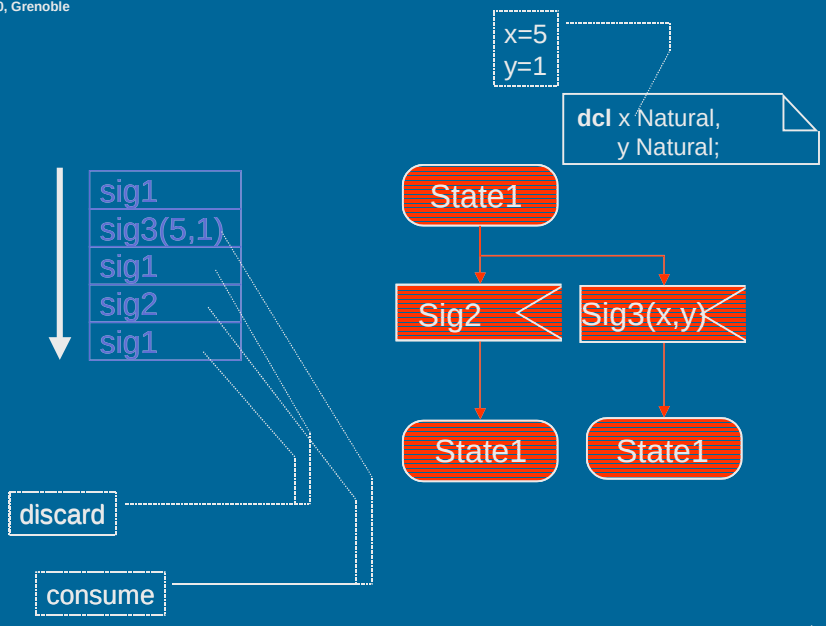
State Transition

- a state machine is always either
  - in a state waiting for a signal
  - or performing a transition
- a transition results in
  - entering a new state or
  - stopping the agent

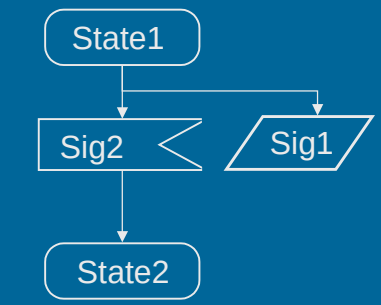


Transition Triggers

- the first signal of input queue is removed by an input that identifies the signal
- if there is no input for the first signal, it is discarded
- during an input data carried by a signal may be copied onto local variables
- reference to originating agent can be obtained by implicit expression **sender**

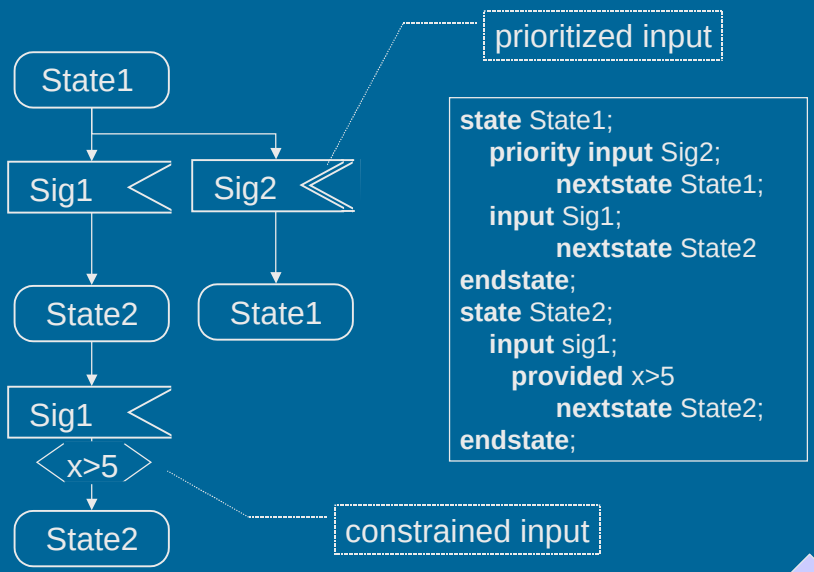


- consumption of signals can be deferred until a new state is entered
  - signals saved in this state
  - valid until a new state is entered
  - avoids implicit discard

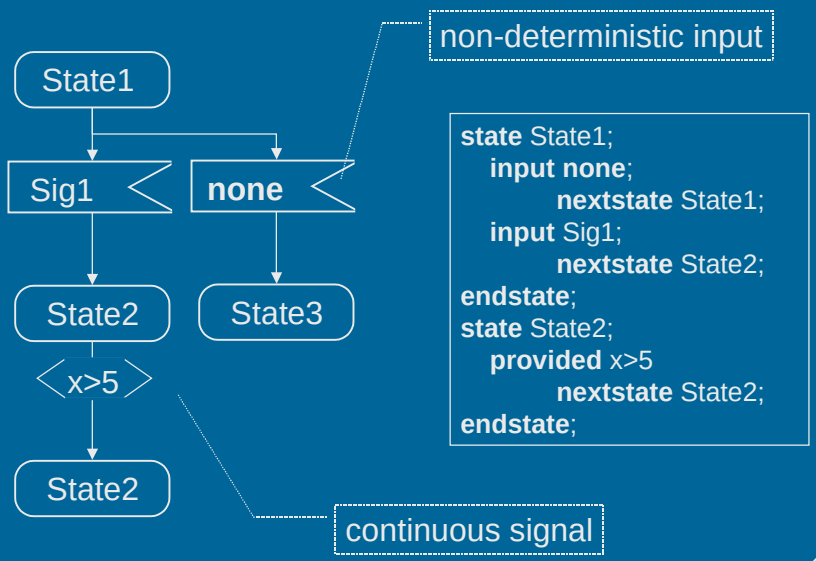



```
state State1;
   input Sig1;
         nextstate State2;
   save Sig1;
endstate;
```

