

42: A Component-Based Approach to Virtual Prototyping of Heterogeneous Embedded Systems

Ph.D. Defense

Tayeb BOUHADIBA



Directrice de thèse : Florence MARANINCHI

Jury:

Marc POUZET

Rapporteur

Lionel SEINTURIER

Rapporteur

Jean-Bernard STEFANI

Examineur

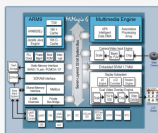
September 15th 2010



Embedded Systems



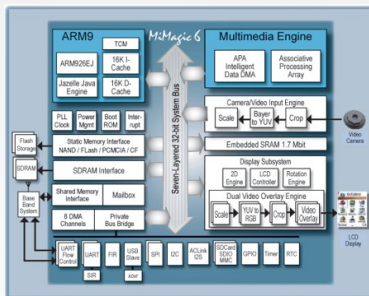
Consumer Electronics



Safety Critical Systems



Embedded Systems, Components & Heterogeneity



Software

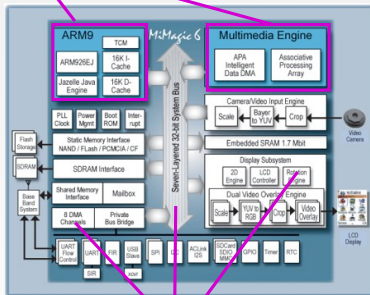
+

Hardware



Embedded Systems, Components & Heterogeneity

Processors



Hardware IP's
(Intellectual Properties)

Software

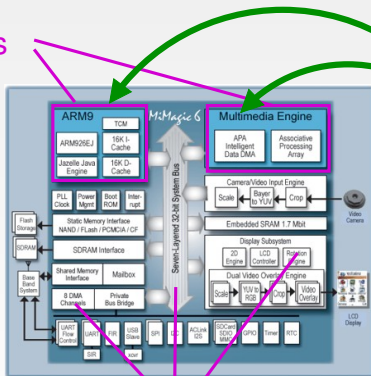
+

Hardware




Embedded Systems, Components & Heterogeneity

Processors



Hardware IP's
(Intellectual Properties)


 Software
 +
 Hardware



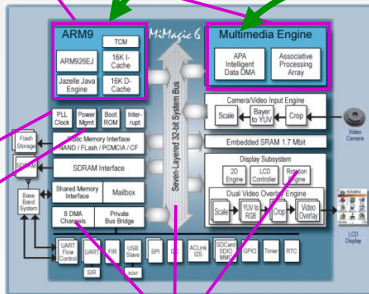
Embedded Systems, Components & Heterogeneity

Processors

Analog



Digital



Hardware IP's
(Intellectual Properties)



Software



Hardware



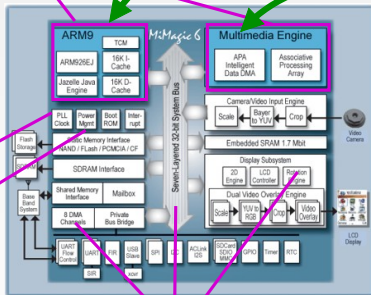
Embedded Systems, Components & Heterogeneity

Processors

Analog



Digital



Hardware IP's
(Intellectual Properties)

Synchronous

Asynchronous



Software

+

Hardware



Virtual Prototyping

Virtual Prototyping

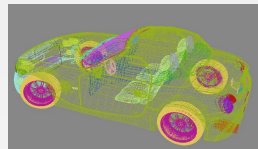
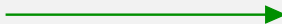
=
An Executable Model Before the System is
Manufactured



Virtual Prototyping



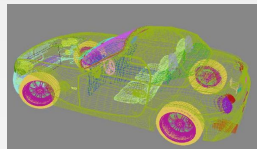
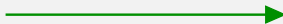
Modeling



Virtual Prototyping



Modeling



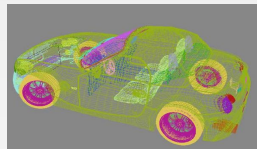
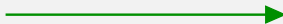
Software



Virtual Prototyping



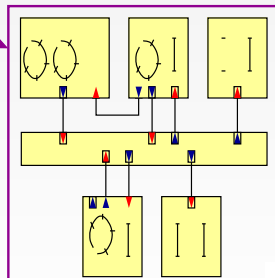
Modeling



Virtual Prototyping



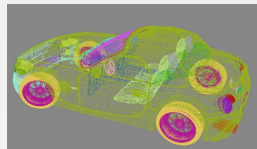
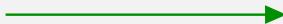
Software



Virtual Prototyping



Modeling

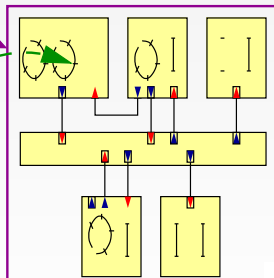


Virtual Prototyping



Executability

Software

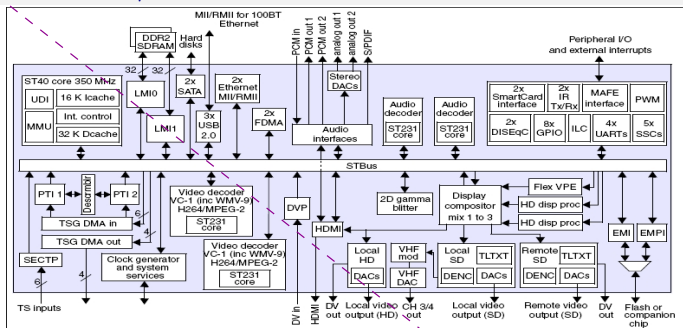


Contents

- 1 Introduction & Sources of Inspiration
 - Virtual Prototyping of Embedded Systems
 - Modeling Hardware/Software with Synchronous Languages
 - Ptolemy
 - Specifying Components: Contracts
- 2 Overview of 42 & Examples [\[GPCE07\]](#)
- 3 Hardware Simulation by Interpreting Contracts [\[COORD09\]](#)
- 4 Using 42 Together with Existing Approaches [\[EMSOFT09\]](#)
- 5 Some Related Work



SystemC/TLM for Systems-on-a-Chip



```

while(true)
x = 42;
while (x neq 0)
y = read(addrM + x);
write(addrM , x);
...
write(addrD+0x0, addrM);
write(addrD+0x1, 42);
write(addrD+0x4, 0x01);
...

```

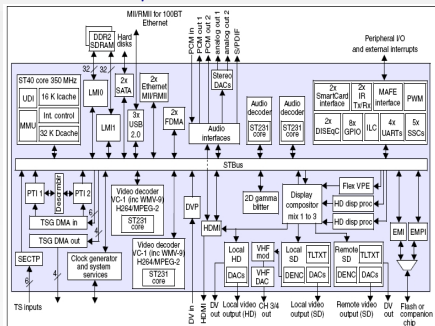
Embedded Software

Hardware Architecture



A SoC (System-on-a-Chip)

SystemC/TLM for Systems-on-a-Chip



TLM
(Transaction-Level Modeling)
provides

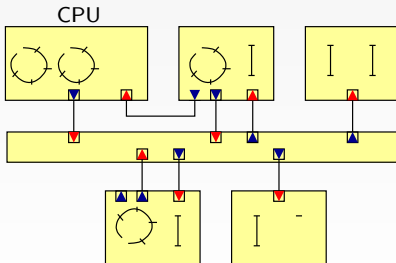
Virtual Prototype

```

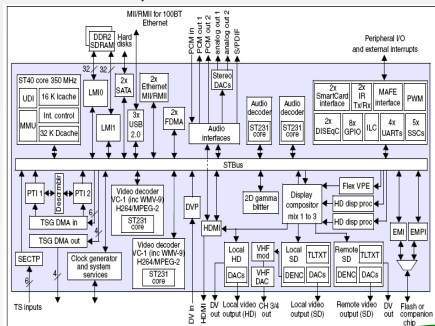
while(true)
x = 42;
while (x neq 0)
y = read(addrM + x);
write(addrM , x);
...
write(addrD+0x0, addrM);
write(addrD+0x1, 42);
write(addrD+0x4, 0x01);
...

```

Embedded Software



SystemC/TLM for Systems-on-a-Chip



TLM
(Transaction-Level Modeling)
provides

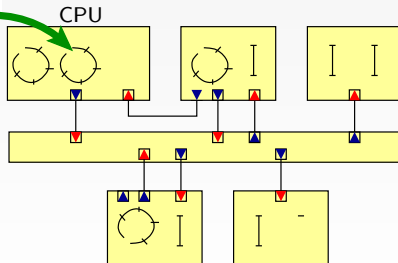
Virtual Prototype

```

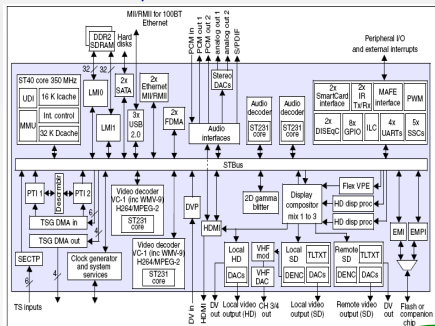
while(true)
x = 42;
while (x neq 0)
y = read(addrM + x);
write(addrM , x);
...
write(addrD+0x0, addrM);
write(addrD+0x1, 42);
write(addrD+0x4, 0x01);
...

```

Embedded Software



SystemC/TLM for Systems-on-a-Chip



TLM
(Transaction-Level Modeling)
provides

- Early Available

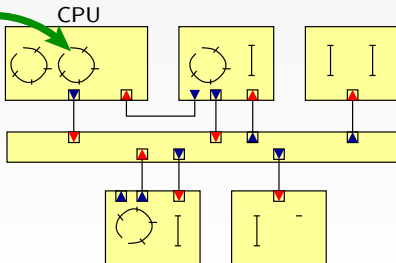
Virtual Prototype

```

while(true)
x = 42;
while (x neq 0)
y = read(addrM + x);
write(addrM , x);
...
write(addrD+0x0, addrM);
write(addrD+0x1, 42);
write(addrD+0x4, 0x01);
...

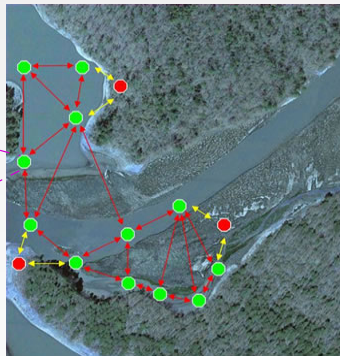
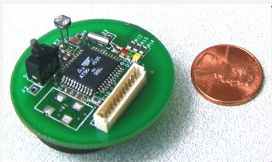
```

Embedded Software



Virtual Prototyping of Sensor Networks

CPU, Sensor, Memory, Radio, **Battery**



- Several sensors communicating by radio
- Network lifetime depends on energy consumption
- **Use of virtual prototyping to study non-functional aspects (e.g., energy)**



Contents

- 1 Introduction & Sources of Inspiration
 - Virtual Prototyping of Embedded Systems
 - Modeling Hardware/Software with Synchronous Languages
 - Ptolemy
 - Specifying Components: Contracts
- 2 Overview of 42 & Examples [\[GPCE07\]](#)
- 3 Hardware Simulation by Interpreting Contracts [\[COORD09\]](#)
- 4 Using 42 Together with Existing Approaches [\[EMSOFT09\]](#)
- 5 Some Related Work



Modeling HW/SW with Synchronous Languages



PFS (Proximity Flight Safety)
is Embedded in the ATV

ATV (Automated Transfer Vehicule)



Modeling HW/SW with Synchronous Languages



PFS (Proximity Flight Safety)
is Embedded in the ATV

ATV (Automated Transfer Vehicle)

The original model of the PFS
is written in AADL
(Architecture Analysis and Design Language)
(Not Executable)



Modeling HW/SW with Synchronous Languages

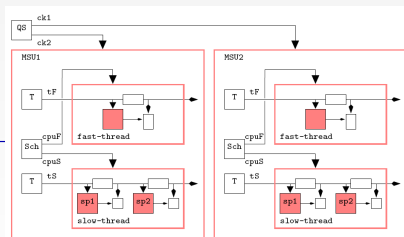


PFS (Proximity Flight Safety) is Embedded in the ATV

The original model of the PFS is written in AADL
(Architecture Analysis and Design Language)
(Not Executable)

ATV (Automated Transfer Vehicle)

Automatic Translation into Lustre



PFS description in Lustre

Simulation

Automatic Testing

"E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, D. Lesens, Virtual Execution of AADL Models via a Translation into Synchronous Programs" [EMSOFT07]



Contents

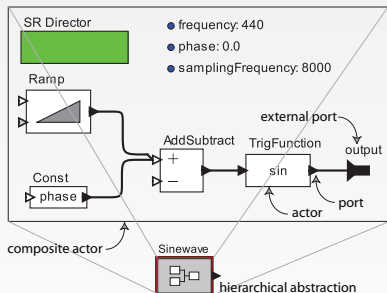
- 1 Introduction & Sources of Inspiration
 - Virtual Prototyping of Embedded Systems
 - Modeling Hardware/Software with Synchronous Languages
 - Ptolemy
 - Specifying Components: Contracts
- 2 Overview of 42 & Examples [\[GPCE07\]](#)
- 3 Hardware Simulation by Interpreting Contracts [\[COORD09\]](#)
- 4 Using 42 Together with Existing Approaches [\[EMSOFT09\]](#)
- 5 Some Related Work



Ptolemy

ptolemy.eecs.berkeley.edu

MoCC : Model of Computation and Communication



- Components are actors
- The director implements a MoCC
- Hierarchical framework

- \exists a catalogue of predefined MoCCs:
Synchronous Reactive, Discrete Event, Continuous Time, etc.



Contents

- 1 Introduction & Sources of Inspiration
 - Virtual Prototyping of Embedded Systems
 - Modeling Hardware/Software with Synchronous Languages
 - Ptolemy
 - Specifying Components: Contracts
- 2 Overview of 42 & Examples [\[GPCE07\]](#)
- 3 Hardware Simulation by Interpreting Contracts [\[COORD09\]](#)
- 4 Using 42 Together with Existing Approaches [\[EMSOFT09\]](#)
- 5 Some Related Work



Classification of Contracts in the CBSE Community

(Component-Based Software Engineering)

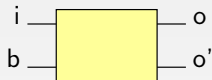
- **Syntactic contracts**
 - Interface Description languages (Java-IDL, WSDL, IDL, etc.)
- **Behavioral contracts**
 - pre and post conditions, assertions, etc. (Design by ContractsTM, Eiffel, iContracts, etc.)
- **Synchronization contracts**
 - Sequence of method-calls, Component synchronization, etc. (PROCOL, Interface automata, Session types, etc.)
- **Quality of service contracts**
 - Resource consumption, Image quality, etc.



Contracts in the Hardware Community

and the Synchronous Languages

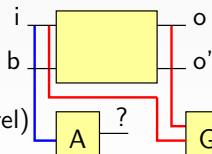
- Don't care conditions
 - Hardware optimization
- Conditional dependencies in Signal
 - Cycle analysis in synchronous circuits
- Assume/Guarantee reasoning
 - Modular verification (K. McMillan)
 - Executable specifications for Lustre (L. Morel)



Input $(ib)=(01)$ never occurs



o depends on i when $b=true$



Observation & Motivations of 42

In each context of Virtual Prototyping of embedded systems, there exists a notion of components and MoCCs related issues

42 aims at providing:

- A **language-independent** component-based framework for modeling hardware/software systems.
- Support for a clean definition of components, and help enforcing the **FAMAPASAP** (Forget As Much As Possible As Soon As Possible) principle.
- Support for integration of **existing code and models** in open virtual prototyping environments.

