MODELING DAN SPESIFIKASI SISTEM TERTANAM

ERI PRASETYO
UNIVERSITAS GUNADARMA
Embedded Systems Design Flow

1. **Concept**
   - Hardware Components
   - Software Components

2. **Specification**
   - HW/SW Partitioning

3. **Estimation - Exploration**
   - (Synthesis, Layout, …)

4. **Design**
   - (Compilation, …)

5. **Validation and Evaluation (area, power, performance, …)**

6. **Software**
7. **Hardware**
Introduction

• Describing embedded system’s processing behavior
  – Can be extremely difficult
  • Complexity increasing with increasing IC capacity
    – Past: washing machines, small games, etc.
      » Hundreds of lines of code
    – Today: TV set-top boxes, Cell phone, etc.
      » Millions of lines of code
  • Desired behavior often not fully understood in beginning
    – Many implementation bugs due to description mistakes/omissions
  – English (or other natural language) common starting point
    • Precise description difficult to impossible
    • Example: Motor Vehicle Code – thousands of pages long...
Models and languages

• How can we (precisely) capture behavior?
  – We may think of languages (C, C++), but *computation model* is the key

• Common computation models:
  – Sequential program model
    • Statements, rules for composing statements, semantics for executing them
  – Communicating process model
    • Multiple sequential programs running concurrently
  – State machine model
    • For control dominated systems, monitors control inputs, sets control outputs
  – Dataflow model
    • For data dominated systems, transforms input data streams into output streams
  – Object-oriented model
    • For breaking complex software into simpler, well-defined pieces
Models vs. Specification Languages

• Computation models describe system behavior
  – Conceptual notion, e.g., recipe, sequential program
• Languages capture models
  – Concrete form, e.g., English, C
• Variety of languages can capture one model
  – E.g., sequential program model → C, C++, Java
• One language can capture variety of models
  – E.g., C++ → sequential program model, object-oriented model, state machine model
• Certain languages better at capturing certain computation models
Text versus Graphics

• Models versus languages not to be confused with text versus graphics
  – Text and graphics are just two types of languages
    • Text: letters, numbers
    • Graphics: circles, arrows (plus some letters, numbers)

X = 1;
Y = X + 1;
MODEL COMPUTATION
Introductory example: An elevator controller

- Simple elevator controller
  - *Request Resolver* resolves various floor requests into single requested floor
  - *Unit Control* moves elevator to this requested floor

- Try capturing in C...

Partial English description

“Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don’t change directions unless there are no higher requests when moving up or no lower requests when moving down…”
Elevator controller using a sequential program model

**Sequential program model**

```c
void UnitControl()
{
    up = down = 0; open = 1;
    while (1)
    {
        while (req == floor); open = 0;
        if (req > floor) { up = 1; }
        else { down = 1; }
        while (req != floor);
        up = down = 0;
        open = 1;
        delay(10);
    }
}
```

```c
void RequestResolver()
{
    while (1)
    
    req = ...
    ...
}
```

```c
void main()
{
    Call concurrently: UnitControl() and RequestResolver()
}
```

**Partial English description**

“Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don’t change directions unless there are no higher requests when moving up or no lower requests when moving down…”

**System interface**

- Inputs: int floor; bit b1..bN; up1..upN-1; dn2..dnN;
- Outputs: bit up, down, open;
- Global variables: int req;

You might have come up with something having even more if statements.
Finite-state machine (FSM) model

• Trying to capture this behavior as sequential program is a bit awkward
• Instead, we might consider an FSM model, describing the system as:
  – Possible states
    • E.g., *Idle, GoingUp, GoingDn, DoorOpen*
  – Possible transitions from one state to another based on input
    • E.g., req > floor
  – Actions that occur in each state
    • E.g., In the *GoingUp* state, \( u,d,o,t = 1,0,0,0 \) (up = 1, down, open, and timer_start = 0)

• Try it... (for Unit Control)

**Partial English description**

“Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don’t change directions unless there are no higher requests when moving up or no lower requests when moving down…”