- input of selected signals can be preferred or constrained
  - priority input
    - transition will be selected even if the signal is not the first in the queue
  - enabling condition
    - transition will be selected only in case the attached condition is true (otherwise save)

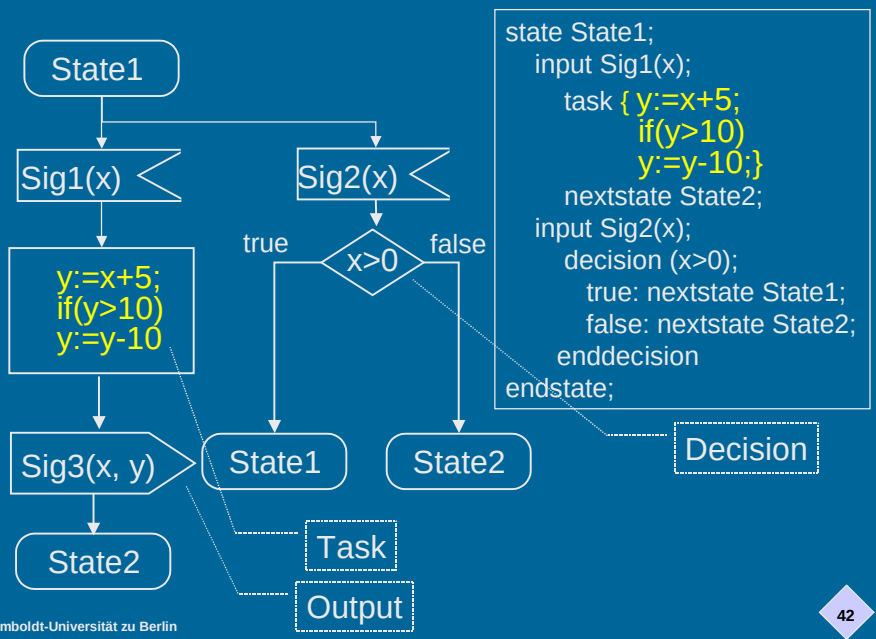


- transitions may also be triggered without an (explicit) signal
  - continuous signal
    - transition will be selected if attached condition is true and no other transition can be selected
      i.e. queue is empty or all other signals are saved
  - non-deterministic transition
    - transition will be selected non-deterministically and independent from any other transition



Transition Actions

- output
  - generation and addressing of signals (identification of receiver or communication path)
- task
  - sequence of simple or compound statements
  - informal text
- decision
  - branching a transition into a series of alternative paths



Algorithmic Notation

- statement lists provide a means for a concise textual notation of algorithms
- applicable in
  - tasks
  - procedure definition
  - operation definition
- programming-language like syntax

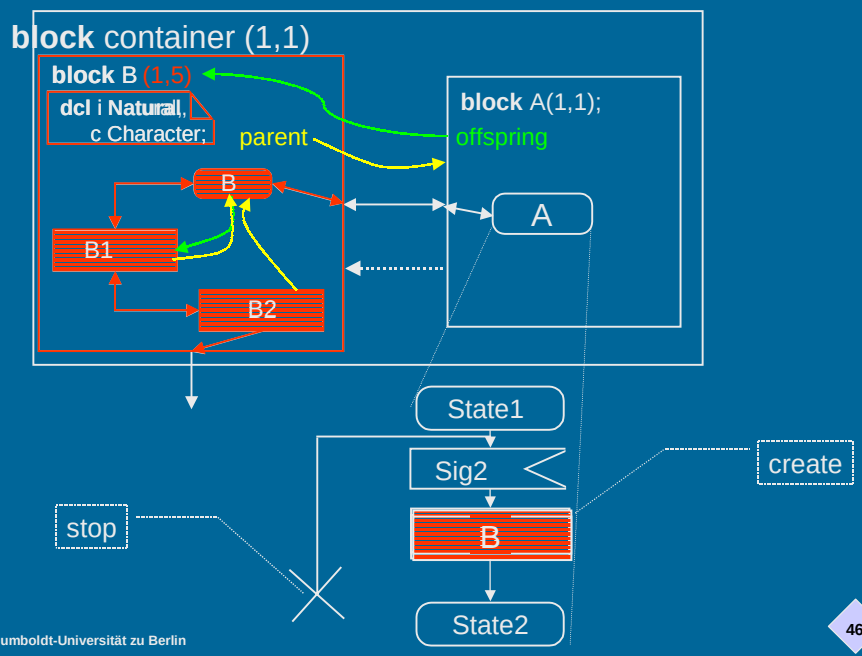
- compound statements
- if-statements
- decision-statements
- loop-statements
- break-statements
- exception-statements

- output, export, return, create,set/reset, raise, call, assignment

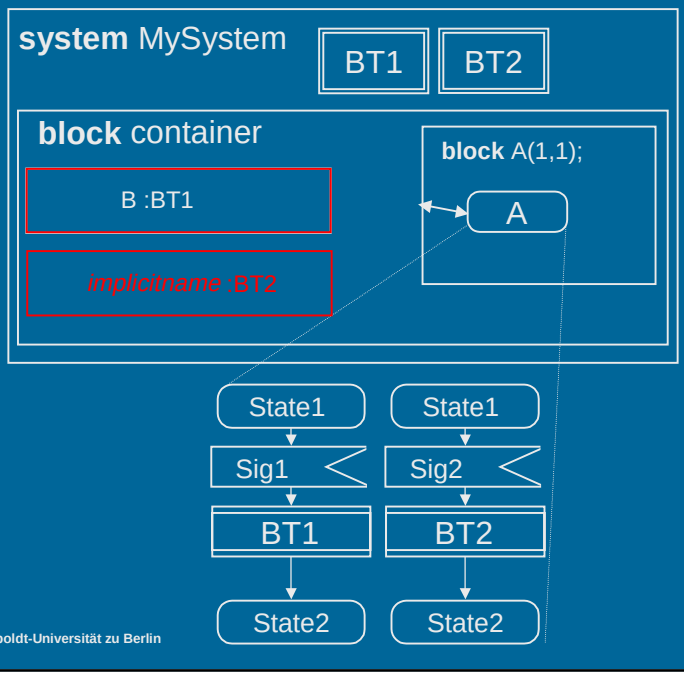
```
task
{
 for(  dcl P Pid :=self, not(P=null);
       dcl i Natural:=0,,i+1)
 { P:=create Proc;
   if(newP=null) break;
   else output Sig(P,i) to parent;
 }
```