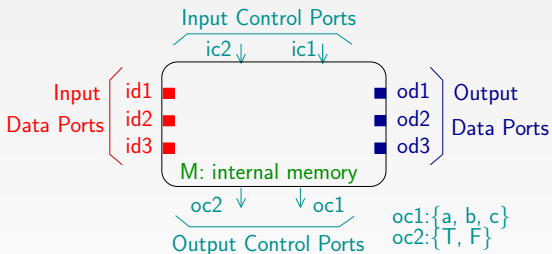


Contents

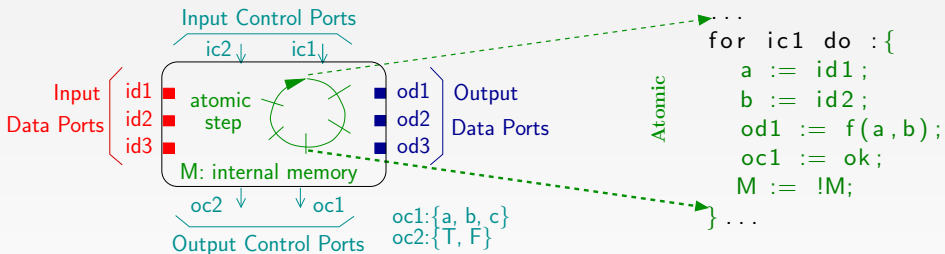
- 1 Introduction & Sources of Inspiration
- 2 Overview of 42 & Examples [GPCE07]
 - Components & Assemblies
 - Explicit Specification of Components (Contracts)
- 3 Hardware Simulation by Interpreting Contracts [COORD09]
- 4 Using 42 Together with Existing Approaches [EMSOFT09]
- 5 Some Related Work
- 6 Summary



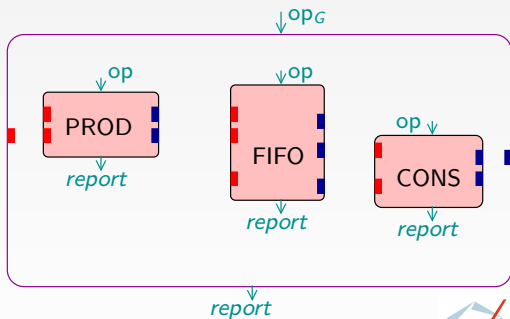
42 in a Nutshell: Basic Components



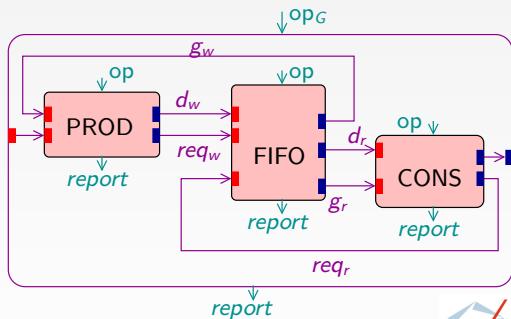
42 in a Nutshell: Basic Components



42 in a Nutshell: Assembling Components

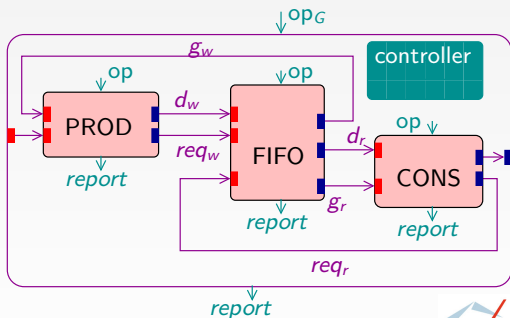


42 in a Nutshell: Assembling Components



42 in a Nutshell: Assembling Components

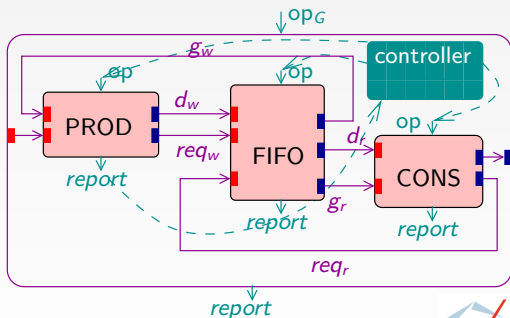
For each OP_G the controller:



42 in a Nutshell: Assembling Components

For each OP_G the controller:

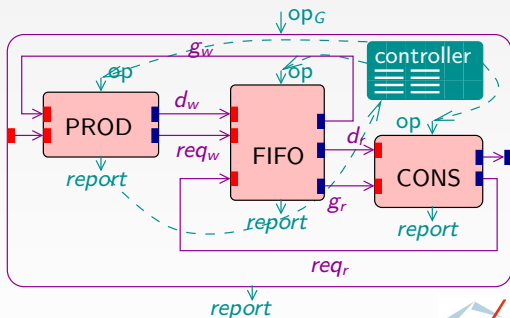
Activates PROD, CONS, FIFO through op
 Reads their output control ports (report)



42 in a Nutshell: Assembling Components

For each OP_G the controller:

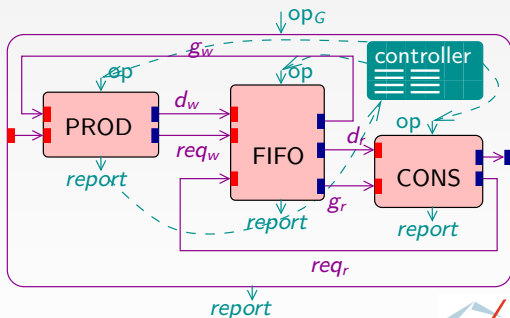
- Activates PROD, CONS, FIFO through op
- Reads their output control ports ($report$)
- Manages a temporary memory (req_r, dr, \dots)



42 in a Nutshell: Assembling Components

For each OP_G the controller:

- Activates PROD, CONS, FIFO through op
- Reads their output control ports ($report$)
- Manages a temporary memory (req_r, dr, \dots)
- Sets global output values



42 in a Nutshell: Assembling Components

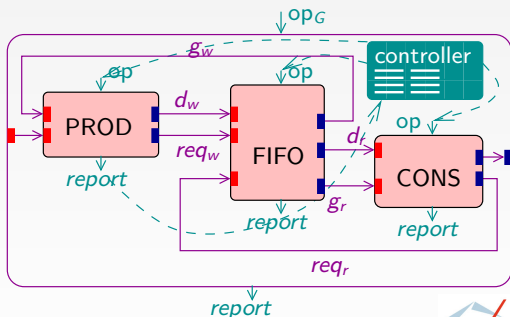
```

Controller is {
var M : bool;
for OPG do :{ /*defines opg.
  dw, dr, reqw, ...: fifo(1,int);
  M := random();
  if (M) {
    PROD.op ; reqw.put; reqw.get;
    FIFO.op ; gw.put; gw.get;
    a := FIFO.report; /*reads oc.
    if(a==ok){ /*output control
      PROD.op;dw.put; dw.get;
      FIFO.op; /*activates FIFO.
      ...
    }
  }
} else {
  CONS.op; reqr.put; reqr.get;
  FIFO.op ; gr.put; gr.get;
  a := FIFO.report;
  ...
}
}
}

```

For each OP_G the controller:

- Activates PROD, CONS, FIFO through op
- Reads their output control ports (report)
- Manages a temporary memory (reqr, dr,...)
- Sets global output values



42 in a Nutshell: Assembling Components

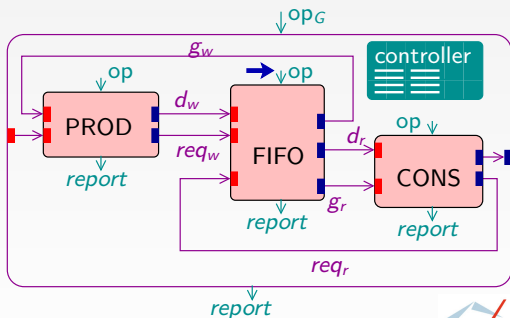
```

Controller is {
  var M : bool;
  for OPG do :{ /* defines opg.
    dw, dr, reqw, ...: fifo(1,int);
    M := random();
    if (M) {
      PROD.op ; reqw.put; reqw.get;
      FIFO.op ; gw.put; gw.get;
      a := FIFO.report; /*reads oc.
      if(a==ok){ /*output control
        PROD.op;dw.put; dw.get;
        FIFO.op; /*activates FIFO.
        ...
      }
    }
  }
  else {
    CONS.op; reqr.put; reqr.get;
    FIFO.op; gr.put; gr.get;
    a := FIFO.report;
    ...
  }
}

```

For each OP_G the controller:

- Activates PROD, CONS, FIFO through op
- Reads their output control ports (report)
- Manages a temporary memory (reqr, dr,...)
- Sets global output values



42 in a Nutshell: Assembling Components

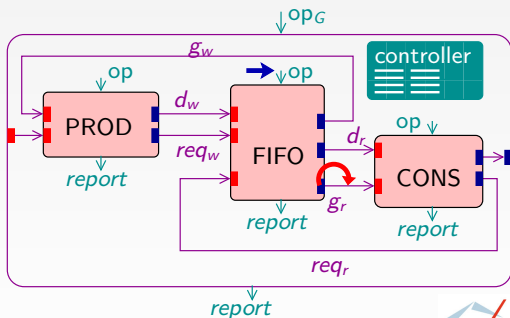
```

Controller is {
  var M : bool;
  for OPG do :{ /* defines opg.
    dw, dr, reqw, ...: fifo(1,int);
    M := random();
    if (M) {
      PROD.op ; reqw.put; reqw.get;
      FIFO.op ; gw.put; gw.get;
      a := FIFO.report; /*reads oc.
      if(a==ok){ /*output control
        PROD.op;dw.put; dw.get;
        FIFO.op; /*activates FIFO.
        ...
      }
    }
  }
  else {
    CONS.op; reqr.put; reqr.get;
    FIFO.op; gr.put; gr.get;
    a := FIFO.report;
    ...
  }
}

```

For each OP_G the controller:

- Activates PROD, CONS, FIFO through op
- Reads their output control ports (report)
- Manages a temporary memory (reqr, dr,...)
- Sets global output values



42 in a Nutshell: Assembling Components

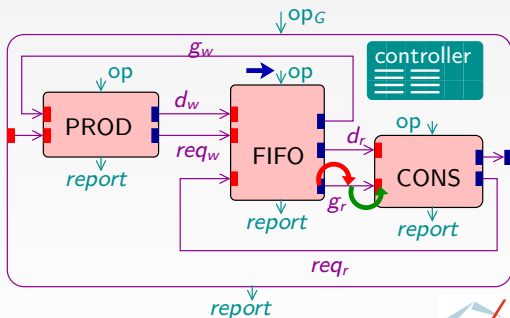
```

Controller is {
var M : bool;
for OPG do :{ /* defines opg.
dw, dr, reqw, ...: fifo(1,int);
M := random();
if (M) {
PROD.op ; reqw.put; reqw.get;
FIFO.op ; gw.put; gw.get;
a := FIFO.report; /*reads oc.
if(a==ok){ /*output control
PROD.op;dw.put; dw.get;
FIFO.op; /*activates FIFO.
...
}
...
} else {
CONS.op; reqr.put; reqr.get;
FIFO.op; gr.put; gr.get;
a := FIFO.report;
...
}
}
}

```

For each OP_G the controller:

- Activates PROD, CONS, FIFO through op
- Reads their output control ports (report)
- Manages a temporary memory (reqr, dr,...)
- Sets global output values



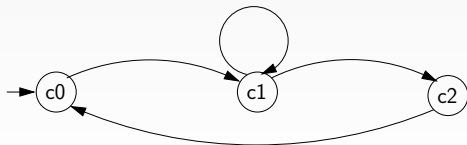
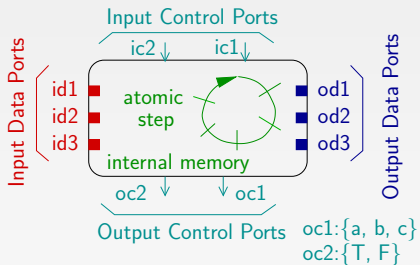
Summary of 42 Basics

Summary of 42

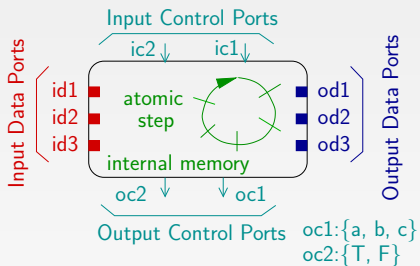
- Hierarchical model \implies modeling heterogeneity
in the same spirit as Ptolemy
- Controllers are expressed as programs
identification of the basic primitives for describing MoCCs
- Separation of control/data to enforce the FAMAPASAP
more details later



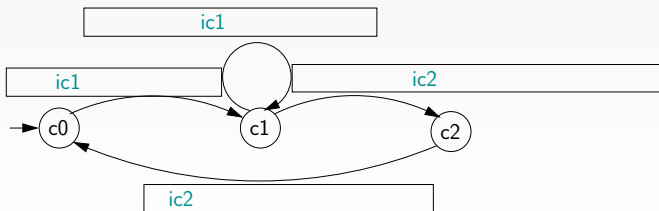
Control Contracts for 42 Components



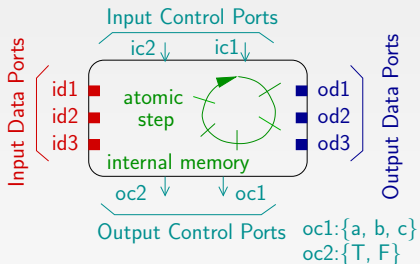
Control Contracts for 42 Components



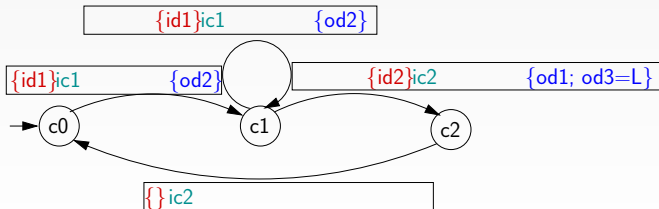
- Allowed activation sequences



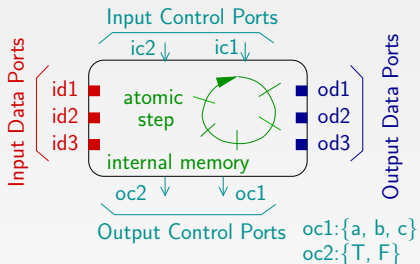
Control Contracts for 42 Components



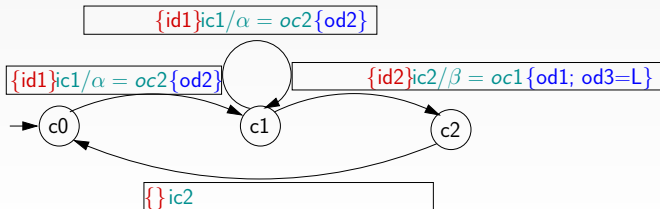
- Allowed activation sequences
- Data dependencies (**Required**, **Provided**)



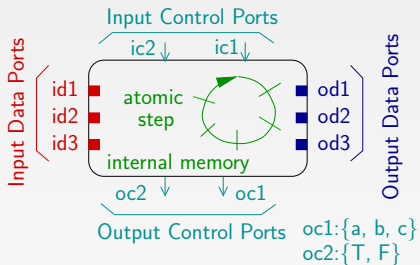
Control Contracts for 42 Components



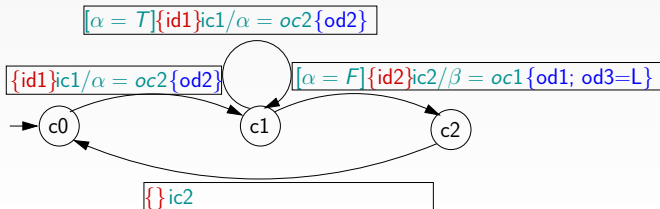
- Allowed activation sequences
- Data dependencies (**Required**, **Provided**)
- Control information



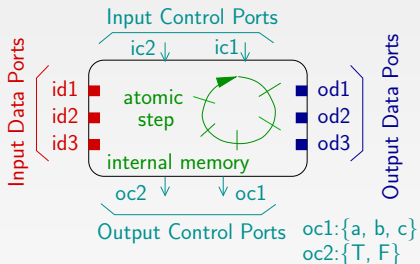
Control Contracts for 42 Components



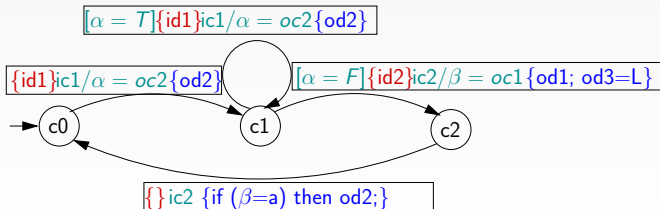
- Allowed activation sequences
- Data dependencies (**Required**, **Provided**)
- Control information
- Conditional activations.