**System interface**

- Unit Control
- Request Resolver
- up
- down
- open
- floor
- req
- up1
- up2
- up3
- upN
- dn1
- dn2
- dn3
- dnN
- b1
- b2
- bN
- buttons inside elevator
- up/down buttons on each floor
- ...
Finite-state machine (FSM) model

UnitControl process using a state machine

\[ \begin{align*}
\text{GoingUp} & : \text{req} > \text{floor} \\
\text{Idle} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor} \\
\text{DoorOpen} & : \text{timer} < 10
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor} \\
\text{GoingUp} & : \text{req} > \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

\[ \begin{align*}
\text{DoorOpen} & : \text{timer} < 10 \\
\text{GoingUp} & : \text{req} == \text{floor} \\
\text{GoingDn} & : \text{req} < \text{floor}
\end{align*} \]

u, d, o, t = 1, 0, 0, 0

\[ u, d, o, t = 0, 0, 1, 0 \]

\[ u, d, o, t = 0, 1, 0, 0 \]

u is up, d is down, o is open, t is timer_start
Formal definition

• An FSM is a 6-tuple $F<S, I, O, F, H, s_0>$
  – $S$ is a set of all states \{s_0, s_1, …, s_l\}
  – $I$ is a set of inputs \{i_0, i_1, …, i_m\}
  – $O$ is a set of outputs \{o_0, o_1, …, o_n\}
  – $F$ is a next-state function ($S \times I \rightarrow S$)
  – $H$ is an output function ($S \rightarrow O$)
  – $s_0$ is an initial state

• Moore-type
  – Associates outputs with states ($H$ maps $S \rightarrow O$)

• Mealy-type
  – Associates outputs with transitions ($H$ maps $S \times I \rightarrow O$)

• Shorthand notations to simplify descriptions
  – Implicitly assign 0 to all unassigned outputs in a state
  – Implicitly AND every transition condition with clock edge (FSM is synchronous)
Finite-state machine with datapath model (FSMD)

- FSMD extends FSM: complex data types and variables for storing data
  - FSMs use only Boolean data types and operations, no variables
- FSMD: 7-tuple $<S, I, O, V, F, H, s_0>$
  - $S$ is a set of states $\{s_0, s_1, \ldots, s_l\}$
  - $I$ is a set of inputs $\{i_0, i_1, \ldots, i_m\}$
  - $O$ is a set of outputs $\{o_0, o_1, \ldots, o_n\}$
  - $V$ is a set of variables $\{v_0, v_1, \ldots, v_n\}$
  - $F$ is a next-state function ($S \times I \times V \rightarrow S$)
  - $H$ is an action function ($S \rightarrow O + V$)
  - $s_0$ is an initial state

- $I, O, V$ may represent complex data types (i.e., integers, floating point, etc.)
- $F, H$ may include arithmetic operations
- $H$ is an action function, not just an output function
  - Describes variable updates as well as outputs
- Complete system state now consists of current state, $s_i$, and values of all variables

We described UnitControl as an FSMD
Describing a system as a state machine

1. List all possible states
2. Declare all variables
3. For each state, list possible transitions, with conditions, to other states
4. For each state and/or transition, list associated actions
5. For each state, ensure exclusive and complete exiting transition conditions
   - No two exiting conditions can be true at same time
     - Otherwise nondeterministic state machine
   - One condition must be true at any given time
     - Reducing explicit transitions should be avoided when first learning
State machine vs. sequential program model

• Different thought process used with each model
• State machine:
  – Encourages designer to think of all possible states and transitions among states based on all possible input conditions
• Sequential program model:
  – Designed to transform data through series of instructions that may be iterated and conditionally executed
• State machine description excels in many cases
  – More natural means of computing in those cases
  – *Not* due to graphical representation (state diagram)
    • Would still have same benefits if textual language used (i.e., state table)
    • Besides, sequential program model could use graphical representation (i.e., flowchart)
Capturing state machines in sequential programming language