Create and Stop

- performance of a create-request action results in the existence of a new agent instance in the indicated agent set
  - creators implicit *offspring* expression refers to new createe
  - createe´s implicit *parent* expression refers to creator
- initial internal structure will be created too



- create request can also be based on a type definition
- new instance will either be created in
  - existing instance set based on that type and defined in the same agent as the creator
  - or implicit instance set based on that type and located in the same agent as the creator



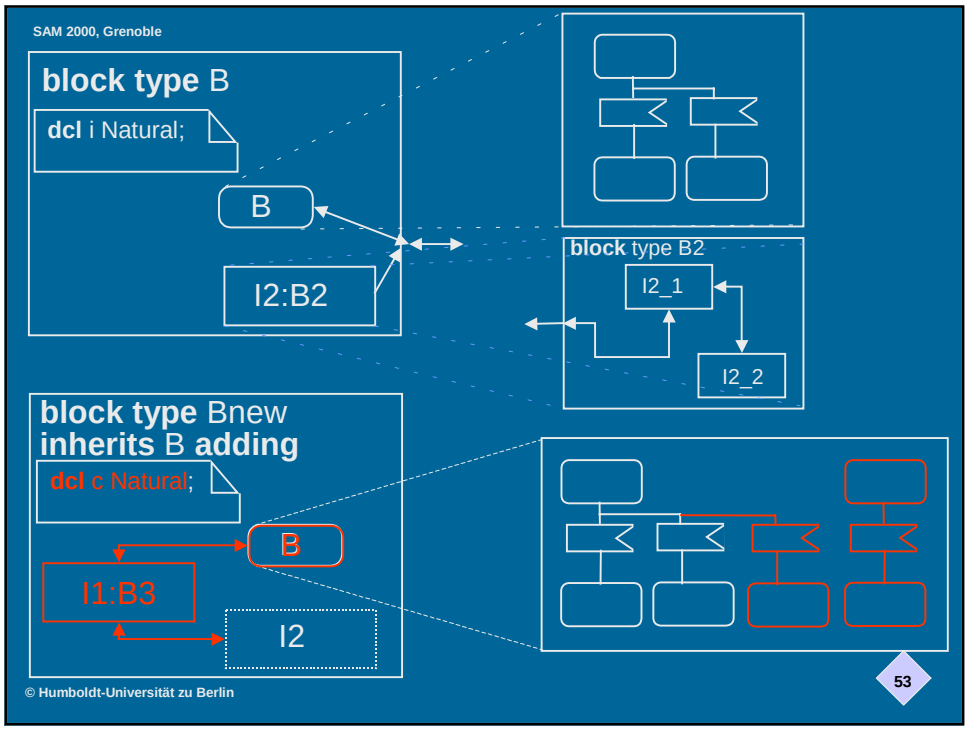
- the execution of a *stop* results in entering a special implicit stopping-state
  - no further execution of transitions
  - if agent contains no further agent instances, it will cease to exist
  - otherwise
    it will the access to the agents variables

Object-Orientation in SDL

- structural typing concepts allow to define the properties of a set of specification elements
- kinds of structural types
  - agent type
  - state type
  - signal (type)
  - procedures (type)
  - data types and interfaces

- type concept corresponds to class concept in other OO languages and notations
  - inheritance
  - virtuality
  - abstraction
  - instance definition & creation
- all instance definitions in SDL are either explicitly or implicitly based on a type

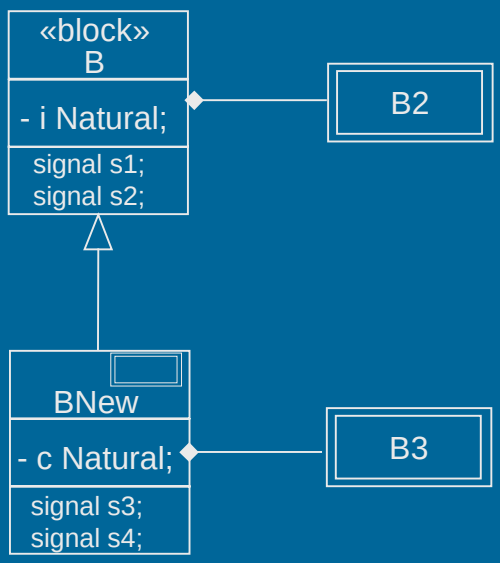
- inheritance allows the definition of a type basing on another (super-) type of the same kind
  - addition of new structural elements
  - addition of new behavioural elements
  - redefinition of virtual elements
- single inheritance supported by all types
- multiple inheritance supported by interface



Type References and Relations

- UML-like class symbols can be used to refer to type definitions and diagrams
- partial specification:
  - type name
  - type attributes
  - type behaviour properties
- multiple references are allowed
  - must all be consistent with type definition