Control Contracts for 42 Components

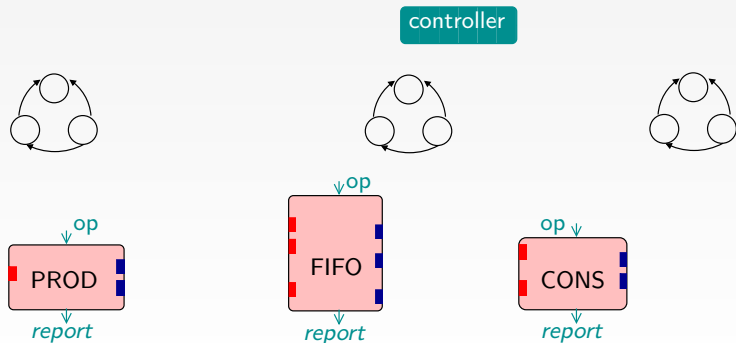


- Allowed activation sequences
- Data dependencies (**Required**, **Provided**)
- Control information
- Conditional activations.
- Conditional data dependencies.



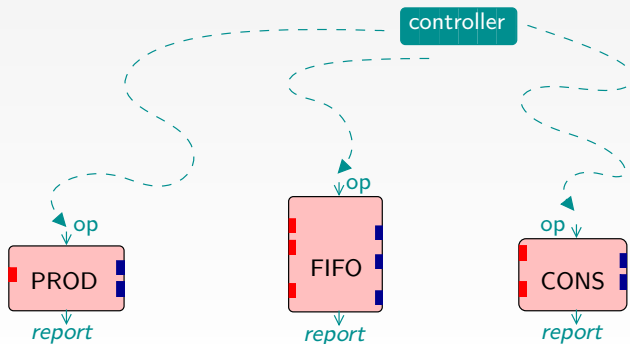
What are 42 Contracts used for?

42 contracts (+ the Architecture Description Language)



What are 42 Contracts used for?

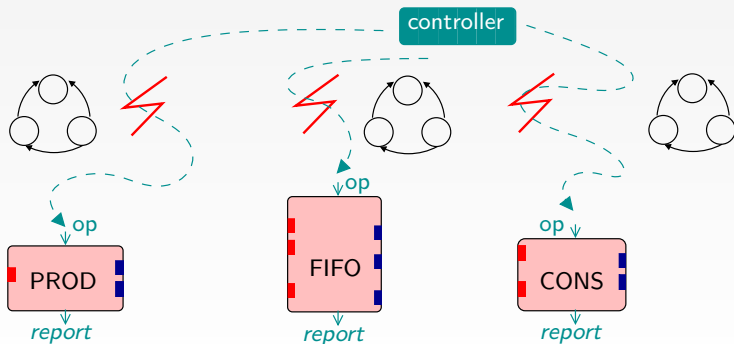
42 contracts (+ the Architecture Description Language)



What are 42 Contracts used for?

42 contracts (+ the Architecture Description Language)

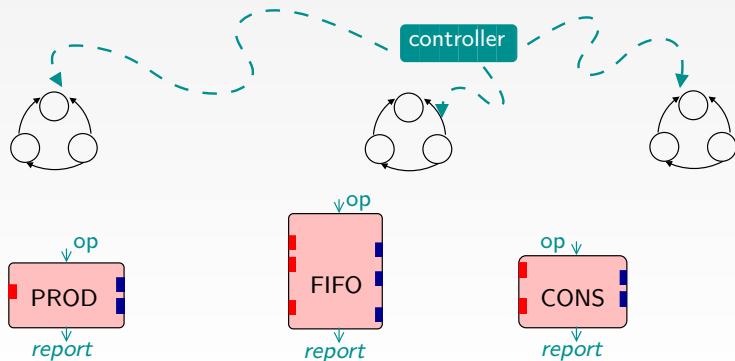
- Testing controllers/components consistency



What are 42 Contracts used for?

42 contracts (+ the Architecture Description Language)

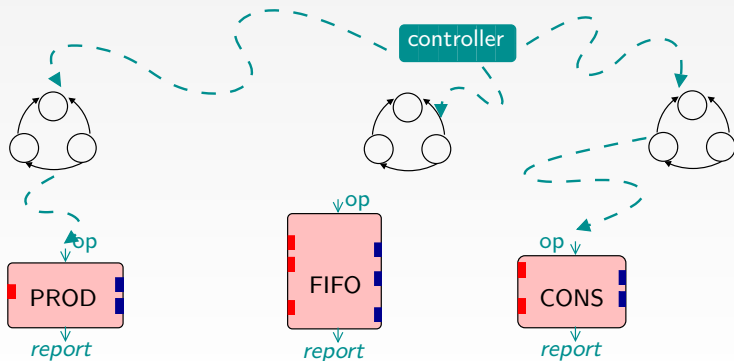
- Testing controllers/components consistency
- Contract Interpretation



What are 42 Contracts used for?

42 contracts (+ the Architecture Description Language)

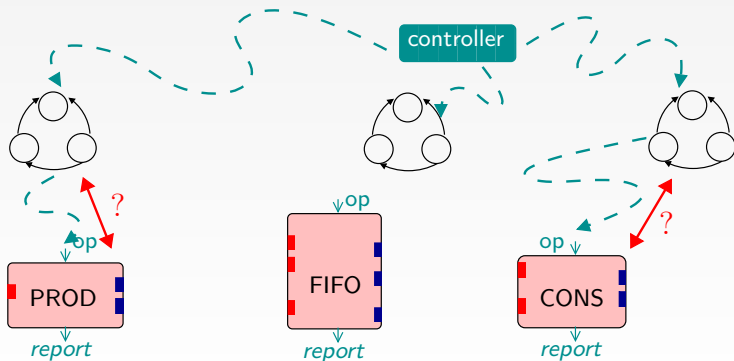
- Testing controllers/components consistency
- Contract Interpretation



What are 42 Contracts used for?

42 contracts (+ the Architecture Description Language)

- Testing controllers/components consistency
- Contract Interpretation
- Testing component/contract consistency



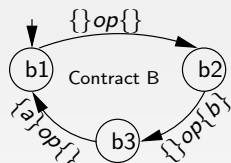
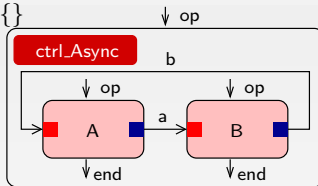
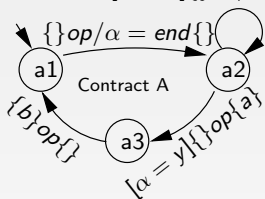
Contents

- 1 Introduction & Sources of Inspiration
- 2 Overview of 42 & Examples [GPCE07]
- 3 Hardware Simulation by Interpreting Contracts [COORD09]**
 - Contract Interpretation
 - Contract Interpretation for Hardware Simulation
 - Executing Embedded Software on Hardware Models
- 4 Using 42 Together with Existing Approaches [EMSOFT09]
- 5 Some Related Work
- 6 Summary

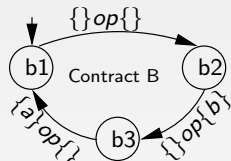
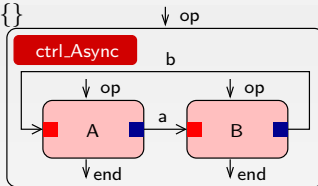
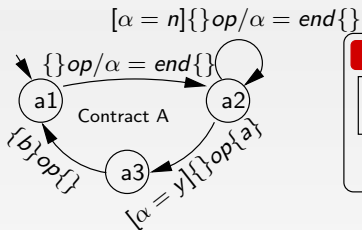


Contract Interpretation: Principle

$$[\alpha = n]\{\{op/\alpha = end\}\}$$



Contract Interpretation: Principle



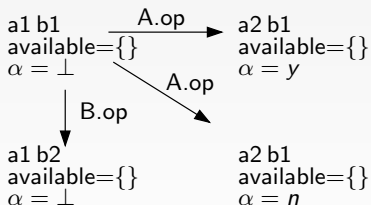
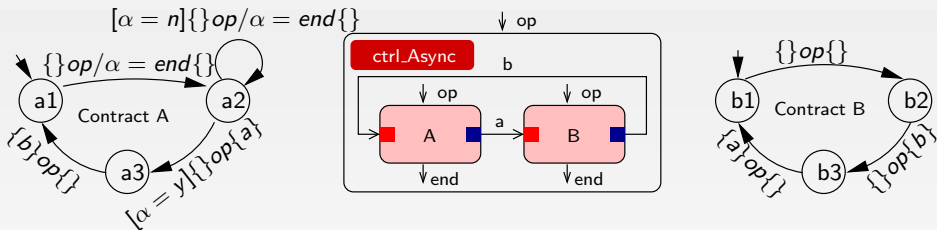
a1 b1
available= $\{\}$
 $\alpha = \perp$

The controller maintains:

- The current state of each contract
- The set of available data
- The values of the variables(α)



Contract Interpretation: Principle

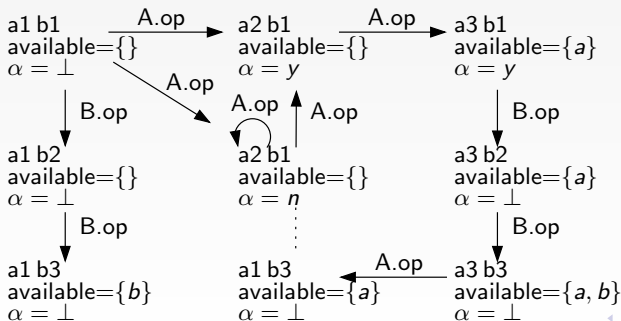
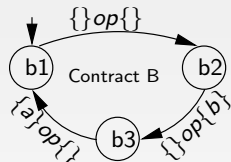
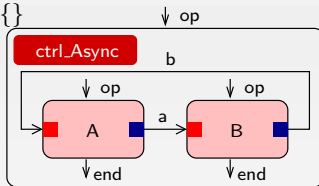
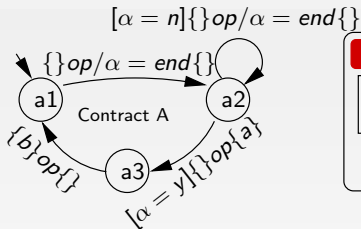


For each global activation :

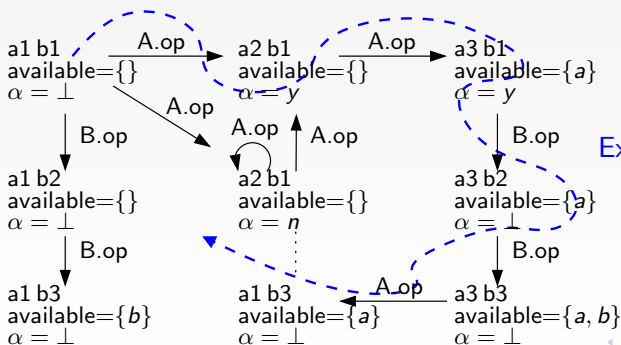
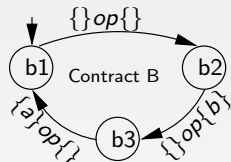
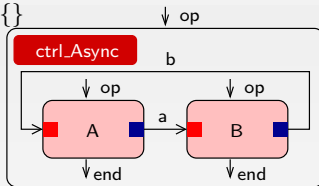
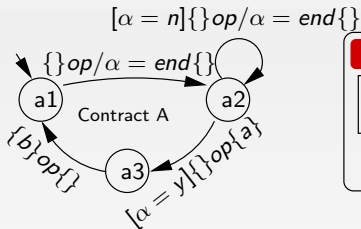
- Depending on the available data it selects a component and activates it
- The set of available data is updated
- The output controls are given non-deterministic values



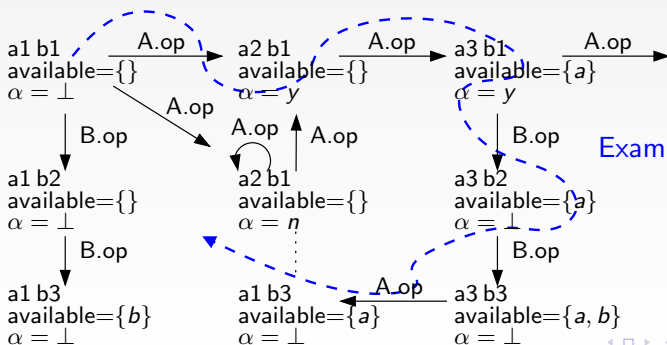
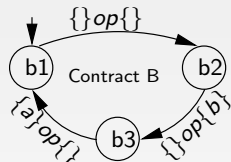
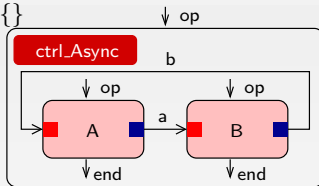
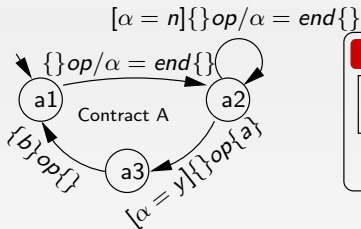
Contract Interpretation: Principle



Contract Interpretation: Principle



Contract Interpretation: Principle

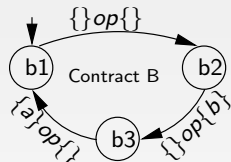
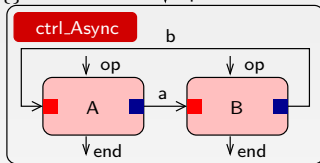
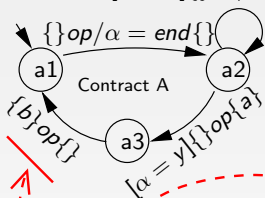