• Despite benefits of state machine model, most popular development tools use sequential programming language
  – C, C++, Java, Ada, VHDL, Verilog, etc.
  – Development tools are complex and expensive, therefore not easy to adapt or replace
    • Must protect investment

• Two approaches to capturing state machine model with sequential programming language
  – Front-end tool approach
    • Additional tool installed to support state machine language
      – Graphical and/or textual state machine languages
      – May support graphical simulation
      – Automatically generate code in sequential programming language that is input to main development tool
        • Drawback: must support additional tool (licensing costs, upgrades, training, etc.)
  – Language subset approach
    • Most common approach...
Language subset approach

- Follow rules (template) for capturing state machine constructs in equivalent sequential language constructs
- Used with software (e.g., C) and hardware languages (e.g., VHDL)
- Capturing \textit{UnitControl} state machine in C
  - Enumerate all states (#define)
  - Declare state variable initialized to initial state (IDLE)
  - Single switch statement branches to current state’s case
  - Each case has actions
    - up, down, open, timer\_start
  - Each case checks transition conditions to determine next state
    - if(...) \{state = ...;\}

\begin{verbatim}
#define IDLE 0
#define GOINGUP 1
#define GOINGDN 2
#define DOOROPEN 3
void UnitControl() {
    int state = IDLE;
    while (1) {
        switch (state) {
            case IDLE:
                up=0; down=0; open=1; timer\_start=0;
                if (req==floor) {state = IDLE;}
                if (req > floor) {state = GOINGUP;}
                if (req < floor) {state = GOINGDN;}
                break;
            case GOINGUP:
                up=1; down=0; open=0; timer\_start=0;
                if (req > floor) {state = GOINGUP;}
                if (!(req>floor)) {state = DOOROPEN;}
                break;
            case GOINGDN:
                up=1; down=0; open=0; timer\_start=0;
                if (req < floor) {state = GOINGDN;}
                if (!(req<floor)) {state = DOOROPEN;}
                break;
            case DOOROPEN:
                up=0; down=0; open=1; timer\_start=1;
                if (timer < 10) {state = DOOROPEN;}
                if (!(timer<10)) {state = IDLE;}
                break;
        }
    }
}
\end{verbatim}

\textit{UnitControl} state machine in sequential programming language
# General template

```c
#define S0 0
#define S1 1
...
#define SN N

void StateMachine() {
    int state = S0; // or whatever is the initial state.
    while (1) {
        switch (state) {
            case S0:
                // Insert S0's actions here & Insert transitions T_i leaving S0:
                if( T_0's condition is true ) { state = T_0's next state; /*actions*/ }
                if( T_1's condition is true ) { state = T_1's next state; /*actions*/ }
                ...
                if( T_m's condition is true ) { state = T_m's next state; /*actions*/ }
                break;
            case S1:
                // Insert S1's actions here
                // Insert transitions T_i leaving S1
                break;
            ...
            case SN:
                // Insert SN's actions here
                // Insert transitions T_i leaving SN
                break;
        }
    }
}
```
HCFSM

- Hierarchical/concurrent state machine model (HCFSM)
  - Extension to state machine model to support hierarchy and concurrency
  - States can be decomposed into another state machine
    - Humans not capable of understanding systems containing more than ~5 objects.
    - *With hierarchy* has identical functionality as *Without hierarchy*, but has one less transition (z)
    - Known as OR-decomposition
    - States can execute concurrently
    - Known as AND-decomposition
UnitControl with FireMode

- FireMode
  - When fire is true, move elevator to 1st floor and open door
    - w/o hierarchy: Getting messy!
    - w/ hierarchy: Simple!