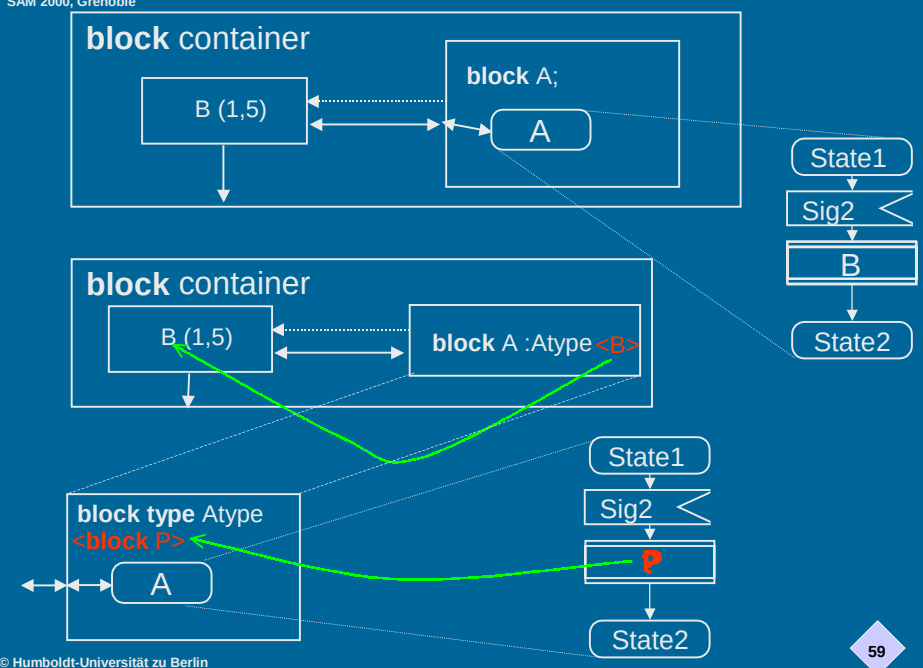
- relations between types can be depicted by
  - associations (binary relations)
    - no predefined semantics implied
  - specialization
    - must be consistent wit a specialization in the type definition (*inherits*-clause)



- type definitions for an element can be given in
  - the scope unit where the entity can be given
  - any surrounding scope unit or
  - any type definition for such a scope unit
  - a package
- instances of such a type can be defined
  - where the type is visible and
  - the element is allowed

Context Dependencies

- types may refer to instances and types in their defining context
  - definition of instances may be limited to the same scope unit (e.g. in case of instance references)
- types may refer to instances and types in their instantiation context
  - formal context parameters in defining context
  - actual context parameters in instantiation context

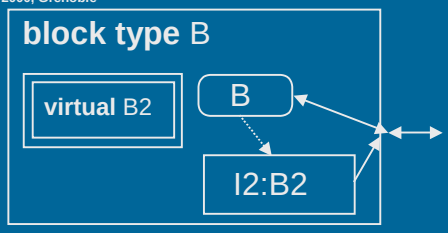


Context Parameter Kinds

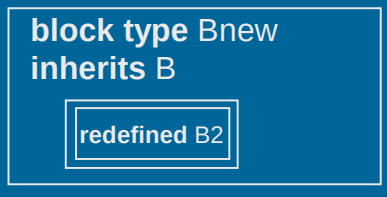
- block (type), process (type)
- composite state type
- procedure, remote procedure
- signal
- data type, interface, exception
- variable, remote variable, timer, synonym
- gate

Abstract and Virtual Types

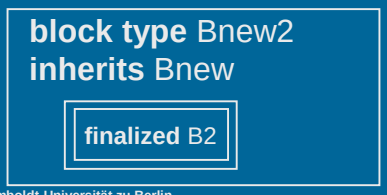
- types marked with the keyword *abstract* do not directly have instances
  - pure classification
  - used as super-type in an inheritance hierarchy
- *virtual* types local to another type may be redefined in a specialisation of that type
  - must be contained in a type definition
  - redefinition can be constrained
- system type can not be abstract or virtual



- I2 is instance set of virtual type B2



- I2 is instance set of redefined type B2

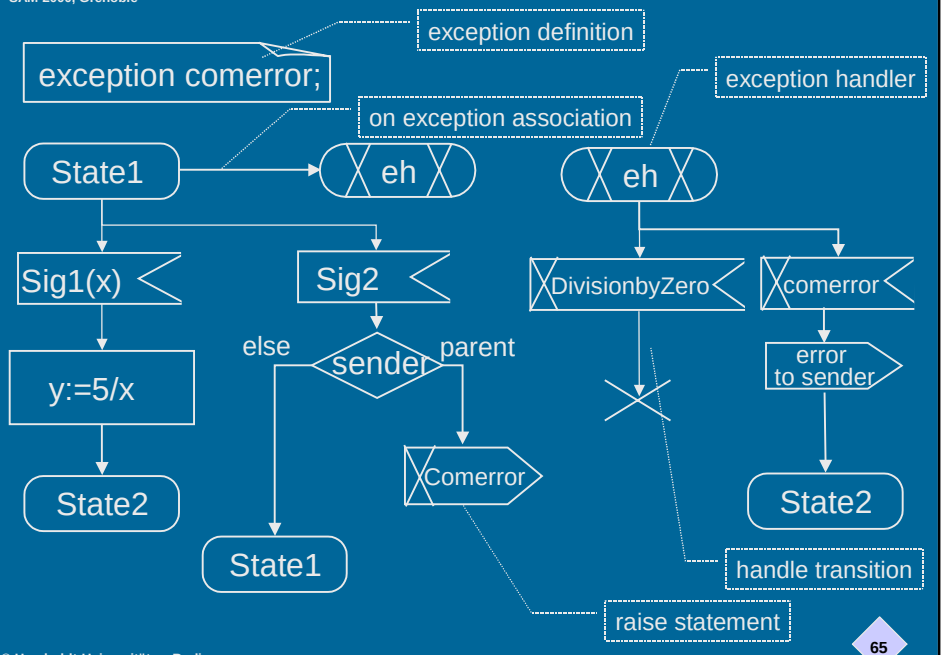


- I2 is instance set of finalised type B2
- no further redefinition allowed

- redefinition/finalisation must be
  - subtype of original virtual type or
  - subtype of virtuality constraint
    (**virtual block type** B **atleast** Base)
- references to a virtual or redefined type refer to the most recent redefinition
- finalised types can not be redefined further

Advanced State Machines

- exceptions are used to denote and handle unexpected or exceptional behaviour
  - exception: the type of cause
  - exception handler:behaviour to occur after an exception (handle-clauses)
  - onexception: attaches exception handler to a behaviour unit
  - raise: forces a transition to throw an exception