Example execution path

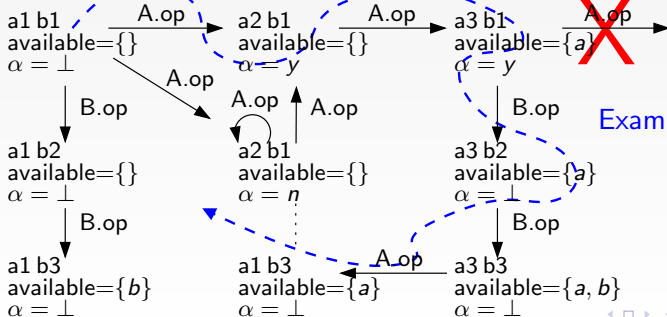


Contract Interpretation: Principle

$$[\alpha = n] \{ \} op / \alpha = end \{ \}$$

 $\downarrow op$


requires b



Modeling Hardware with 42: Case Study

Embedded Software

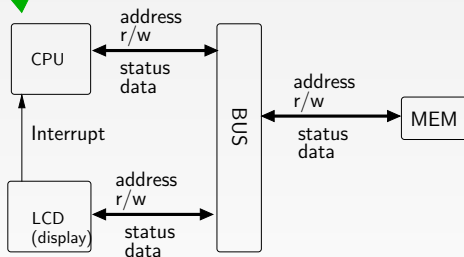
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
/*writing the green image*/
for(int x=0; x<width * height)
    write_mem(green);
    write_lcd(0x01,0x1) ;
    wait_interrupt ();

/*writing the blue image*/
for(int x=0; x<width * height)
    write_mem(blue);
    write_lcd(0x01,0x1);
    wait_interrupt ();

/*writing the red image*/
for(int x=0; x<width * height)
    write_mem(red);
    write_lcd(0x01,0x1);
    wait_interrupt ();
}
}

```



Modeling Hardware with 42: Case Study

Embedded Software

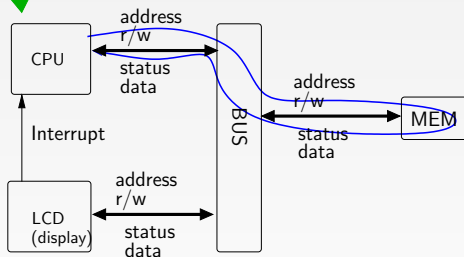
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
/*writing the green image*/
for(int x=0; x<width * height)
    write_mem(green);
    write_lcd(0x01,0x1) ;
    wait_interrupt ();

/*writing the blue image*/
for(int x=0; x<width * height)
    write_mem(blue);
    write_lcd(0x01,0x1);
    wait_interrupt ();

/*writing the red image*/
for(int x=0; x<width * height)
    write_mem(red);
    write_lcd(0x01,0x1);
    wait_interrupt ();
}
}

```



Modeling Hardware with 42: Case Study

Embedded Software

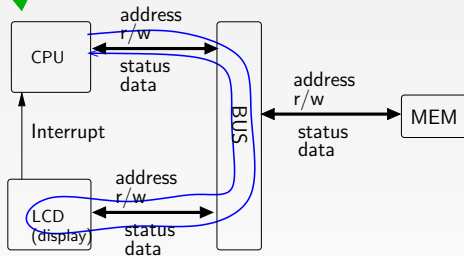
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
/*writing the green image*/
for(int x=0; x<width * height)
    write_mem(green);
    write_lcd(0x01,0x1) ;
    wait_interrupt ();

/*writing the blue image*/
for(int x=0; x<width * height)
    write_mem(blue);
    write_lcd(0x01,0x1);
    wait_interrupt ();

/*writing the red image*/
for(int x=0; x<width * height)
    write_mem(red);
    write_lcd(0x01,0x1);
    wait_interrupt ();
}
}

```



Modeling Hardware with 42: Case Study

Embedded Software

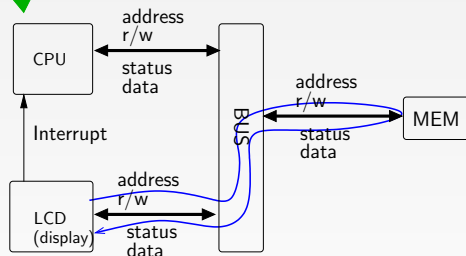
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
  /*writing the green image*/
  for(int x=0; x<width * height)
    write_mem(green);
  write_lcd(0x01,0x1) ;
  wait_interrupt ();

  /*writing the blue image*/
  for(int x=0; x<width * height)
    write_mem(blue);
  write_lcd(0x01,0x1);
  wait_interrupt ();

  /*writing the red image*/
  for(int x=0; x<width * height)
    write_mem(red);
  write_lcd(0x01,0x1);
  wait_interrupt ();
}
}

```



Modeling Hardware with 42: Case Study

Embedded Software

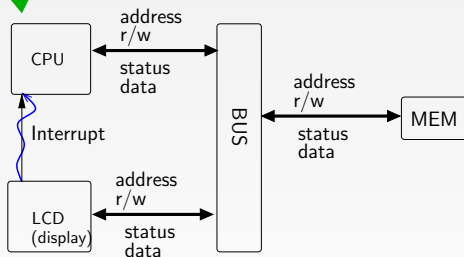
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
/*writing the green image*/
for(int x=0; x<width * height)
    write_mem(green);
    write_lcd(0x01,0x1) ;
    wait_interrupt();

/*writing the blue image*/
for(int x=0; x<width * height)
    write_mem(blue);
    write_lcd(0x01,0x1);
    wait_interrupt();

/*writing the red image*/
for(int x=0; x<width * height)
    write_mem(red);
    write_lcd(0x01,0x1);
    wait_interrupt();
}
}

```



Modeling Hardware with 42: Case Study

Embedded Software

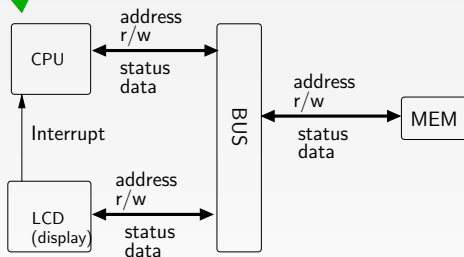
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
  /*writing the green image*/
  for(int x=0; x<width * height)
    write_mem(green);
  write_lcd(0x01,0x1) ;
  wait_interrupt();

  /*writing the blue image*/
  for(int x=0; x<width * height)
    write_mem(blue);
  write_lcd(0x01,0x1);
  wait_interrupt();

  /*writing the red image*/
  for(int x=0; x<width * height)
    write_mem(red);
  write_lcd(0x01,0x1);
  wait_interrupt();
}
}

```



a



b



c



d



Modeling Hardware with 42: Case Study

Embedded Software

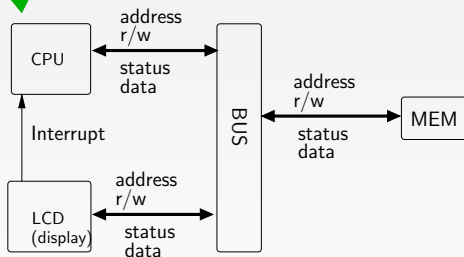
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
/**writing the green image*/
for(int x=0; x<width * height)
    write_mem(green);
    write_lcd(0x01,0x1) ;
    wait_interrupt ();

/**writing the blue image*/
for(int x=0; x<width * height)
    write_mem(blue);
    write_lcd(0x01,0x1);
    wait_interrupt ();

/**writing the red image*/
for(int x=0; x<width * height)
    write_mem(red);
    write_lcd(0x01,0x1);
    wait_interrupt ();
}
}

```



a

ok



b



c



d



Modeling Hardware with 42: Case Study

Embedded Software

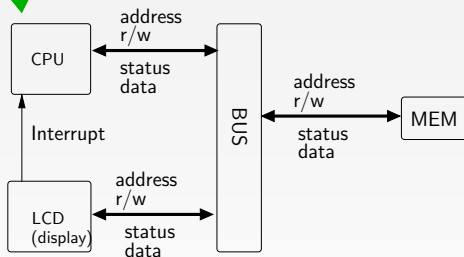
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
/**writing the green image*/
for(int x=0; x<width * height)
    write_mem(green);
    write_lcd(0x01,0x1) ;
    wait_interrupt ();

/**writing the blue image*/
for(int x=0; x<width * height)
    write_mem(blue);
    write_lcd(0x01,0x1);
    wait_interrupt ();

/**writing the red image*/
for(int x=0; x<width * height)
    write_mem(red);
    write_lcd(0x01,0x1);
    wait_interrupt ();
}
}

```



a
ok



b
ok



c



d



Modeling Hardware with 42: Case Study

Embedded Software

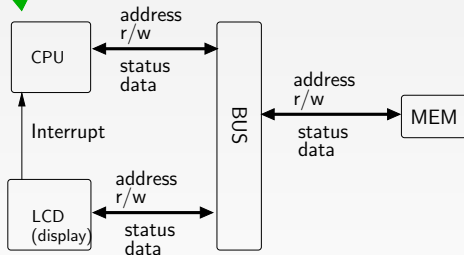
```

#define width 240
#define height 240
#define green 0x0000AABB
...
int main() {
while(1) {
  /*writing the green image*/
  for(int x=0; x<width * height)
    write_mem(green);
  write_lcd(0x01,0x1);
  wait_interrupt();

  /*writing the blue image*/
  for(int x=0; x<width * height)
    write_mem(blue);
  write_lcd(0x01,0x1);
  wait_interrupt();

  /*writing the red image*/
  for(int x=0; x<width * height)
    write_mem(red);
  write_lcd(0x01,0x1);
  // wait_interrupt();
}
}

```



a
ok



b
ok



c
bug
(Synchronization)



d



Modeling Hardware with 42: Case Study

Embedded Software

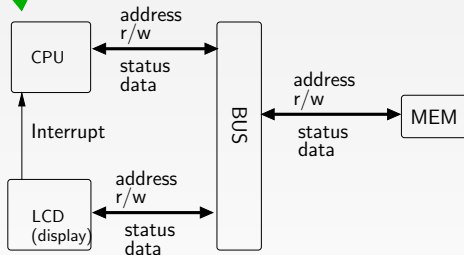
```

#define width 240
#define height 24
#define green 0x0000AABB
...
int main() {
while(1) {
  /*writing the green image*/
  for(int x=0; x<width * height)
    write_mem(green);
  write_lcd(0x01,0x1);
  wait_interrupt();

  /*writing the blue image*/
  for(int x=0; x<width * height)
    write_mem(blue);
  write_lcd(0x01,0x1);
  wait_interrupt();

  /*writing the red image*/
  for(int x=0; x<width * height)
    write_mem(red);
  write_lcd(0x01,0x1);
  // wait_interrupt();
}
}

```



a
ok



b
ok



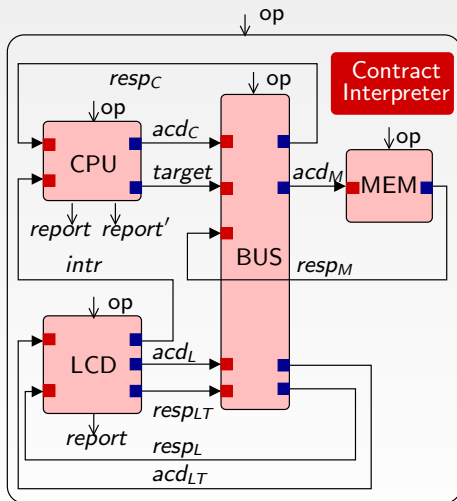
c
bug
(Synchronization)



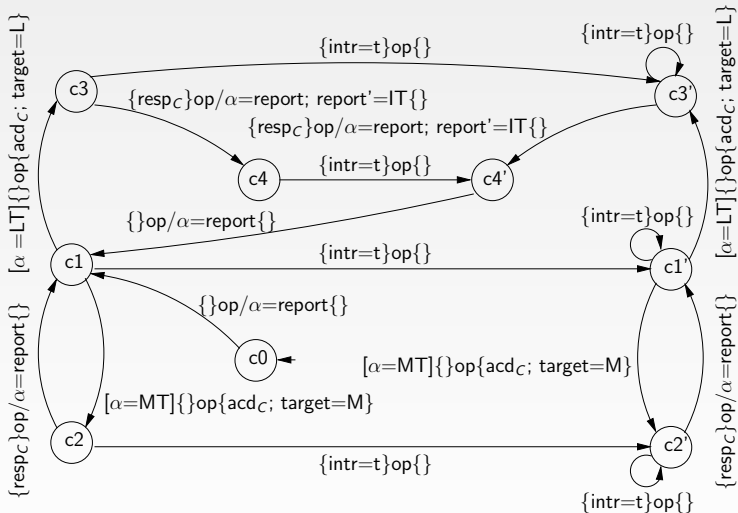
d
bug
(Data)



Modeling Hardware with 42: The 42 Model



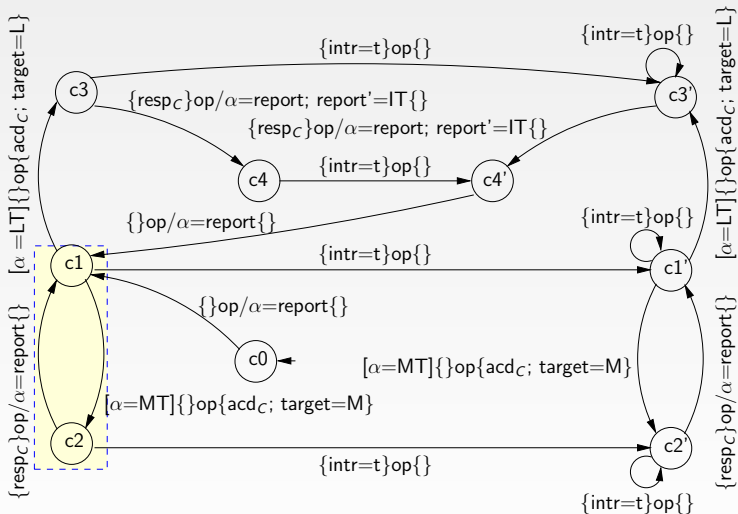
Modeling Hardware with 42: The contracts (CPU)



The contract of the CPU is in fact the contract of (CPU + Software)



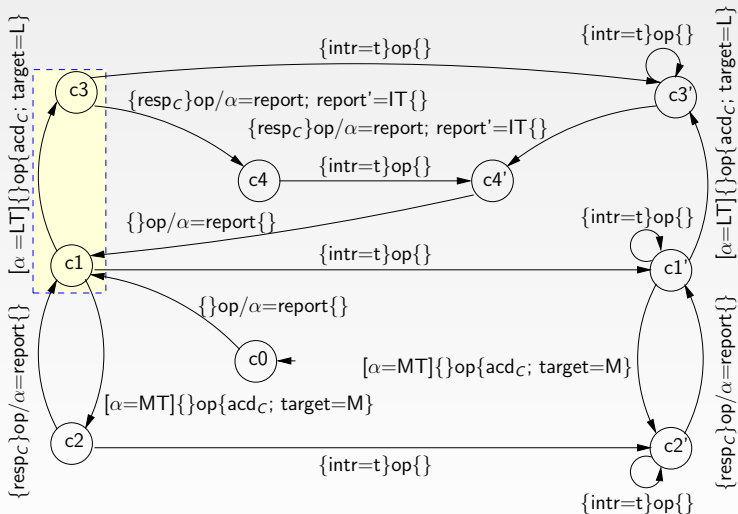
Modeling Hardware with 42: The contracts (CPU)