With hierarchy

With concurrent RequestResolver
Program-state machine model (PSM): HCFSM plus sequential program model

- Program-state’s actions can be FSM or sequential program
  - Designer can choose most appropriate
- Stricter hierarchy than HCFSM
  - transition between sibling states only, single entry
  - Program-state may “complete”
    - Reaches end of sequential program code, OR
    - FSM transition to special complete substate
- PSM has 2 types of transitions
  - Transition-immediately (TI): taken regardless of source program-state
  - Transition-on-completion (TOC): taken only if condition is true AND source program-state is complete

```
up = down = 0; open = 1;
while (1) {
  while (req == floor);
  open = 0;
  if (req > floor) { up = 1;}
  else {down = 1;}
  while (req != floor);
  open = 1;
  delay(10);
}
```

```
up = 0; down = 1; open = 0;
while (floor > 1);
```

```
fire
```

```
!fire
```

```
NormalMode
```

```
FireMode
```

```
UnitControl
```

```
ElevatorController
```

```
RequestResolver
```

```
int req;
```

- NormalMode and FireMode described as sequential programs
- Black square originating within FireMode indicates !fire is a TOC transition
  - Transition from FireMode to NormalMode only after FireMode completed
Concurrent Process Model

• Describes functionality of system in terms of two or more concurrently executing subtasks
• Many systems easier to describe with concurrent process model because inherently multitasking
  
  E.g., simple example:
  – Read two numbers X and Y
  – Display “Hello world.” every X seconds
  – Display “How are you?” every Y seconds

• Try it … assume a delay() function in C to advance time

• More effort would be required with sequential program or state machine model

Simple concurrent process example

```
ConcurrentProcessExample() {
    x = ReadX()
    y = ReadY()
    Call concurrently:
        PrintHelloWorld(x) and PrintHowAreYou(y)
}
PrintHelloWorld(x) {
    while( 1 ) {
        print "Hello world."
        delay(x);
    }
}
PrintHowAreYou(y) {
    while( 1 ) {
        print "How are you?"
        delay(y);
    }
}
```

Enter X: 1
Enter Y: 2
Hello world.  (Time = 1 s)
Hello world.  (Time = 2 s)
How are you?   (Time = 2 s)
Hello world.  (Time = 3 s)
How are you?   (Time = 4 s)
Hello world.  (Time = 4 s)
...
Dataflow Model

• Definition: Data flow modeling is … “the process of identifying, modeling and documenting how data moves around an information system.

Example: Video on on demand system
Asynchronous Dataflow Model

- Derivative of concurrent process model
- Nodes represent transformations
  - May execute concurrently
- Edges represent flow of tokens (data) from one node to another
  - May or may not have token at any given time
- When all of node’s input edges have at least one token, node may fire
- When node fires, it consumes input tokens processes transformation and generates output token
- Nodes may fire simultaneously
- Several commercial tools support graphical languages for capture of dataflow model
  - Can automatically translate to concurrent process model for implementation
  - Each node becomes a process

Z = (A + B) * (C - D)
Kahn Process Network Model

- Each component is a program/task/process, not an FSM
- Communication is by FIFOs; no overflow considered
  - writes never have to wait,
  - reads wait if FIFO is empty.
- Only 1 sender and 1 receiver per FIFO
- Channels transmit information within an unpredictable but finite amount of time
- In general, execution times are unknown

KPNs are

- Determinate
  - for a given input, the result will always be the same, regardless of the speed of the nodes
- Turing complete
  - anything which can be computed can be computed by a KPN
Synchronous Dataflow Model

- With digital signal-processors (DSPs), data flows at fixed rate
- Multiple tokens consumed and produced per firing
- Synchronous dataflow model takes advantage of this
  - Each edge labeled with number of tokens consumed/produced each firing
  - Global clock controls firing rate
Parallel Scheduling of SDF Models

SDF is suitable for automated mapping onto parallel processors and synthesis of parallel circuits.

Many scheduling optimization problems can be formulated. Some can be solved, too!

Source: ptolemy.eecs.berkeley.edu/presentations/03/streamingEAL.ppt
Other Models Of Computation

• Differential Equation Model
  – Capable of modeling analog circuits and physical systems
  \[
  \frac{\partial^2 x}{\partial t^2} = b
  \]

• Discrete Event Model
  – Consists of events carrying a totally ordered time stamp, indicating the time at which the event occurs
  – Sequential specification languages such as VHDL, Verilog, SystemC, SpecC, Simulink, etc implement this model
Tools: SCADE

“SCADE Suite, including the SCADE KCG Qualified Code Generator, is used by AIRBUS and many of its main suppliers for the development of most of the A380 and A400M critical on board software, and for the A340-500/600 Secondary Flying Command System, aircraft in operational use since August 2002.”