- exception handlers can be attached to all kinds of behaviour by an onexception:
  - complete state machine, state,
  - input transition, transition/algorithm action
  - terminator, connect,
  - single procedure, single operation
  - single remote procedure
  - exception handler, handle transition
- in case of an exception the most local active exception handler will be selected

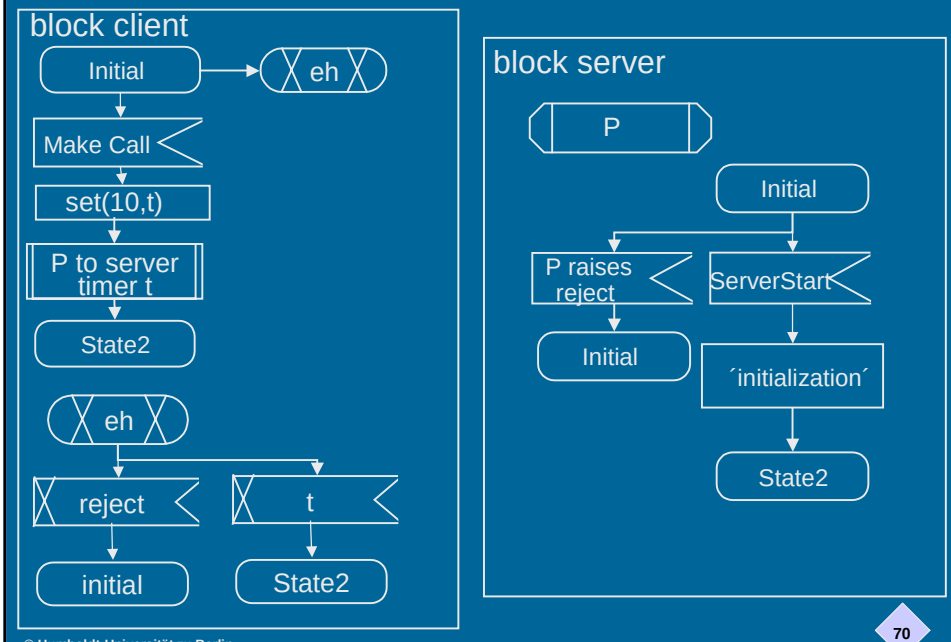
Procedures

- procedures are a means to group and name recurrent behaviour
- notation corresponds to agent state machine
  - local states, inputs and transitions
  - local variables, parameters
- procedures are a type
- exceptions raise but not handled in a procedure are mentioned explicitly

Remote Procedures

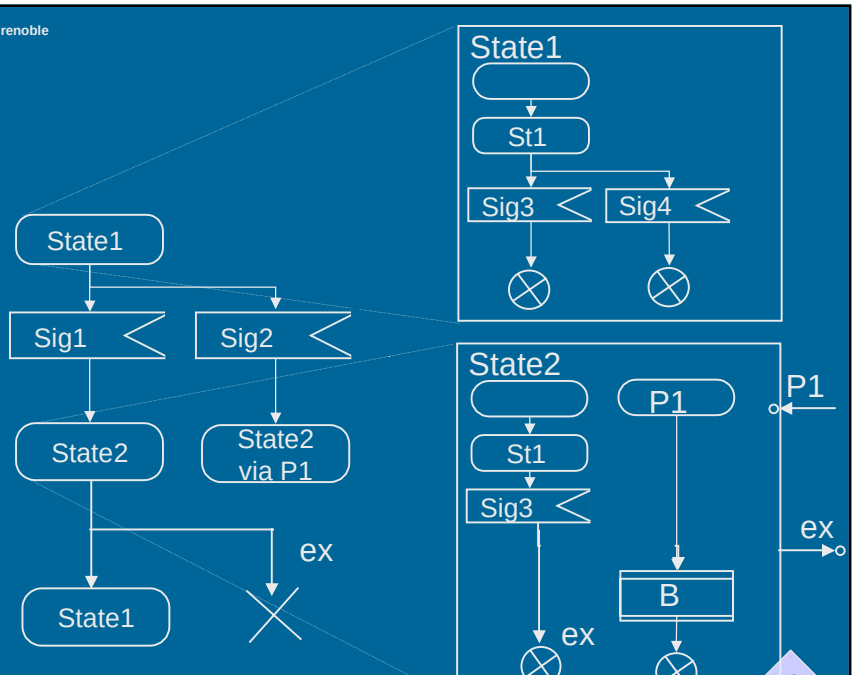
- an agent can make its procedures available for other agents
  - remote procedures
  - realized by two-way communication between caller and server
- after a call to a remote procedure the caller is blocked until he receives the procedure return from the server

- remote procedure call may deadlock
  - can be prevented by an associated timer, which raises an exception
- server accepts calls for remote procedures in any state
  - execution may be deferred by *save*
  - execution may be rejected by *input <p> raise <deny>*
- exceptions raised by the remote procedure are raised at client and server side