The contract of the CPU is in fact the contract of (CPU + Software)



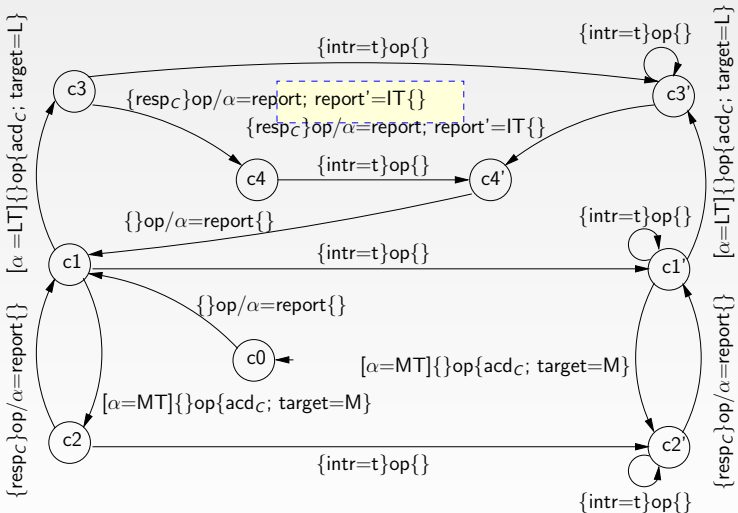
Modeling Hardware with 42: The contracts (CPU)



The contract of the CPU is in fact the contract of (CPU + Software)



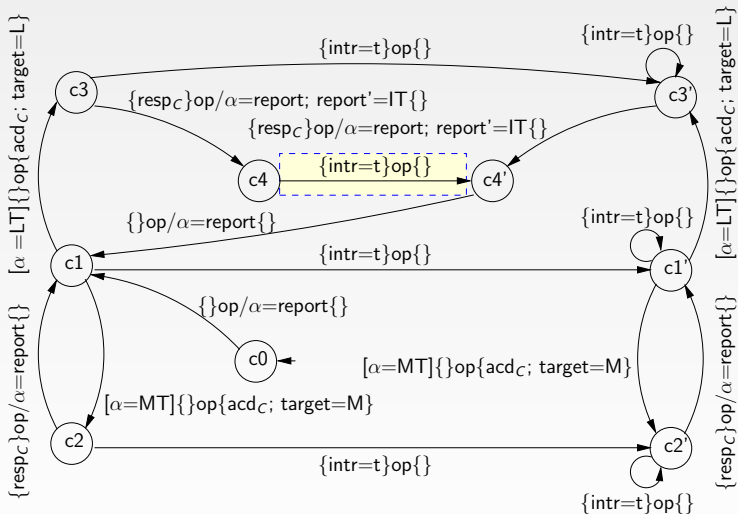
Modeling Hardware with 42: The contracts (CPU)



The contract of the CPU is in fact the contract of (CPU + Software)



Modeling Hardware with 42: The contracts (CPU)



The contract of the CPU is in fact the contract of (CPU + Software)



Detecting Synchronization Bugs by Executing Contracts

If the software doesn't wait for the interrupt \implies



We don't have the software yet. We execute only the contracts.

- **Synchronization Bug:** The contract doesn't wait for interrupts
- **Detection:** Deadlock during contract interpretation



Detecting SW Bugs by Executing the SW Against the Contract

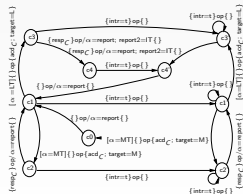
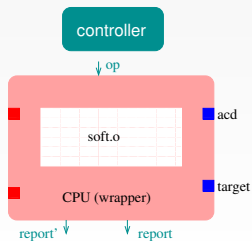
```

a) int main(){
    while (true){
        int x = ...
        while(x>0){
            x - -;
            ...
            b) write_mem(adr, data);
            ...
            c) write_lcd(adr2, data2);
            ...
            e) wait_interrupt();
            if(y!=0)
            f) write_mem(adr, data)
            ...
        }
    }
  
```

software

wrapper

controller



Detecting SW Bugs by Executing the SW Against the Contract

```

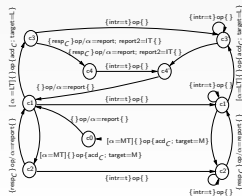
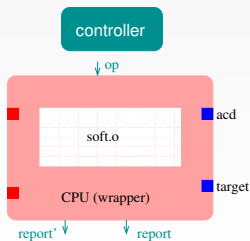
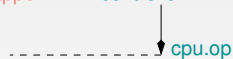
a) int main(){
    while (true){
        int x = ...
        while(x>0){
            x --;
            ...
        }
        b) write_mem(adr, data);
        ...
        c) write_lcd(adr2, data2);
        ...
        e) wait_interrupt();
        if(y!=0)
        f) write_mem(adr, data)
        ...
    }
}

```

software

wrapper

controller

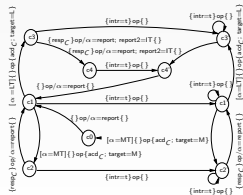
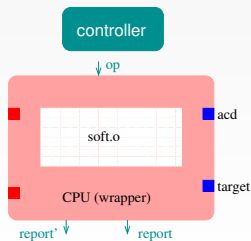
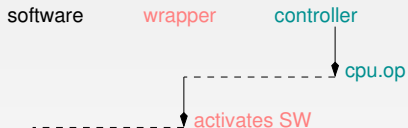


Detecting SW Bugs by Executing the SW Against the Contract

```

a) int main(){
    while (true){
        int x = ...
        while(x>0){
            x --;
            ...
        }
        b) write_mem(adr, data);
        ...
        d) write_lcd(adr2, data2);
        ...
        e) wait_interrupt();
        if(y!=0)
        f) write_mem(adr, data)
        ...
    }
}

```



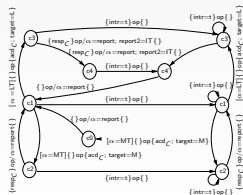
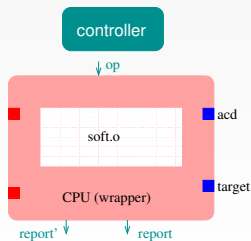
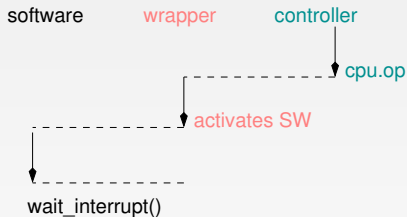
Detecting SW Bugs by Executing the SW Against the Contract

```

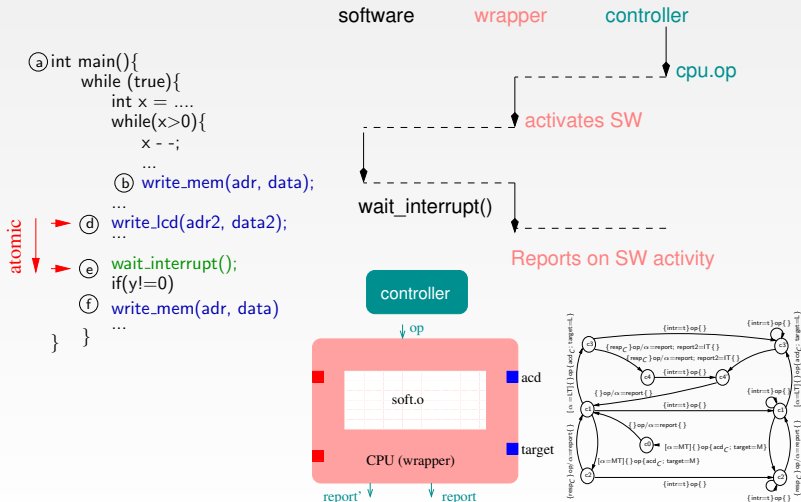
a) int main(){
    while (true){
        int x = ...
        while(x>0){
            x --;
            ...
        }
        b) write_mem(adr, data);
        ...
        d) write_lcd(adr2, data2);
        ...
        e) wait_interrupt();
        if(y!=0)
        f) write_mem(adr, data)
        ...
    }
}

```

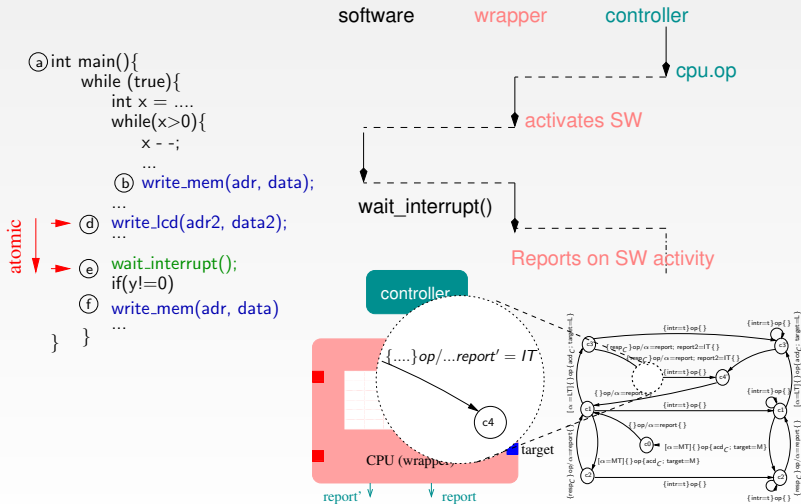
atomic ↓



Detecting SW Bugs by Executing the SW Against the Contract



Detecting SW Bugs by Executing the SW Against the Contract



Summary

- Contracts are **non-deterministic abstractions** of the behavior of components and allow for **early execution**
- Contract execution detects **synchronization bugs**
- Executing contract together with implementation allows to check their **consistency**



Contents

- 1 Introduction & Sources of Inspiration
- 2 Overview of 42 & Examples [GPCE07]
- 3 Hardware Simulation by Interpreting Contracts [COORD09]
- 4 Using 42 Together with Existing Approaches [EMSOFT09]
 - TLM with SystemC
 - 42-ization of SystemC/TLM
 - Typical Uses of 42 Contracts with SystemC-TLM
- 5 Some Related Work
- 6 Summary



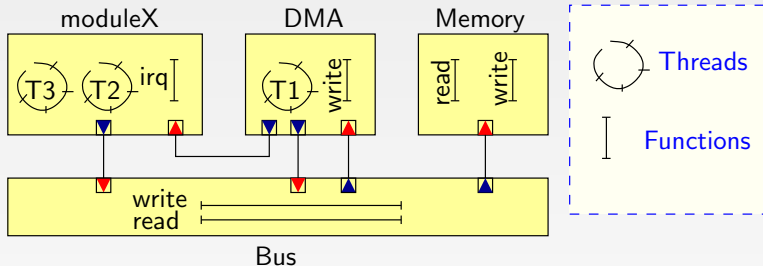
Motivation

- TLM = Transaction-Level Modeling
 - + Widely used
 - + Powerful (large systems, heterogeneous models, ...)
 - + Intrinsically component-based
 - – Very informal, models are mixed with the simulation mechanics

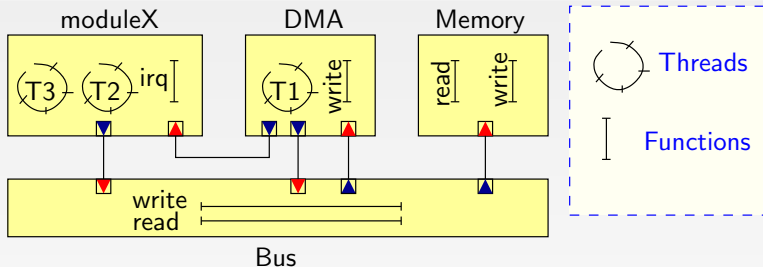
- Motivation
 - Propose a clear definition of TLM components
 - Simplify the simulation
 - Focus on Control Flow and Synchronizations
 - Find bugs earlier



TLM in practice (SystemC)



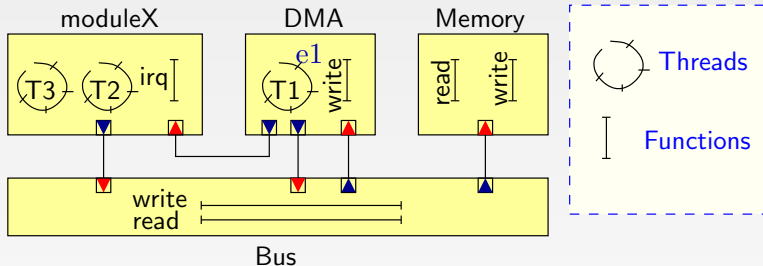
TLM in practice (SystemC)



Some Guidelines:



TLM in practice (SystemC)

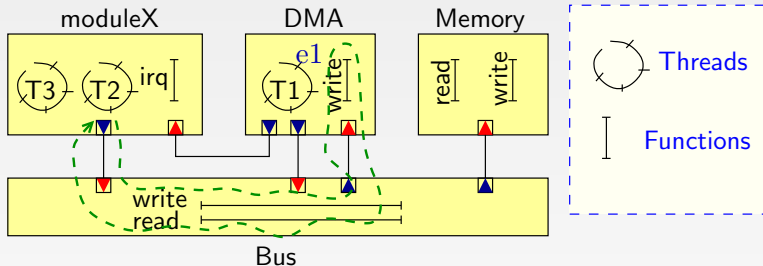


Some Guidelines:

- Inside a module:
 - Shared variables
 - Events (wait, notify)



TLM in practice (SystemC)

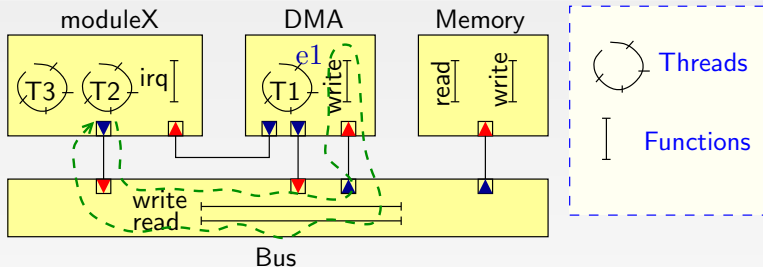


Some Guidelines:

- Inside a module:
 - Shared variables
 - Events (wait, notify)
- Between modules:
 - **Function calls** through ports
 - ...



TLM in practice (SystemC)



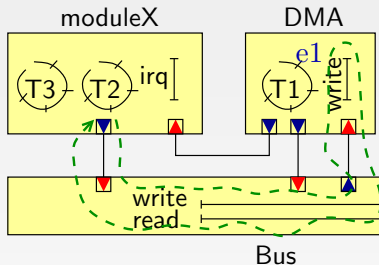
Some Guidelines:

- Inside a module:
 - Shared variables
 - Events (wait, notify)
- Between modules:
 - **Function calls** through ports
 - ...

A non-preemptive scheduler manages the execution of threads



TLM in practice (SystemC)



```

moduleX::T2(){
  while(true){
    x = 42;
    while(x > 0){
      y = p.read(addrM + x)^0xFF;
      p.write(addrM, x);
    }
    p.write(addrD+0x0, addrM);
    p.write(addrD+0x1, 42);
    p.write(addrD+0x4, 0x01);
    wait(e0);
    ...
    e2.notify();
  }
  ...
}

moduleX::irq(int n){
  ...
  e0.notify();
  ...
  wait(e2);
}

```

Example Implementation of *moduleX*

Some Guidelines:

- Inside a module:
 - Shared variables
 - Events (wait, notify)
- Between modules:
 - Function calls through ports
 - ...

