Asynchronous design using the DISP programming language and the tools di2pn and petrify

Mark B. Josephs and Dennis P. Furey

Centre for Concurrent Systems & VLSI
School of CISM
South Bank University, London

Overview

- Background on DISP programming language
- Example 1: controller for micropipeline stage
- Example 2: datapath for an adder cell
- Validation and synthesis with di2pn and petrify tools
Background on DISP programming language

**CSP = Communicating Sequential Processes**

- original version introduced by Hoare in 1978
- evolved into Occam language for programming Transputers in 1984
- structured, parallel programming language
- processes communicate on channels
  * `send(e)` and `recv(x)` are both blocking (synchronous) calls, synchronising with effect of distributed assignment `x:=e`
- theoretical studies usually simplify CSP to a *process algebra*
  * modelling channel communication with undirected events
  * excluding program variables and assignment

**DI-Algebra = Delay-Insensitive Algebra**

- process algebra introduced by Josephs and Udding in 1989
- facilitates formal verification of asynchronous circuits
  * abstracting logic blocks as processes, wires as channels, and transitions as communications
- variant of “theoretical” CSP that captures delay-insensitivity
  * `send` is non-blocking (asynchronous), but `recv` is blocking
  * order of sending on distinct channels makes no difference
  * order of receiving on distinct channels makes no difference
  * sending twice on a channel without receiving (an ack) from another channel is unsafe
- no notion of “successful termination”
  * processes cannot be composed in sequence
  * cyclic behaviour has to be modelled by recursion
DISP = Delay-Insensitive Sequential Processes

- introduced by Josephs and Furey in 2000, by incorporating CSP’s notion of successful termination into DI-Algebra
  * sequential composition and iteration of processes are allowed
  * upon termination, any communications still in transit are passed on to successor process
  * input/output-bursts used as primitive statements, cf. Burst-Mode specifications
  * program variables and assignment not supported
- convenient for précisé description of switching behaviour of asynchronous circuits
  * controllers that use request and acknowledge signals
  * self-timed datapaths
- similar expressivity to Martin’s “handshaking expansions”

Example 1: controller for micropipeline stage

Environment:

IN par OUT par LT

where

IN = pushback aIN ;
  forever do aIN/rIN end

OUT = forever do rOUT/aOUT end

LT = forever do rLT/aLT end
Simple controller:

pushback aOUT;
forever do
  rIN,aOUT/rLT, rOUT; aLT/aIN
end

This unfolds to

pushback aOUT;
forever do
  rIN,aOUT/rLT, rOUT; aLT/aIN;
  rIN,aOUT/rLT, rOUT; aLT/aIN
end

Semi-decoupled controller:

pushback aOUT;
forever do
  rIN/rLT;
  aLT/aIN par aOUT/rOUT;
  rIN,aOUT/rLT, rOUT; aLT/aIN
end

Fully-decoupled controller:

pushback aOUT;
forever do
  rIN/rLT;
  aLT/aIN par aOUT/rOUT;
end
Example 2: datapath for an adder cell

Environment:

forever do
  aD,aS/rA,rB,rC
end
par A par B par C par D par S

where

A = pushback rA;
  forever do
    ( rA/fA; rA/fA ) or ( rA/tA; rA/tA )
  end

...

D = forever do
  select fD/aD alt tD/aD end
end

S = forever do
  select fS/aS alt tS/aS end
end
Adder cell:

forever do
  select fA,fB/fD then
    select fC/fS then fA,fB,fC/fD,fS
    alt tC/tS then ( fA,fB/fD par tC/tS )
    end
  alt fB,fC/fD then
    select fA/fS then fA,fB,fC/fD,fS
    alt tA/tS then ( fB,fC/fD par tA/tS )
    end
  alt fC,fA/fD then
    select fB/fS then fA,fB,fC/fD,fS
    alt tB/tS then ( fC,fA/fD par tB/tS )
    end
  alt tA,tB/tD then
    select fC/fS then ( tA,tB/tD par fC/fS )
    alt tC/tS then tA,tB,tC/tD,tS
    end
  alt tB,tC/tD then
    select fA/fS then ( tB,tC/tD par fA/fS )
    alt tA/tS then tA,tB,tC/tD,tS
    end
  alt tC,tA/tD then
    select fB/fS then ( tC,tA/tD par fB/fS )
    alt tB/tS then tA,tB,tC/tD,tS
    end
end
Validation and synthesis with di2pn and petrify tools

| DISP specification = pair of programs (module + environment) |

- **di2pn** and **petrify** can be used to
  - automatically validate DISP specification
  - automatically synthesise asynchronous logic that implements the module
- **di2pn** is front end
  - translates specification into Petri net
  - uses same text-file format as petrify
- **petrify** does actual validation and logic synthesis, interpreting Petri net as STG
  - no need for designer to look at the Petri net!
- tools downloadable from http://www.sbu.ac.uk/~fureyd/di2pn/ and http://www.lsi.upc.es/~jordic/petrify/

- **petrify** -no checks that specification is delay-insensitive (Petri-net is 1-safe) and free from deadlock
  - e.g., **FATAL ERROR**: (Boundedness): marking exceeds the capacity for place p39+
  - e.g., **FATAL ERROR**: No initial marking has been defined
  - e.g., Error: There are deadlock states in the system.
    - Trace of events to a deadlock state:
      - b a t54
  - N.B. -no option tells petrify that a (transformed) Petri net is not required
• DISP programs do not express the direction (+ or -) of each transition
  * dit2pm yields STG with “toggle-transitions”
• The decision as to whether to initialise external signals of module to logic-0 or logic-1 has to be taken prior to logic synthesis
  * By default, petrify assumes that all signals are initially 0
  * Directive .initial state a b ... can be inserted into the text-file for those signals that are initially 1
• petrify -untog transforms Petri net so that toggle-transitions are replaced by rising and falling transitions
  * Designer can re-examine specification at this lower level of abstraction

• petrify -no -csc checks that STG has “complete state coding” property
  * Required before logic minimisation can be used to synthesise speed-independent implementation
  * Self-timed datapaths should have csc!
• petrify -no -cg or -gc attempts to solve for csc and to perform logic minimisation, targeting complex gates or generalised C elements
Conclusion

petrify — powerful tool for analysis and transformation of Petri nets and for synthesis of asynchronous logic

di2pn — front-end to petrify that allows designs to be conveniently entered as DISP programs

Combination of tools offers innovative methodology for design of (relatively small) asynchronous logic blocks

Methodology has potential for application to real-world design problems