Abstraction of Concurrency and Communication in VHDL

Peter J. Ashenden
University of Adelaide

Philip A. Wilsey
University of Cincinnati

This work was partially supported by Wright Laboratory
under USAF contract F33615-95-C-1638.

VHDL in the design flow

- At the system level, specify aspects of the system
  - structure (architectural partitioning)
  - behaviour (data transforms, reactions)
  - . . .
- VHDL allows description of structure and behaviour
- Prefer single language approach
  - system level to detailed design (hardware/software)
  - aids refinement
  - avoid semantic mismatch

Problem with VHDL

- Behaviour includes concurrency and communication
- Communication performed using signals
  - assignment, resolution, waits
- Signals do not express synchronization
  - *cf.* system description and programming languages
  - assignment schedules a message for a particular time
  - receiver may miss the message
    - if it does not respond at that time

Ad hoc solutions

- Communication using handshaking protocol
  - requires extra signals
  - requires protocol implementation
- Explicitly instantiated message queues
  - extra components
- Both detract from abstraction of communication
  - introduction of extraneous artefacts

Previous extension proposals

- Vista OO-VHDL (Swamy *et al*)
- LaMI (Benzakki & Djaffri)
- Bournemouth/IBM-UK (Cabanis *et al*)

➡ Based on OO extensions to VHDL

  – entity/architecture as a class, components as objects
  – entity specifies operations invoked by other processes
  – architecture encapsulates state

⇨ Monitor semantics with concurrency control

Other description languages

for describing behaviour, eg:

- StateCharts
- Estelle
- SDL
- CSP
- . . .

➡ Concurrent processes with message passing

StateCharts

- Extended FSM
  - hierarchical states
  - concurrent composition of substates
  - transitions on *events* (optionally *guarded*)
  - communication:
    - action in one chart ⇒ event in another chart
- Used in UML
  - events can be parameterized
  - procedural actions
- States are statically specified

Estelle

- Hierarchy of extended NFAs (*tasks*)
  - instances of declared modules
  - can be statically or dynamically instantiated
  - transitions enabled by event occurrence (msg arrival)
  - actions on transitions are Pascal-like statements
  - lock-step parallelism (select-transition/fire/action)
- Communication: message passing via links
  - links can be statically or dynamically created
  - non-blocking send, buffered at receiver

SDL

- Hierarchy of statically specified process sets
- Processes: statically or dynamically instantiated
  - instances of declared process types
  - behaviour specified as extended FSM
    - transitions enabled by message arrival
    - actions: assignment, branching, process creation, procedure call, message send
- Communication: message passing via channels
  - channels between process sets
  - non-blocking send, buffered at receiver
- Also a form of RPC

CSP

- Statically specified processes
  - contain sequentially executed actions
  - variable assignment, send, receive
- Synchronous message passing
  - statically specified channels

Comparison with VHDL

- Concurrent process model with message passing
  ≡ concurrent FSMs communicating by events
- VHDL
  - statically specified processes & channels (signals)
  - asynchronous unbuffered message passing (?!)
- Alternatives
  - static vs. dynamic process instantiation
  - static vs. dynamic channel creation
  - sync vs. async message passing

Language design issues

- Message passing mechanism and semantics
- Communication abstraction in entity interface
- Dynamic process creation/termination semantics
  - Process abstraction and interface
- Integration with existing language
- Integration with other extension (eg, SUAVE)

Message passing mechanism (1)

- Alternative 1: introduce new mechanism
  - eg, message channel, send & receive operations

```
channel elevator_call : floor_number;
channel elevator_location : floor_number;

elevator : process is
begin
    . . .
    receive calling_floor from elevator_call;
    send current_floor to elevator_location;
    . . .
end process elevator;
```


Communication in entity interface (1)

- Alternative 1

```
entity operator_console is
    port ( channel status : in status_msg;
           channel command : out command_msg );
end entity operator_console;
```

Message passing mechanism (2)

- Alternative 2: generalize existing mechanism
  - eg, abstract signals
  - transactions queued without delay mechanism
  - wait receives next value from queue

```
signal elevator_call : floor_number abstract;
signal elevator_location : floor_number abstract;

elevator : process is
begin
    . . .
    wait on elevator_call;    -- receive next message
    calling_floor := elevator_call;
    elevator_location <= current_floor;   -- send message
    . . .
end process elevator;
```


Communication in entity interface (2)

- Alternative 2

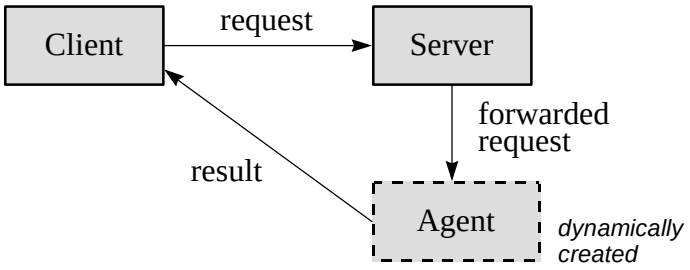
```
entity operator_console is
    port ( signal status : in status_msg abstract;
                signal command : out command_msg abstract );
end entity operator_console;
```

Dynamic process creation

- Eg, for multithreaded server
- Process types
  - needed for dynamic instantiation
  - parameterized by channel/signal
- Channel/signal types
  - needed for dynamic creation of channel/signal
  - for communication with dynamically created process

Process & channel types

- Example
  - multithreaded server creates new agent to handle each incoming request



Process & channel types

```
type result_value is . . . ;
type result_channel is channel result_value;
type result_ptr is access result_channel;

type request_info is record
        . . . ;
        result_please : result_ptr;
    end record request_info;

type client is process body
        port ( channel request : out request_info );
        variable result : result_ptr := new result_channel;
    begin
        send ( . . . , result ) to request;
        receive . . . from result.all;
    end process body client;
```


Process & channel types (cont)

```
type server is process body
        port ( channel request : in request_info );

        type agent is process body
                port ( channel request : in request_info );
                variable info : request_info;
            begin
                receive info from request;
                . . .
                send . . . to info.result_please.all;
                exit;
            end process body agent;

        . . .
```

Process & channel types (cont)

```
        . . .
        variable info : request_info;
        channel agent_request : request_info;
        type agent_ptr is access agent;
        variable new_agent : agent_ptr;
    begin  -- server
        receive info from request;
        new_agent := new agent
                                port map ( agent_request );
        send info to agent_request;
    end process body server;
```


Conclusions

- Improve support for system modeling in VHDL
  - more abstract communication
  - process abstraction
    - process and channel types
    - dynamic creation/termination
- Integrate with existing language
  - extend existing mechanism where appropriate
- Enables single-language design flow
  - hardware/software co-design