A non-ages th



Structural Correspondence

```

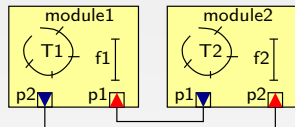
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

```

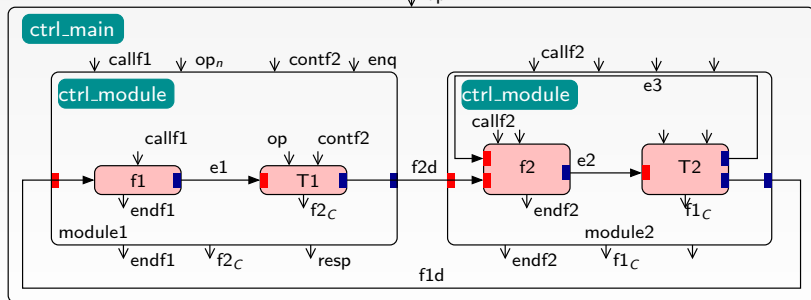
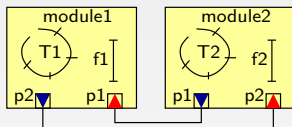
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

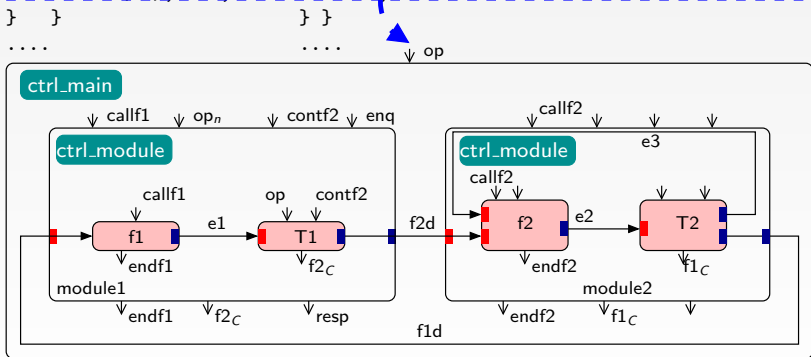
module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

An activation of the main component with **OP** corresponds to what happens in the SystemC-TLM model when the scheduler elects a new thread to execute.



Structural Correspondence

```

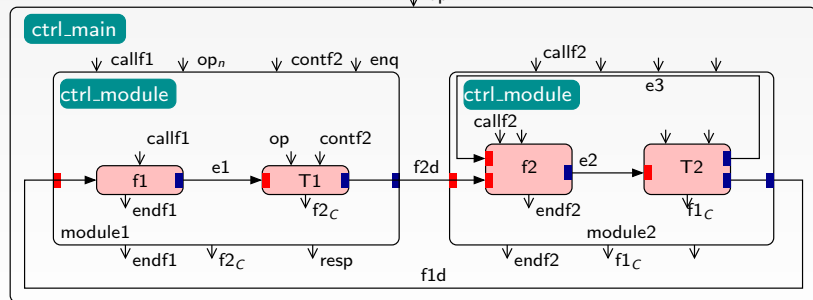
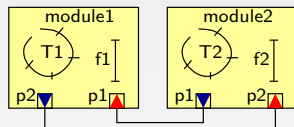
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

```

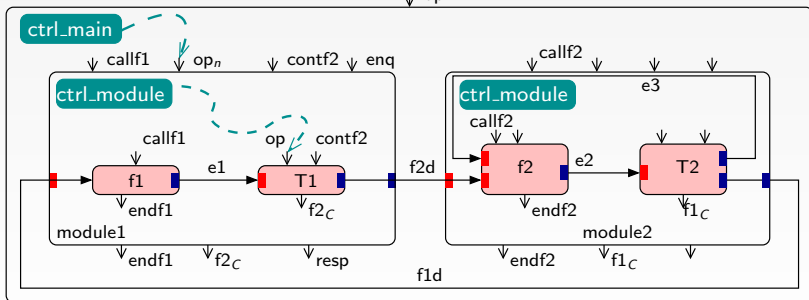
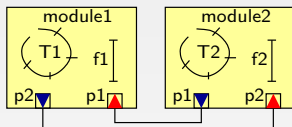
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

```

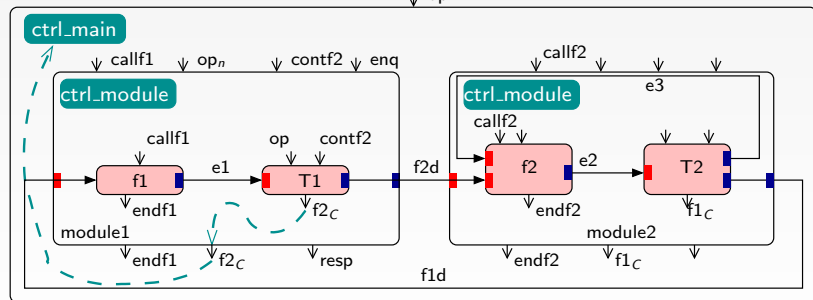
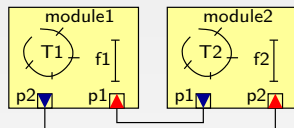
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

```

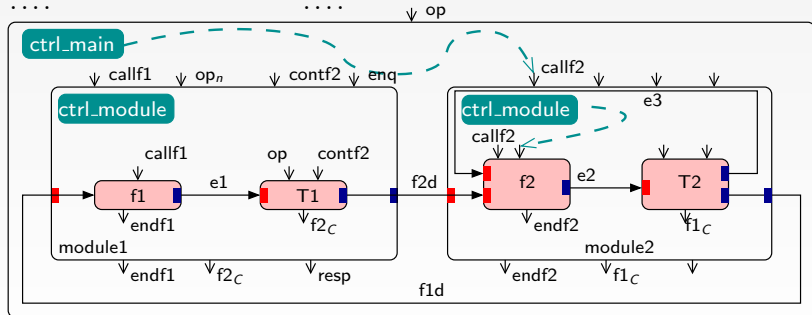
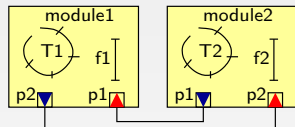
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

▶ module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

```

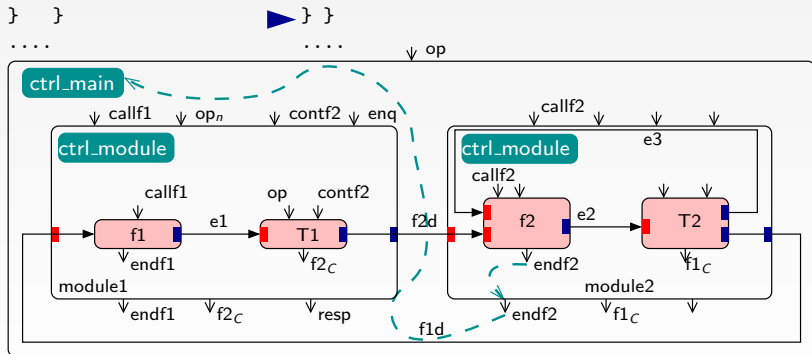
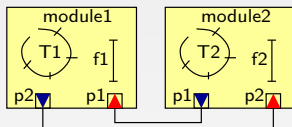
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }

```



Structural Correspondence

```

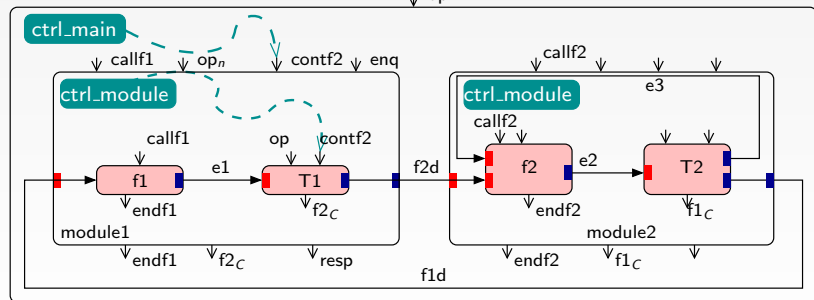
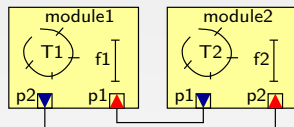
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

```

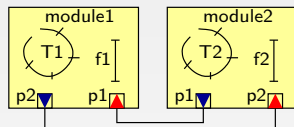
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  }
}

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
}
}

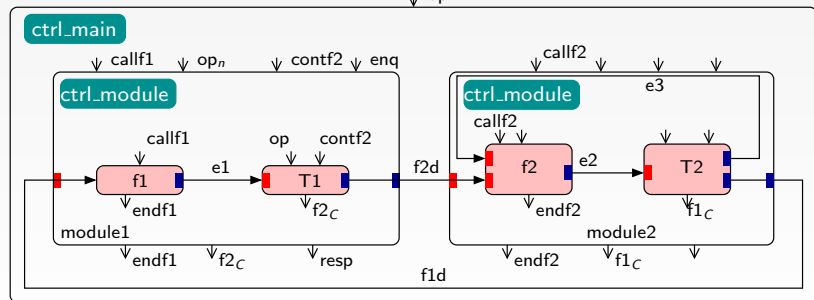
```



....

....

↓ op



Structural Correspondence

```

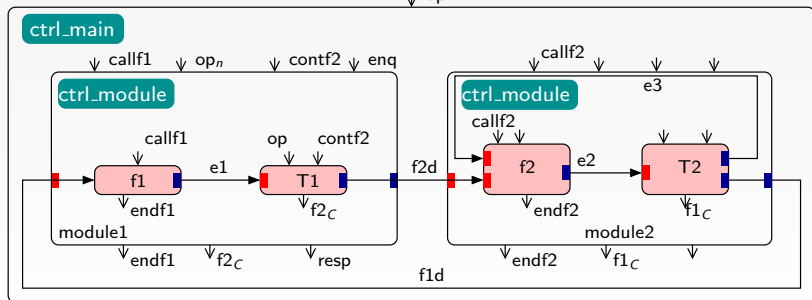
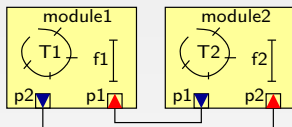
module1::T1(){
  while(true){
    x++;
    ...;
    p2.f2(x);
    ...;
    wait(e1); ...;
  } }
.....

```

```

module2::f2(int x){
  ...;
  y = x * 42;
  ...;
  notify(e2);
  ...;
  ...;
} }
.....

```



Structural Correspondence

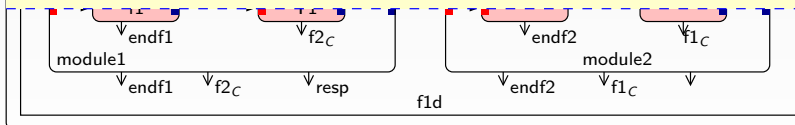
An activation of the main component with **OP** corresponds to what happens in the SystemC-TLM model when the scheduler elects a new thread to execute.

```

} }
....
} }
....
  
```

↓ op

- ++ The 42 model behaves like the SystemC one
- But is less efficient
- ⇒ We will not re-write SystemC in 42
- ⇒ We use 42 control contracts



Extracting Control Contracts From SC-TLM Code

```

a while (true) {
    x++;
    e3.notify();
    e4.notify();
    b wait(e1);
    if (x < 42) {
        c p.f(x);
        d p.g(x);
    }
    y=x+1;
    while (y < 5) {
        e wait(e2);
    }
}

```

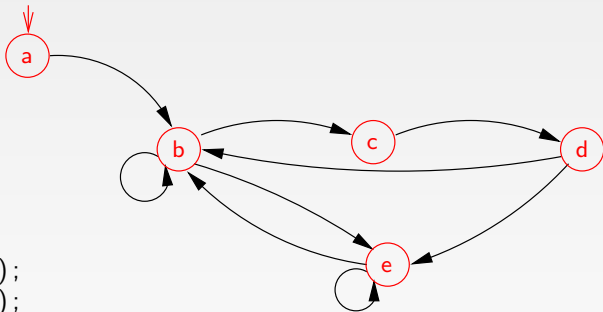
- States correspond to **wait** statements or **function calls**.



Extracting Control Contracts From SC-TLM Code