François Pilarski, Systems Engineering Framework - Senior Manager Engineering, Systems & Integration Tests; Airbus France.

Source: http://www.esterel-technologies.com/products/scade-suite/
Tools: Simulink

**Semantics?** “Simulink uses an idealized timing model for block execution and communication. Both happen infinitely fast at exact points in simulated time. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps.” [Nicolae Marian, Yue Ma]

~synchronous dataflow MoC

[Image of Simulink diagram]

Code Generation

- Code automatically generated from description

```c
/* Switch: '<Root>/SW2' incorporates:
 * Sum: '<Root>/Sum1'
 * Gain: '<Root>/G1'
 * Sum: '<Root>/Sum2'
 * Gain: '<Root>/G3'
*/
for(i1=0; i1<10; i1++) {
    if(rtU.In1[i1] * 3.0 >= 0.0) {
        rtb_SW2_c[i1] = rtU.In1[i1] - rtDWork.Delay_DSTATE[i1];
    } else {
        rtb_SW2_c[i1] = (rtDWork.Delay_DSTATE[i1] - rtU.In1[i1]) * 5.0;
    }
}

/* Outport: '<Root>/Out1' */
rtY.Out1[i1] = rtb_SW2_c[i1];

/* Update for UnitDelay: '<Root>/Delay' */
rtDWork.Delay_DSTATE[i1] = rtb_SW2_c[i1];
```
Concurrenty

- Convenient way of describing concurrency.
- AND-super-states: FSM is in all (immediate) sub-states of a super-state.

![State Machine Diagram](image_url)

**Answering-Machine**

- **Line-Monitoring**
  - **Lwait**
  - **Lproc**
    - **Hang-up** (Caller)
    - **Ring**

- **Key-Monitoring (Excluding On/Off)**
  - **Kwait**
  - **Kproc**
    - **Key-Pressed**
    - **Done**

- **On/Off**
  - **Key-On**
  - **Key-Off**
  - **Off**
Line-monitoring and key-monitoring are entered and left, when service switch is operated.
Timers

- Timer needs to be modeled in embedded systems.
- StateCharts uses special edges for timeouts.

If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.
Timers in Answering Machine

Lproc

4 s

timeout

play text

beep

lift off

talk

return (callee)

dead

8 s record

timeout

silent

beep
General Form of Edge Labels

- Events:
  - Exist only until the next evaluation of the model
  - Can be either internally or externally generated

- Conditions:
  - Refer to test of values of variables or a test of the current state of the system

- Reactions:
  - Can either be assignments for variables or creation of events

- Example:
  - service-off [not in Lproc] / service:=0
StateCharts: Simulation Phases

• How are edge labels evaluated?

• Three phases:
  1. Effect of external changes on events and conditions is evaluated,
  2. The set of transitions to be made in the current step and right hand sides of assignments are computed,
  3. Transitions become effective, variables obtain new values.

• Separation into phases 2 and 3 guarantees deterministic and reproducible behavior.


- In a single phase environment, executing the left state first would assign the old value of $b (=0)$ to $a$ and $b$. Executing the right state first would assign the old value of $a (=1)$ to $a$ and $b$. The execution would be non-deterministic.

- In phase 2, variables $a$ and $b$ are assigned to temporary variables. In phase 3, these are assigned to $a$ and $b$. As a result, variables $a$ and $b$ are swapped.
Reflects Model of Clocked Hardware

• In an actual clocked (synchronous) hardware system, both registers should be swapped deterministically
  – Designers like determinism – a real system would not work if it behaved unpredictably!

• Same separation into phases found in other languages as well, especially those that are intended to model hardware.