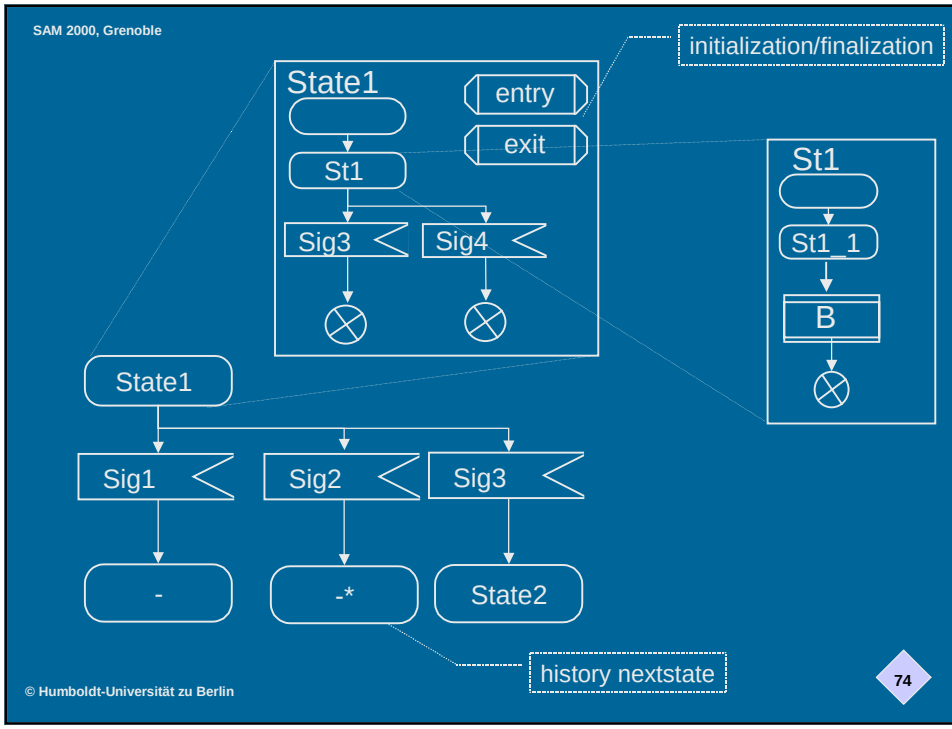


Composite States

- composite states are a means to hierarchically structure state machines
  - nesting of states
  - agent can be in more than one state at a time
  - Harel´s state charts
- composite state is itself a sub-state machine
- state machine of an agent is in fact a top-level composite state



- composite states share agent´s input queue
- internal transitions with the same trigger as external transitions have higher priority
- excactly one transition is executed
  - possibly concatenated with triggerless transitions
- special procedures may be used to define initialisation and finalisation
  - called implicitly upon entering/leaving a composite state

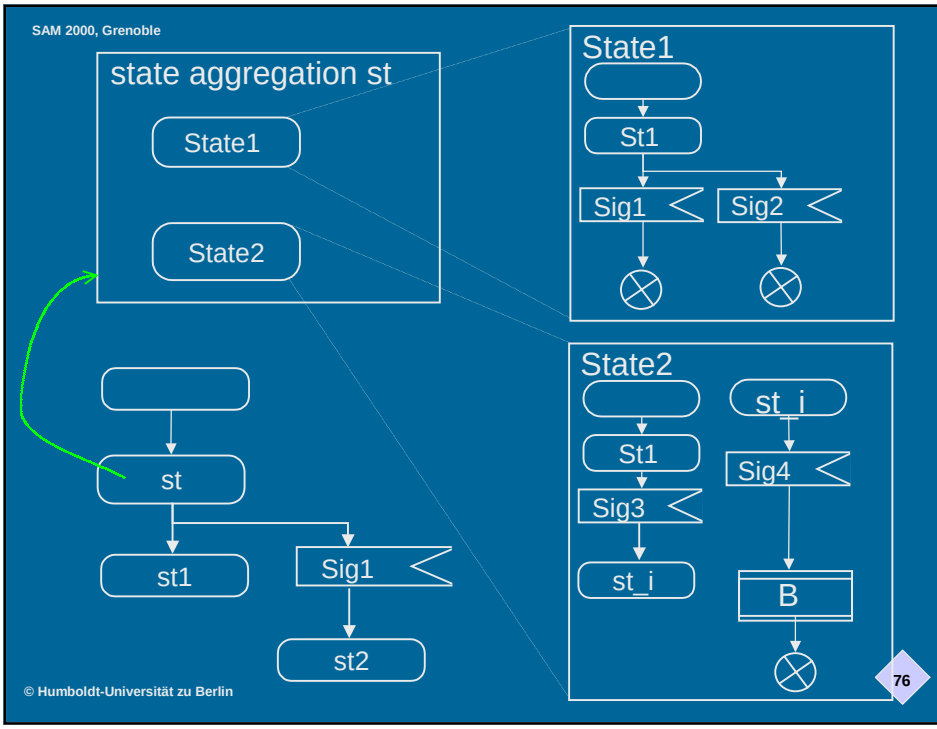


SAM 2000, Grenoble

State Aggregation

- state aggregations partition the state space of an agents state machine
- each partition handles a different set of the input stimuli
- excactly one partition is executing a transition at any point in time
  - multiple enabled transitions are executed in an interleaved manner

SAM 2000, Grenoble



- composite states and state aggregations can be classified
  - composite state type definition
  - typebased composite states/state aggregations
- concept & notation similar to agent types
- instances are static
  - live&die with containing agent
  - multiple instances in the same scope must have different names

Virtual Behaviour Elements

- allow the redefinition or replacement of behaviour elements in a type specialisation
- redefinition and finalisation similar to structural elements
- available for
  - procedures
  - transitions
  - exception handle transitions

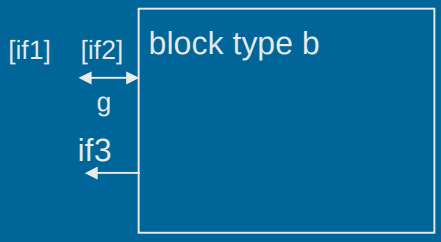
Interface

- pure typing concept used for typed communication between agents
- interface definition groups and names a set of
  - remote variable
  - remote procedure
  - signal definitions
- gates and channels paths can be typed by interfaces

- contained remote procedures & variables and signals are defined and visble where interface is visible
- interface can use also existing definitions for such elements
- multiple inheritance is available for interfaces
- interfaces are used as matching constraints for implicit channels