• Some specification languages for hierarchical state machines (UML, …) do not include the three simulation phases.
  – correspond more to a SW point of view with no synchronous clocks
  – LabView allows turning the multi-phased simulation on and off.
Steps

• Execution of a StateCharts model consists of a sequence of (status, step) pairs
  – Status = values of all variables + set of events + current time
  – Step = execution of the three phases

\[
\begin{align*}
\text{Status} & \quad \quad \text{Step} \\
\quad & \quad \quad \quad \\
\quad & \quad \quad \quad \\
\quad & \quad \quad \quad \\
\quad & \quad \quad \quad \\
\end{align*}
\]
Broadcast Mechanism

• Values of variables are visible to all parts of the StateCharts model.

• New values become effective in phase 3 of the current step and are obtained by all parts of the model in the following step.

• StateCharts implicitly assumes a broadcast mechanism for variables.

• StateCharts is appropriate for local control systems (w/ shared memory communication), but not for distributed applications for which updating variables may take some time.
Evaluation of StateCharts

• Pros:
  – Hierarchy allows nesting of AND/OR-superstates.
  – Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
  – Back-end tools translate StateCharts into C or VHDL – enabling hw/sw implementations

• Cons:
  – Generated C programs frequently inefficient,
  – Not useful for distributed applications,
  – No program constructs,
  – No object-oriented support
SDL

- Specification and Description Language
  - Designed for specification of distributed systems.
- Defined by International Telecommunication Union (ITU): Z.100 recommendation in 1980
- Provides textual and graphical formats.
- Like StateCharts, it is based on HCFSM model of computation; each FSM is called a process,
- It uses message passing for communications, and supports operations on data.
  - Better for capturing distributed systems
FSMs/Processes in SDL
Operations on Data

- Variables can be declared locally for processes.
- Their type can be predefined or defined in SDL.
- SDL supports abstract data types (ADTs).
Communication in SDL

- Communication between FSMs (processes) is based on message-passing, assuming a potentially indefinitely large FIFO-queue.
  - Each process fetches next entry from FIFO,
  - Checks if input enables transition,
    - if yes: transition takes place,
    - if no: input is ignored (exception: SAVE-mechanism).
Deterministic?

Let tokens be arriving at FIFO at the same time:
Order in which they are stored, is unknown:

All orders are legal: Simulators can show different
behaviors for the same input, all of which are correct.
Process Interaction Diagrams

• Interaction between processes can be described in process interaction diagram
  – Special case of block diagrams.
• In addition to processes, the diagrams contain channels and declarations of local signals.
Hierarchy in SDL

- Process interaction diagrams can be included in blocks. The root block is called system.

Processes cannot contain other processes, unlike in StateCharts.
Timers can be declared locally. Elapsed timers put signal into queue (not necessarily processed immediately).

- RESET removes timer (also from FIFO-queue).
Description of Network Protocols
Vending Machine

Machine selling pretzels, (potato) chips, cookies, and doughnuts:
accepts nickels, dime, quarters, and half-dollar coins.

Not a distributed application.

System VendingMachine

Ccoins
[nickel, dime, quarter, half] → CoinInterface

Cadd
[add] → CointCtrl
[reject_further_coins, accept_coins]

Creject
[reject_coin]

CamontDisplay
[amount_entered]

CemptyDisplay
[pretzel_empty, chip_empty, cookie_empty, doughnut_empty]

CspitPurchased
[spit_pretzel, spit_chip, spit_cookie, spit_doughnut]

Cchange
[spit_change] → CchangeInterface

CexaktDisplay
[exact_only]

CspitChange
[spit_nickname, spit_dime]

SIGNAL
[dime, nickel, quarter, half, pur_pretzel, pur_cookie, pur_doughnut, pur_chip, add(int), spit_change(int), amount_entered(int), reject_further_coins, exact_only, accept_coins, reject_coins, spit_dime, spit_nickel, pretzel_empty, spit_pretzel, chip_empty, spit_chip, cookie_empty, spit_cookie, doughnut_empty, spit_doughnut, reload_pretzel, reload_chip, reload_cookie, reload_doughnut]

SYNTYPE items=INTEGER
CONSTANTS 0:7
ENDSYNTYPE items;

SYNTYPE int=INTEGER
CONSTANTS 0:127
ENDSYNTYPE int;
Decode Requests

Block DecodeRequests

Radd [add] → AmountHandler → Rchange [spit_change]
Rcointctrl [accept_coins, reject_further_coins]
RamountDisplay [amount_entered]

Rpretzel [pur_pretzel, reload_pretzel] → PretzelHandler → Rpretzel_e [pretzel_empty]
Rchips [pur_chip, reload_chip] → ChipHandler → Rchips [spit_chip]
Rchip_e [chip_empty]

Rcookie [pur_cookie, reload_cookie] → CookieHandler → Rcookie_e [cookie_empty]
Rcookie_s [spit_cookie]

Rdoughnut [pur_doughnut, reload_doughnut] → DoughnutHandler → Rdoughnut_e [doughnut_empty]
Rdoughnut_s [spit_doughnut]

CONNECT Cadd AND Radd;
CONNECT Ccoinctrl AND Rcointctrl;
CONNECT Cchange AND Rchange;
CONNECT CAmountDisplay AND RamountDisplay;
CONNECT Crequest AND Rpretzel, Rchip, Rcookie,
Rdoughnut;
CONNECT CemptyDisplay AND Rpretzel_e, Rchip_e,
Rcookie_e, Rdoughnut_e;
CONNECT CspitPurchased AND Rpretzel_s,
Rchip_s, Rcookie_s, Rdoughnut_s;

SYNONYM PRETZEL int=50
SYNONYM PCHIP int=15;
SYNONYM PCOOKIE int=55;
SYNONYM PDOUNIGHT int=60;
SYNONYM PMAX int=60;
SYNONYM NITEMS items=7;

SIGNAL sub(int);
ChipHandler

DCL nchip items:=NITEMS;

VIEWED current int;

Process ChipHandler

pur_wait

pur_chip

VIEW(current) >= PCHIP

ja

sub(PCHIP)

nchip:= nchip-1;

spit_chip

nchip=0

ja

nein

pur_wait

chip_empty

empty

reload_chip

nchip:=NITEMS

pur_wait
SDL: Summary

• Excellent for distributed applications
  – Used to specify ISDN
• Commercial tools available from SINTEF, Telelogic, Cinderella (//www.cinderella.dk).
• Not necessarily deterministic
  – Order in which FSMs are reading input is unknown
  – Not a synchronous language
• Timer concept adequate for soft deadlines,
• Limited way of using hierarchies,
• Limited programming language support,
• No description of non-functional properties.
• Becoming less popular
Discrete Event Semantics

Basic discrete event (DE) semantics

- Queue of future actions, sorted by time
- Loop:
  - Fetch next entry from queue
  - Perform function as listed in entry
    - May include generation of new entries
- Until termination criterion = true

<table>
<thead>
<tr>
<th>queue</th>
<th>time</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 10 13 15 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a:=5 b:=7 c:=8 a:=6 a:=9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a 6</th>
<th>b 7</th>
<th>c 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
HDLs use discrete event (DE) semantics

- Used in hardware description languages (HDLs):
- Description of concurrency is a must for HW description languages!
  - Many HW components operate concurrently
  - Typically mapped to “processes“
  - These processes communicate via “signals“
- Examples:
  - MIMOLA [Zimmermann/Marwedel], ~1975
  - ...
  - VHDL (very prominent example in DE modeling)
    One of the 3 most important HDLs:
    VHDL, Verilog, SystemC
VHDL

• HDL = hardware description language
• Textual HDLs replaced graphical HDLs in the 1980s (better for complex behavior).
• VHDL = VHSIC hardware description language
• VHSIC = very high speed integrated circuit
• 1980: Definition started by DoD in 1980
• 1984: first version of the language defined, based on ADA, PASCAL
• 1987: IEEE standard 1076; 1992 revision;
• 1999: VHDL-AMS models analog
• 2006: Major extensions
Entities and Architectures

• Each design unit is called an entity.
• Entities are comprised of entity declarations and one or several architectures.