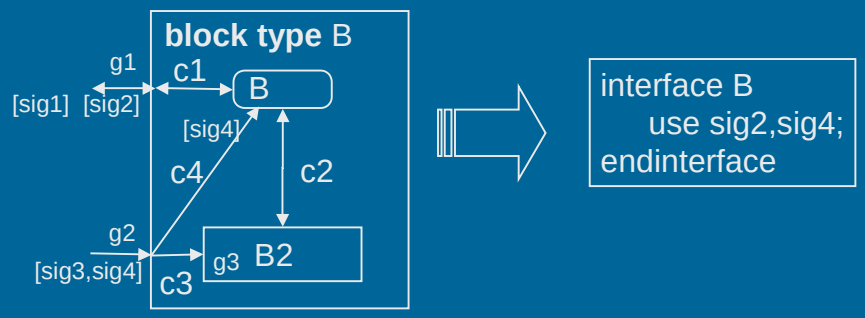
```
interface if1;
    signal sig1;
    procedure P;
    dcl I Natural;
endinterface;

signal sig2,sig3;
interface if2;
    use sig2, sig3;
endinterface;
```

```
interface if3
    inherits if1,if2;
```

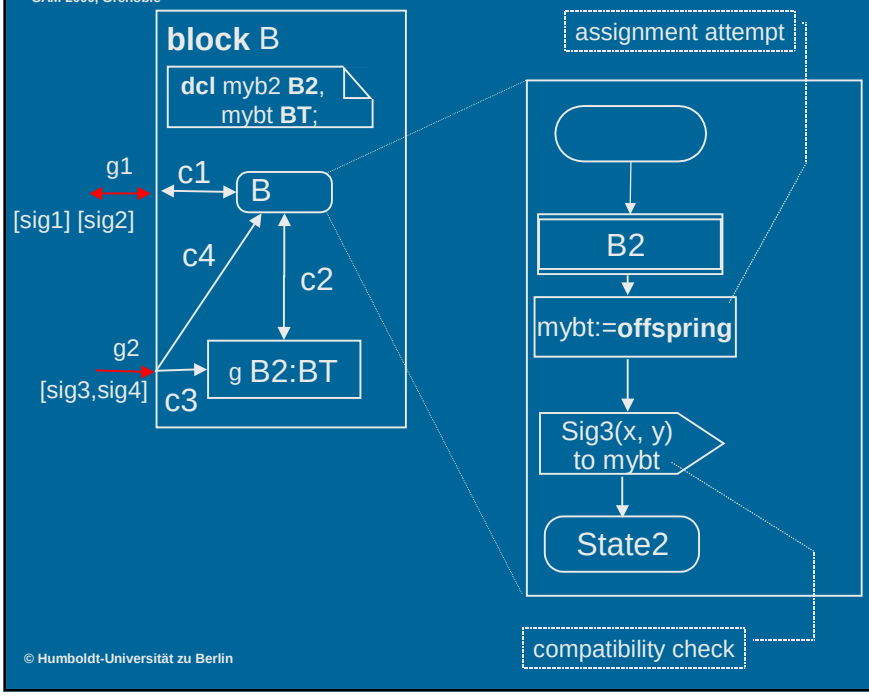


Agent Implicit Interface

- each agent and agent type introduces an implicit interface
  - same name as agent (type)
- contains all
  - signals accepted by the agents state machine
  - remote variables/procedures provided by agents state machine
- inherits all interfaces on gates connected to agents state machine



- each (explicit or implicit) interface implies a specialization of the Pid-type
- can be used to refer to agent instances in
  - variables, expressions
  - output, imports or remote procedure calls
  - assignment attempt
- dynamic type check tests
  - provision of remote variables/procedures
  - acceptance of signals
  - may raise an exception (InvalidReference)



Data Types

- two main kinds of data types
  - value types
  - object types (references)

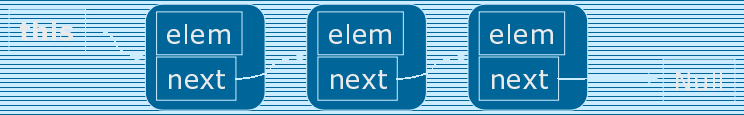
  conversion possible
- data type definition specifies
  - data elements and structure
  - operators and methods for data manipulation
- package *Predefined* contains a set of general data type definitions

- value types correspond to the newtype concept of SDL-92
  - axioms and generators have been removed
  - behaviour of operators defined by algorithms or transition actions (functional description)
  - constructors:
    - literals
    - structs
    - choices

- object types define references to values
  - references are local to agents
- definition similar to value types
  - conversion possible
- polymorphic assignments

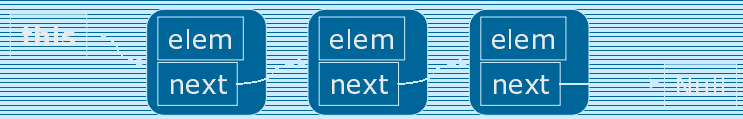
- further properties of data types:
  - inheritance
  - context parameters
  - methods allow operation-calls in programming-language like dot-notation
  - local data types, constants, exceptions
  - visibility of data elements can be controlled


```
object type List <type Elem>;
    struct
        elem            Elem;
        private next    List;
    operators Make(Elem)->List;
    methods   add(Elem)-> List;
```




```
endobject type;
```

```
object type List <type Elem>;
```




```
operator  Make(e Elem) {  return (. e,Null .); }
method    add(e Elem) {
    dcl last Elem;
    for(last=this,last.next/=Null,last.next);
    last.next := (. e .);
}
endobject type;
```

```
object type NatList inherits List<Natural>;
dcl myList NatList;
myList:=Make(5); myList.add(2);
```


Package Predefined

simple types

- Boolean, Integer, Natural, Real, Character, Duration, Time
- Charstring, Bit, Bitstring,
- Octet, Octetstring
- Time, Duration, Pid

data templates

- String, Powerset, Bag,
- Array, Vector

exceptions

- OutOfRange, InvalidReference,
- NoMatchingAnswer, UndefinedVariable,
- UndefinedField, InvalidIndex,
- DivisionByZero, Empty

Data assignments

value types

- strong type check
- assignment restrictions for specialised types

- object creation

object types

- references local to agent
- polymorphic assignments for specialised types
- virtual methods
- assignment attempts

- value extraction


```
value type Charstring
... /* predefined */
endvalue type
```

```
dcl val_var Charstring,
    ref_var object Charstring;
```

`ref_var := Make('Hello world')`	creates a reference to an object
`val_var := ref_var`	value extraction
`ref_var := val_var`	creates a refererence and copies the value into the object (clone)
`val_var := 'foo'`	value assignment
`ref_var:=Null, val_var := ref_var`	exception: InvalidReference

```
object type MyStruct;
struct   n Natural;
         y Boolean optional;
endobject type;

object type MyStruct1
inherits MyStruct adding
         c Character;
endobject type;

object type MyStruct2
inherits MyStruct adding
         s Charstring
endobject type;
```

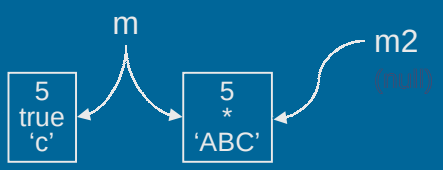
```
dcl   m MyStruct,
      m1 MyStruct1,
      m2 MyStruct2;
```

```
m1:= (. 5, true,'c');
m2:= (. 5,,'ABC');
m:= m1;
m:=m2; m2:=Null;
m1:=m;
m2:=m;
m2:=m1;
```

assignment

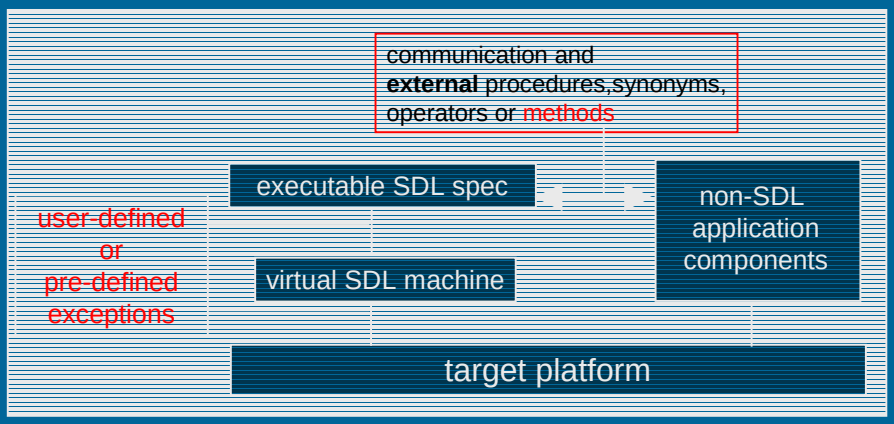
assignment attempt

static error



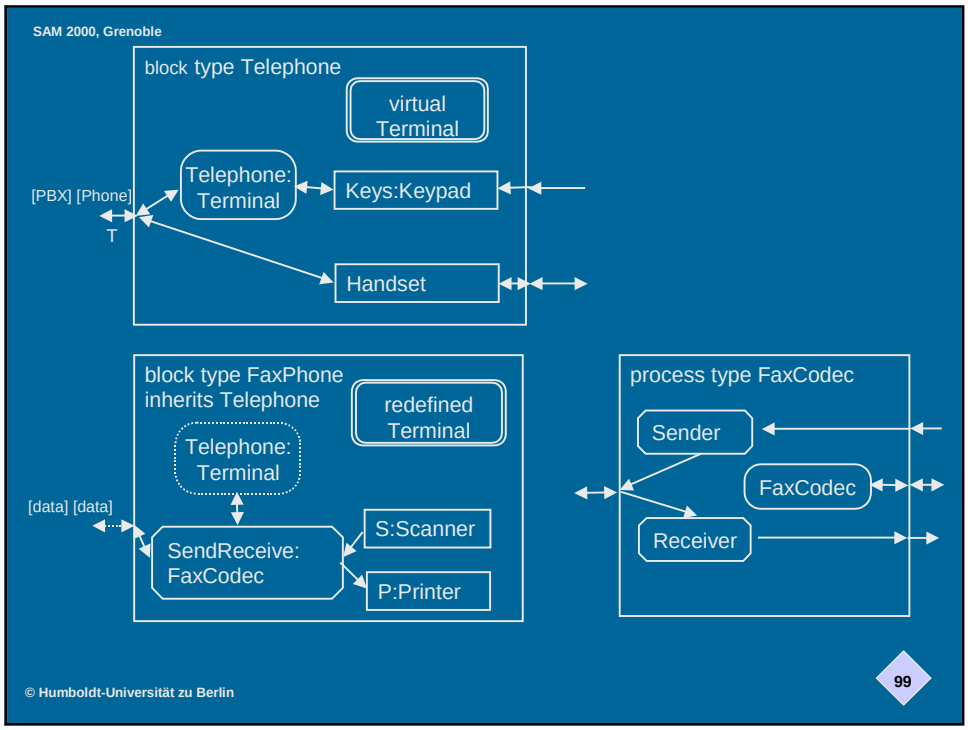
- each data type (obejct or value) implicitly inherits from ***Any*:**
  - operators *equal* and *clone*
  - methods *copy* and *is_equal*
- each object type has additionally the operations
  - *Make* and *Null*
- each struct type has for each file the methods
  - *<field>Modify, <field>Extract,*
  - *<field>Present* (optional fields only)

Development of Real-Time Systems with SDL



Further Information

- tutorial slides and author contact
  www.informatik.hu-berlin.de/Institut/struktur/systemanalyse/
  {fischer|holz}@informatik.hu-berlin.de
- ITU standards and recommendations
  - www.itu.ch or www.itu.int/itudoc/itu-t/approved/z/index.html
- SDL Forum Society
  - www.sdl-forum.org
- conferences and workshops
  - bi-annual SDL Forum - next Copenhagen 2001


SAM 2000, Grenoble
block type Telephone
virtual Terminal
Telephone: Terminal
Keys:Keypad
[PBX] [Phone]
T
Handset
block type FaxPhone inherits Telephone
redefined Terminal
Telephone: Terminal
[data] [data]
SendReceive: FaxCodec
S:Scanner
P:Printer
process type FaxCodec
Sender
FaxCodec
Receiver
© Humboldt-Universität zu Berlin
99