• Each architecture includes a model of the entity. By default, the most recently analyzed architecture is used. The use of another architecture can be requested in a configuration.
entity full_adder is
  port(a, b, carry_in: in Bit;  -- input ports
       sum, carry_out: out Bit);  -- output ports
end full_adder;
Architectures

architecture behavior of full_adder is
begin
    sum <= (a xor b) xor carry_in after 10 Ns;
    carry_out <= (a and b) or (a and carry_in) or
                  (b and carry_in) after 10 Ns;
end behavior;

Architectural bodies can be
- behavioral bodies or - structural bodies.
Bodies not referring to hardware components are called behavioral bodies.
Full Adder Simulation Output
architecture structure of full_adder is

component half_adder
  port (in1, in2: in Bit; carry: out Bit; sum: out Bit);
end component;

component or_gate
  port (in1, in2: in Bit; o: out Bit);
end component;

signal x, y, z: Bit; -- local signals
begin -- port map section
  i1: half_adder port map (a, b, x, y);
  i2: half_adder port map (y, carry_in, z, sum);
  i3: or_gate port map (x, z, carry_out);
end structure;
VHDL processes

Processes model parallelism in hardware.

General syntax:

\[\text{label:} \quad \text{--optional}\]

\text{process}
\quad \text{declarations} \quad \text{--optional}

\text{begin}
\quad \text{statements} \quad \text{--optional}

\text{end process}

\[a \leq b \text{ after } 10 \text{ ns is equivalent to}\]

\text{process}
\quad \text{begin}
\quad \quad a \leq b \text{ after } 10 \text{ ns}

\text{end process}\]
Transport Delay

signal <= transport expression after delay;

This corresponds to models for simple wires

Pulses will be propagated, no matter how short they are.

Example:

\[ c <= \text{transport } a \text{ or } b \text{ after } 10 \text{ ns}; \]
Inertial Delay

- By default, inertial delay is assumed.
  - Suppression of all “spikes” shorter than the delay
- Inertial delay models the behavior of gates.
  - To switch state, gates require that input pulse cross a certain threshold and remain unchanged for a certain period of time (hold time).
  - If the pulse is small, the gate will not change state.
  - The minimum pulse width for an input pulse to cause a change in state for a gate is called the inertial delay of the gate.

Example:
\[
\begin{align*}
c &\leq a \text{ or } b \text{ after } 10 \text{ ns}; \\
c &\leq \text{INERTIAL} \ a \text{ or } b \text{ after } 10 \text{ ns}; \\
c &\leq \text{REJECT} \ 10 \text{ ns} \ \text{INERTIAL} \ a \text{ or } b \text{ after } 10 \text{ ns};
\end{align*}
\]

No pulse of 5 ns
Delta Delay

• What about models where no propagation delays are specified?
• Infinitesimally small delay is automatically inserted by the simulator to preserve correct ordering of events
Wait statements

Four possible kinds of wait-statements:

- **wait on signal list;**
  - wait until signal changes;
  - Example: wait on a;
- **wait until condition;**
  - wait until condition is met;
  - Example: wait until c='1';
- **wait for duration;**
  - wait for specified amount of time;
  - Example: wait for 10 ns;
- **wait;**
  - suspend indefinitely
Sensitivity Lists

Sensitivity lists are a shorthand for a single `wait on`-statement at the end of the process body:

```
process (x, y)
  begin
  prod <= x and y ;
  end process;
```

is equivalent to

```
process
  begin
  prod <= x and y ;
  wait on x,y;
  end process;
```
VHDL: Summary

• Behavioral hierarchy (procedures & functions)
• Structural hierarchy but no nested processes,
• No object-orientation,
• Static number of processes,
• Complicated simulation semantics,
• Too low level for initial specification,
• Good for intermediate language for hardware generation.