```

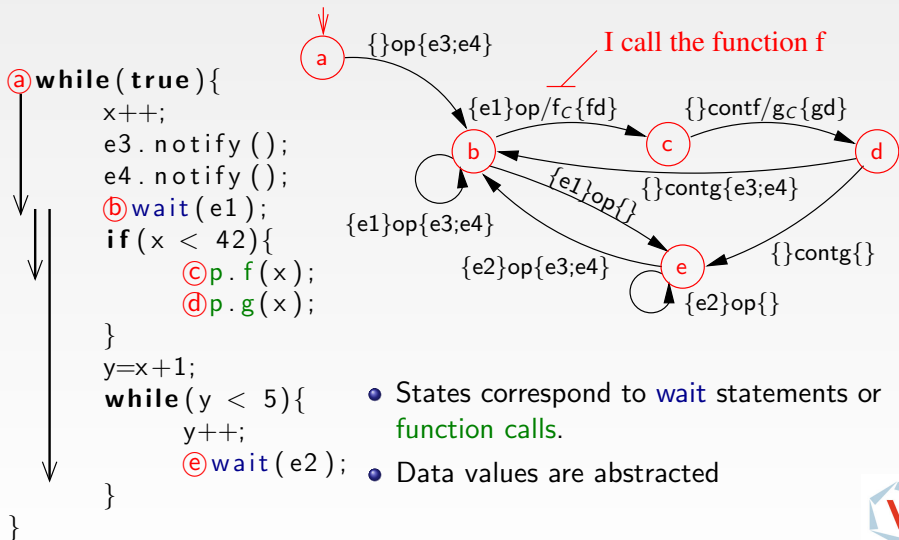
a while( true ){
    x++;
    e3. notify ();
    e4. notify ();
    b wait( e1 );
    if( x < 42 ){
        c p. f( x );
        d p. g( x );
    }
    y=x+1;
    while( y < 5 ){
        y++;
        e wait( e2 );
    }
}
  
```



- States correspond to **wait** statements or **function calls**.
- Data values are abstracted



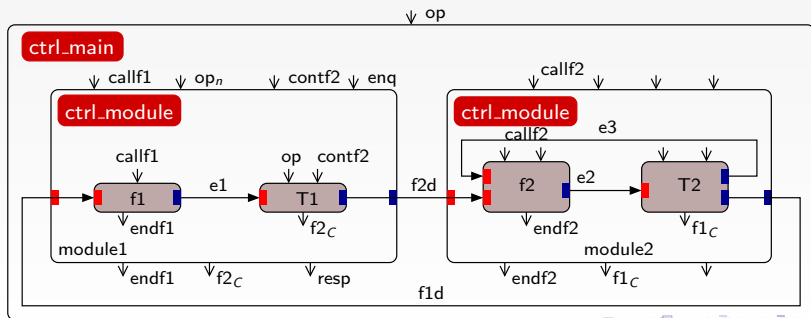
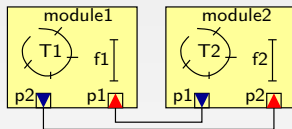
Extracting Control Contracts From SC-TLM Code



- States correspond to **wait** statements or **function calls**.
- Data values are abstracted

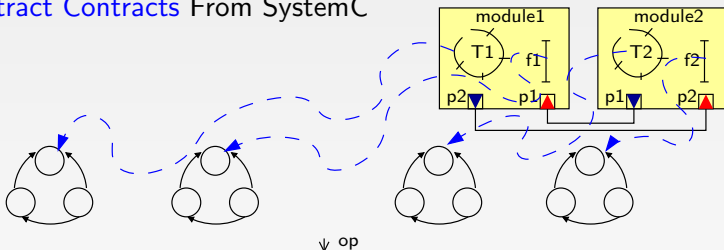


Lightweight Execution for SystemC-TLM

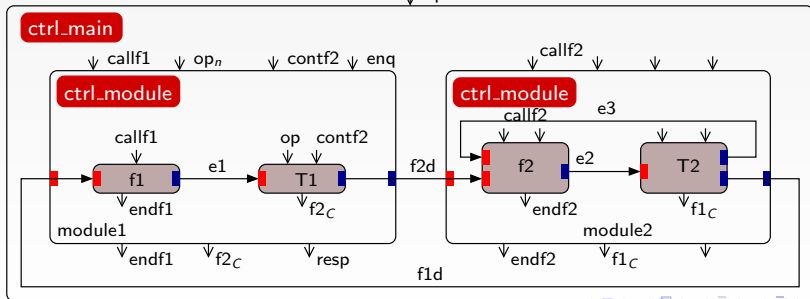


Lightweight Execution for SystemC-TLM

- Extract Contracts From SystemC

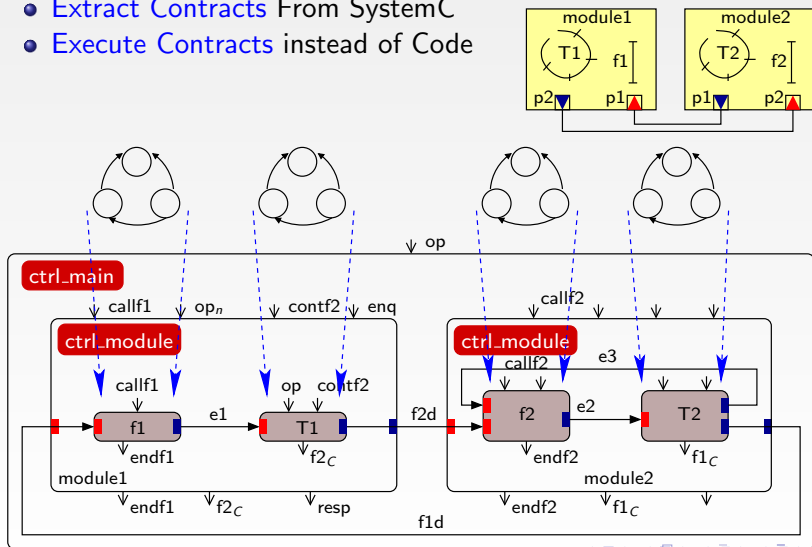


↓ op



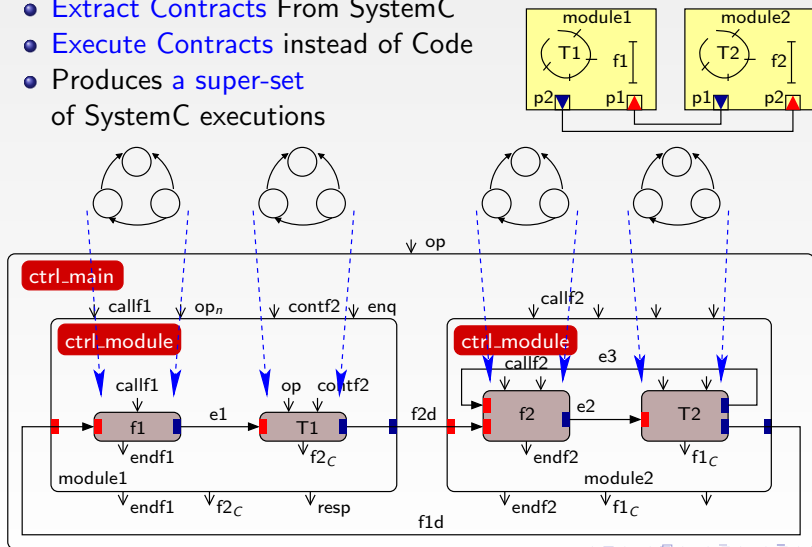
Lightweight Execution for SystemC-TLM

- Extract Contracts From SystemC
- Execute Contracts instead of Code



Lightweight Execution for SystemC-TLM

- Extract Contracts From SystemC
- Execute Contracts instead of Code
- Produces a super-set of SystemC executions

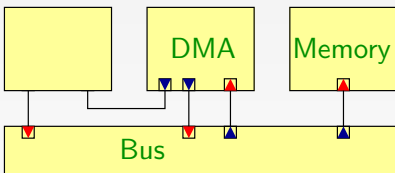


Contents

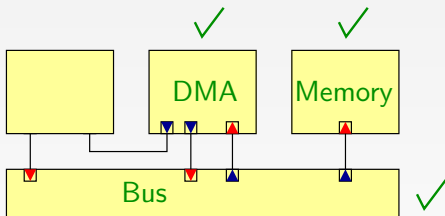
- 1 Introduction & Sources of Inspiration
- 2 Overview of 42 & Examples [GPCE07]
- 3 Hardware Simulation by Interpreting Contracts [COORD09]
- 4 Using 42 Together with Existing Approaches [EMSOFT09]
 - TLM with SystemC
 - 42-ization of SystemC/TLM
 - Typical Uses of 42 Contracts with SystemC-TLM
- 5 Some Related Work
- 6 Summary



TL-Modeling with SystemC: an Example



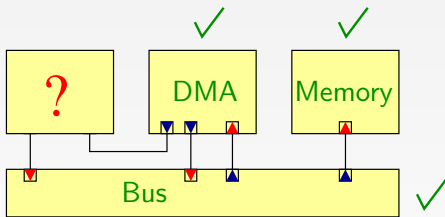
TL-Modeling with SystemC: an Example



- Reuse of **existing components** (e.g., DMA, Memory, Bus)



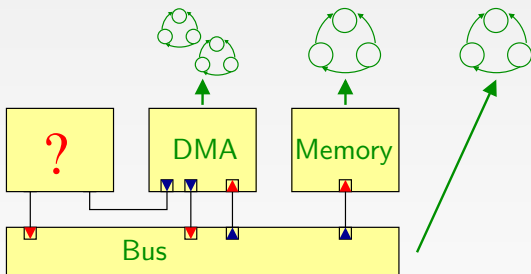
TL-Modeling with SystemC: an Example



- Reuse of **existing components** (e.g., DMA, Memory, Bus)
- Part of the functionality is **not implemented**



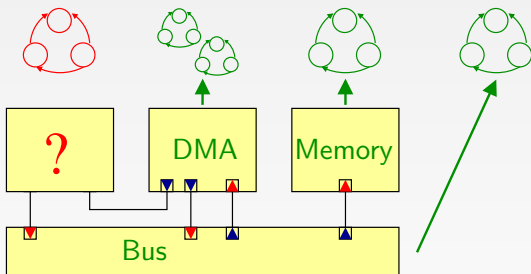
TL-Modeling with SystemC: an Example



- Extract control contracts from existing SC-TLM components



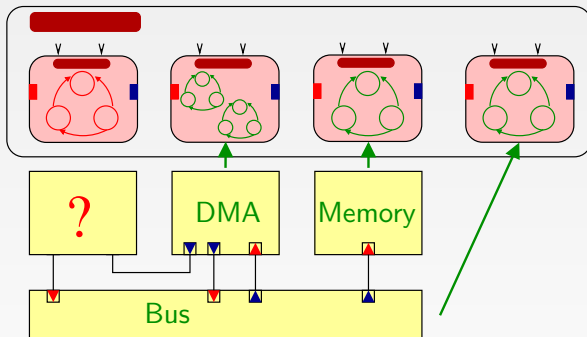
TL-Modeling with SystemC: an Example



- Extract control contracts from existing SC-TLM components
- Write new contracts for the missing ones



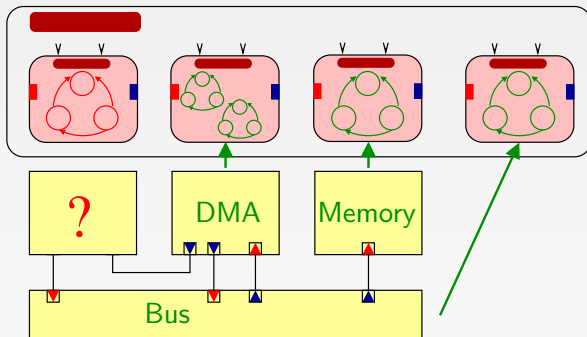
TL-Modeling with SystemC: an Example



- Extract control contracts from existing SC-TLM components
- Write new contracts for the missing ones
- Execute the contracts with 42 controllers



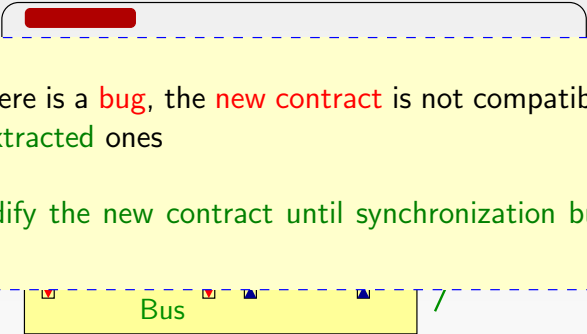
TL-Modeling with SystemC: an Example



- Extract control contracts from existing SC-TLM components
- Write new contracts for the missing ones
- Execute the contracts with 42 controllers
- ⇒ Find Synchronization Bugs



TL-Modeling with SystemC: an Example

- 
- If there is a **bug**, the **new contract** is not compatible with the **extracted** ones
 - **Modify the new contract until synchronization bugs are fixed**

- Extract control contracts from **existing SC-TLM components**
- Write new contracts for the **missing ones**
- Execute the contracts with 42 controllers
- ⇒ Find Synchronization Bugs



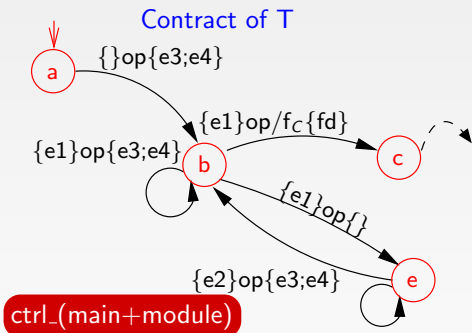
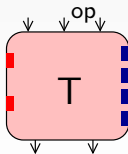
Executing the Implementation against the Contract

Component implementation:

```

.....
module x :: T() {
  @a while (true) {
    x++;
    e3.notify();
    e4.notify();
    @b wait(e1);
    if (x < 42) {
      @c p.f(x);
      @d p.g(x);
    }
    .....
  }
}

```



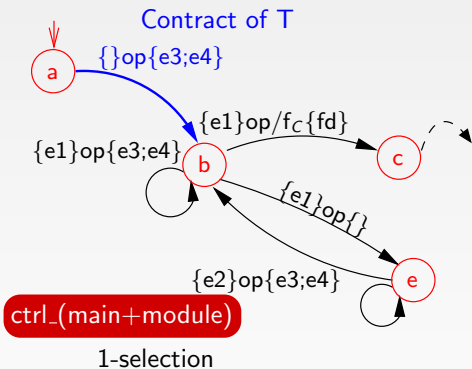
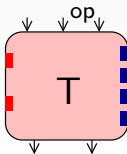
Executing the Implementation against the Contract

Component implementation:

```

....
module x :: T() {
  @a while (true) {
    x++;
    e3.notify();
    e4.notify();
    @b wait(e1);
    if (x < 42) {
      @c p.f(x);
      @d p.g(x);
    }
    ....
  }
}

```



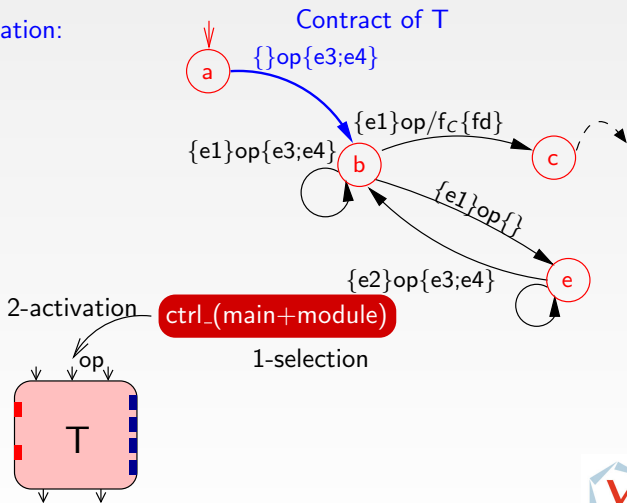
Executing the Implementation against the Contract

Component implementation:

```

.....
module x :: T() {
  @ while (true) {
    x++;
    e3.notify();
    e4.notify();
    @ wait (e1);
    if (x < 42) {
      @ p.f(x);
      @ p.g(x);
    }
  }
  .....
}

```



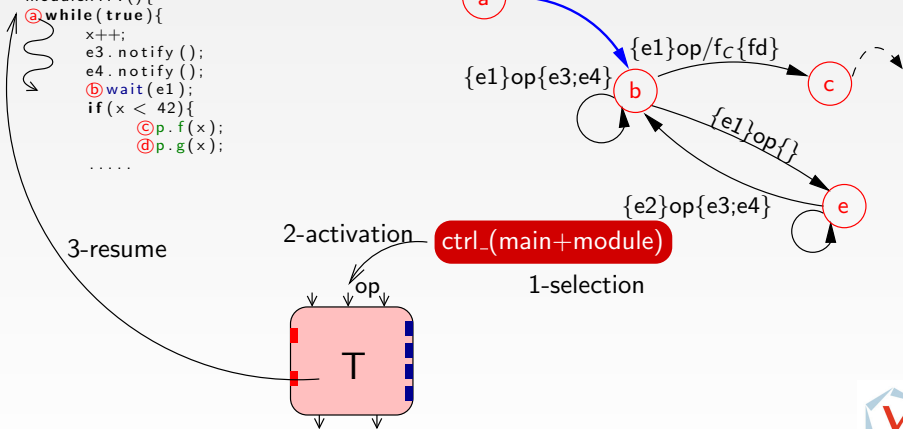
Executing the Implementation against the Contract

Component implementation:

```

.....
module x :: T() {
  @while (true) {
    x++;
    e3.notify();
    e4.notify();
    @wait (e1);
    if (x < 42) {
      @p.f(x);
      @p.g(x);
    }
    .....
  }
}

```



Executing the Implementation against the Contract

Component implementation:

```

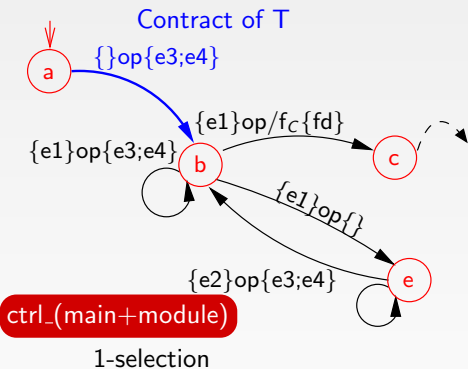
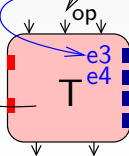
.....
module x :: T() {
  @while (true) {
    x++;
    e3.notify();
    e4.notify();
    @wait (e1);
    if (x < 42) {
      @p.f(x);
      @p.g(x);
    }
    .....
  }
}

```

...notify()

3-resume

2-activation



Executing the Implementation against the Contract

Component implementation:

```

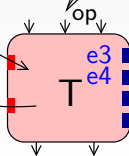
.....
module x :: T() {
  @while (true) {
    x++;
    e3.notify();
    e4.notify();
    @wait (e1);
    if (x < 42) {
      @p.f(x);
      @p.g(x);
    }
    .....
  }
}

```

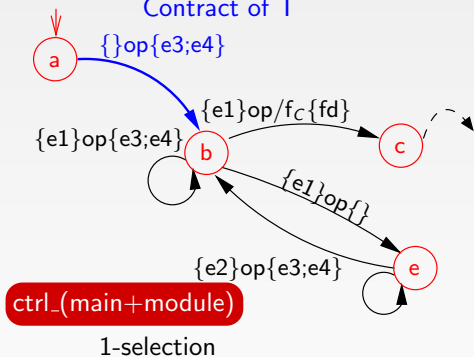
4-wait(...)
4-suspend

3-resume

2-activation



Contract of T



Executing the Implementation against the Contract

Component implementation:

```

.....
module x :: T() {
  @a while (true) {
    x++;
    e3.notify();
    e4.notify();
    @b wait(e1);
    if (x < 42) {
      @c p.f(x);
      @d p.g(x);
    }
    .....
  }
}

```

4-wait(...)
4-suspend

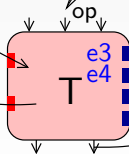
3-resume

2-activation

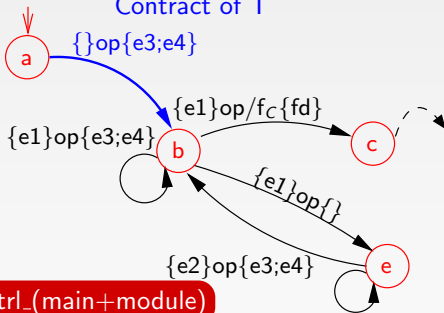
ctrl_(main+module)

1-selection

5-op finished



Contract of T



Executing the Implementation against the Contract

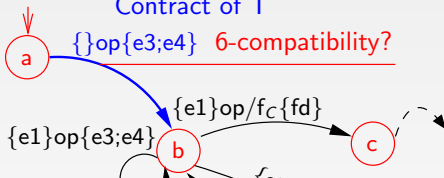
Component implementation:

```

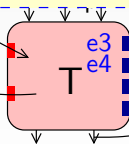
.....
module x :: T() {
  @while (true) {
    x++;
    e3.notify();
    e4.notify();
    @wait(e1);
    if (x < 42) {

```

Contract of T



This execution step should notify *e3* and *e4* (calls to *e.notify()*). Otherwise, the implementation does not respect the contract.



5-op finished



Summary

- Use of 42 with an existing approach (TLM)
- Separation of the semantics of models from the execution mechanics
- Support for reasoning on component synchronizations
- A tool for extracting control contracts from SystemC code [Master 1, P. Delahaye, 2010]
- Possibility of connection to verification tools



Contents

- 1 Introduction & Sources of Inspiration
- 2 Overview of 42 & Examples [\[GPCE07\]](#)
- 3 Hardware Simulation by Interpreting Contracts [\[COORD09\]](#)
- 4 Using 42 Together with Existing Approaches [\[EMSOFT09\]](#)
- 5 **Some Related Work**
- 6 Summary



Some Related Work

● Ptolemy

42

- Simulation tool for heterogeneous E.S.
- Catalogue of predefined MoCCs
- Modeling discrete/continuous systems

Semantics of heterogeneity
 Programmable MoCCs
 focuses on discrete ones

● Fractal

- Language independent component-based approach
- Dedicated to design and deployment
- Does not deal with MoCCs

Similar
 Virtual Prototyping
 Central notion

● Formal Models for Embedded Systems

- Formal Verification \implies Reduced expressiveness Favor expressiveness
- Describing MoCCs and Heterogeneity with TAG machines

Operational description of MoCCs



Contents

- 1 Introduction & Sources of Inspiration
- 2 Overview of 42 & Examples [GPCE07]
- 3 Hardware Simulation by Interpreting Contracts [COORD09]
- 4 Using 42 Together with Existing Approaches [EMSOFT09]
- 5 Some Related Work
- 6 Summary**

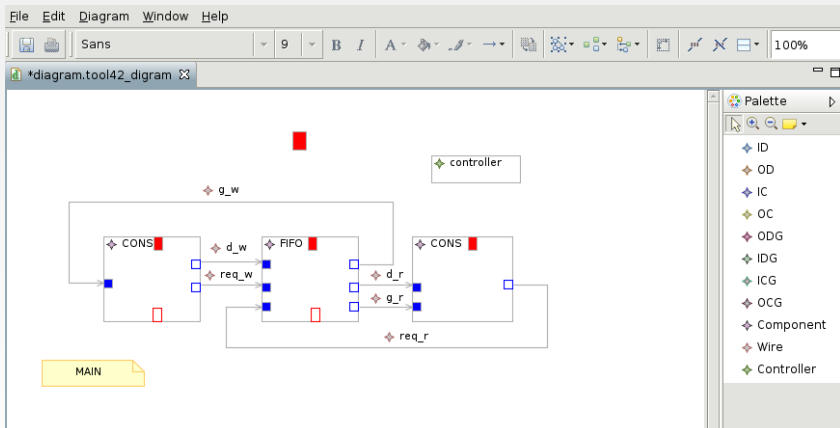


Contributions

- A **language-independent** component-based model for heterogeneous embedded systems, and **reuse of existing code**
- An **executable specification** language for components
- A rich set of examples for modeling **MoCCs and heterogeneity**
- A complete case-study on the use of **42 with existing approaches**



A tool for describing/executing 42 models



Future Work

- Extending the semantics of 42 with a quantitative notion of time
- Investigate on expressiveness and readability of contracts
- Modeling non-functional aspects and their relation with functional models [MOCCs'08]



Communication on 42



Tayeb Bouhadiba and Florence Maraninchi, *Contract-based coordination of hardware components for the development of embedded software*, COORDINATION, Lecture Notes in Computer Science, vol. 5521, Springer, 2009, pp. 204–224.



Tayeb Bouhadiba, Florence Maraninchi, and Giovanni Funchal, *Formal and executable contracts for transaction-level modeling in systemc*, EMSOFT , ACM, 2009, pp. 97–106.



Florence Maraninchi and Tayeb Bouhadiba, *42: programmable models of computation for a component-based approach to heterogeneous embedded systems*, GPCE, ACM, 2007, pp. 53–62.



Florence Maraninchi and Tayeb Bouhadiba, *42: programmable models of computation for a component-based approach to heterogeneous embedded systems*, In SYNCHRON'07 International Open Workshop on Synchronous Programming.



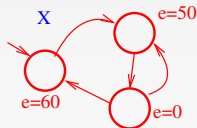
Tayeb Bouhadiba, Florence Maraninchi, Karine Altisen, Matthieu Moy. *Computational Modeling of Non-Functional Properties with the Component Model 42*, Position paper for the ARTIST MoCC'08 Workshop, july 2008, Eindhoven



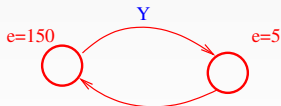
Computational Modeling of Energy Consumption (Principle)

N-Functional

Radio consumption

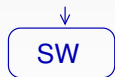
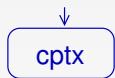


CPU consumption



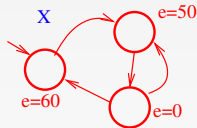
Computational Modeling of Energy Consumption (Principle)

Functional

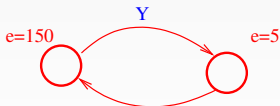


N-Functional

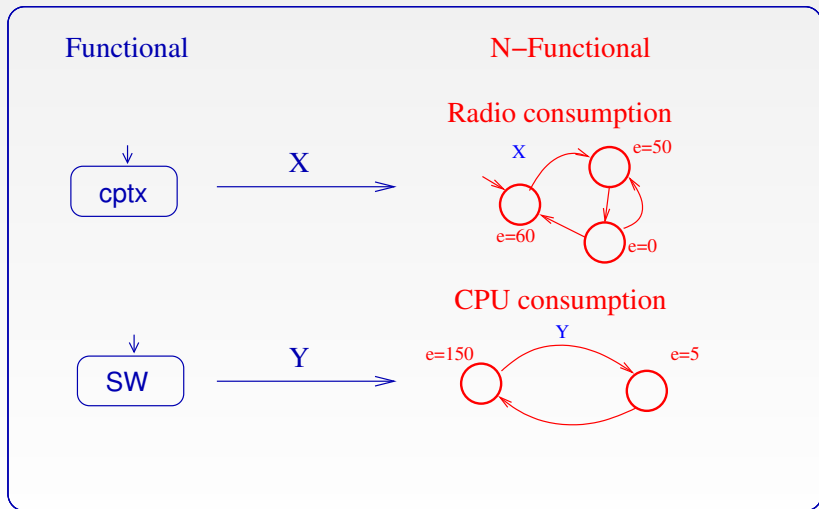
Radio consumption



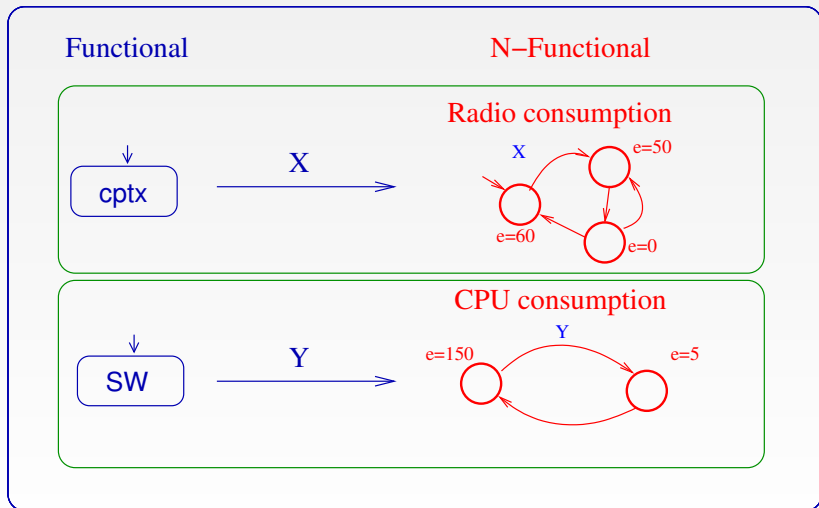
CPU consumption



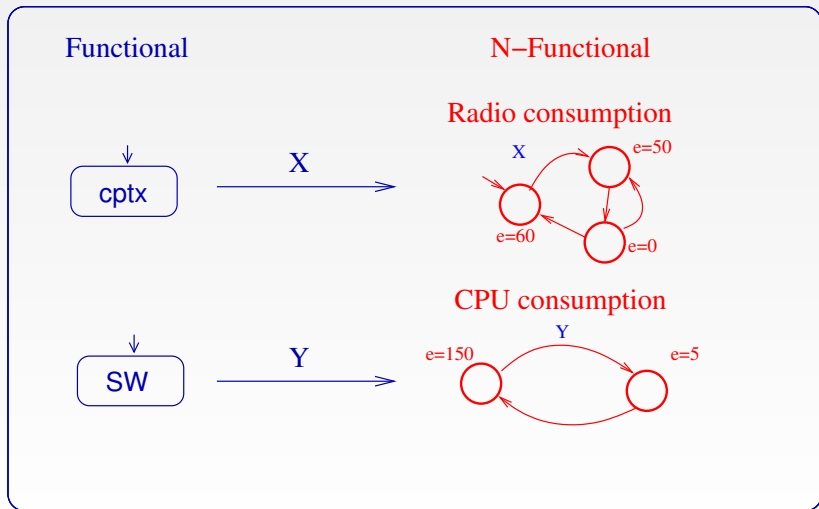
Computational Modeling of Energy Consumption (Principle)



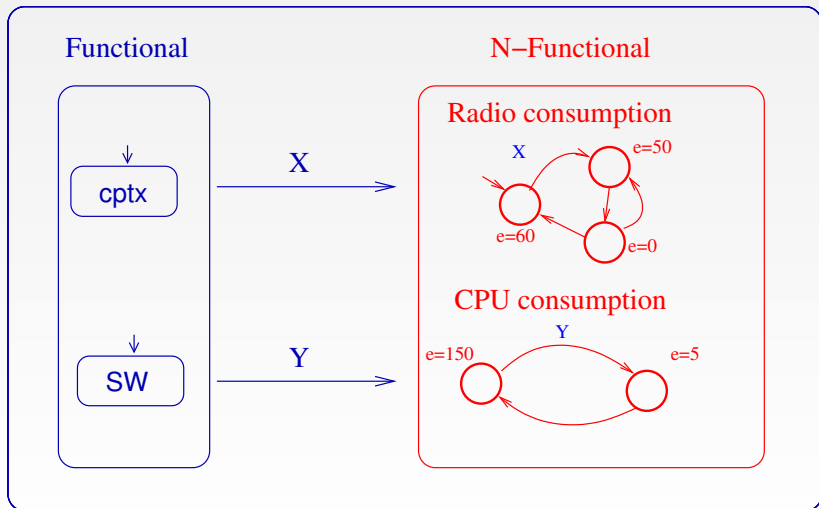
Computational Modeling of Energy Consumption (Principle)



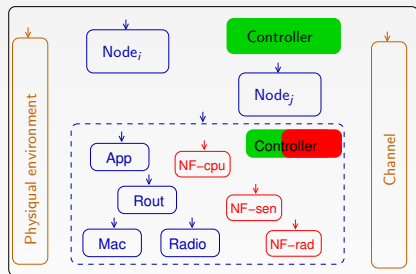
Computational Modeling of Energy Consumption (Principle)



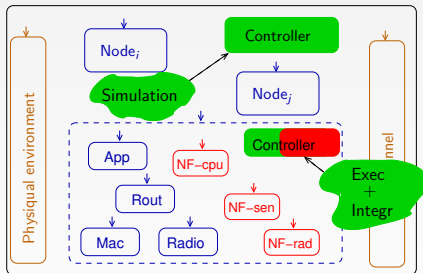
Computational Modeling of Energy Consumption (Principle)



Integrating Functional/Non-Functional Model (option 1)



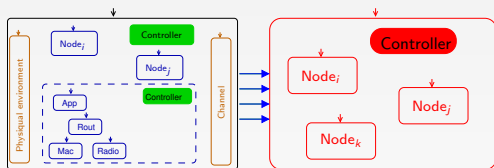
Integrating Functional/Non-Functional Model (option 1)



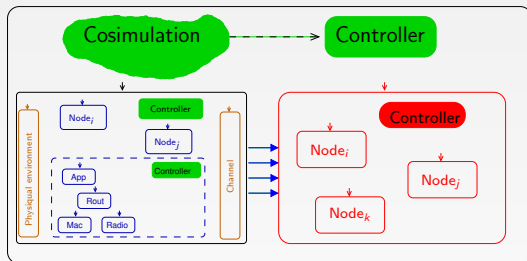
- The simulation **MoCC** is still the same.
- No separation between **Non Functional / Functional**
- Complicated **MoCC**.
- Each Component (node) will always be run as Func/N-Func component.



Integrating Functional/Non-Functional Model (option 2)



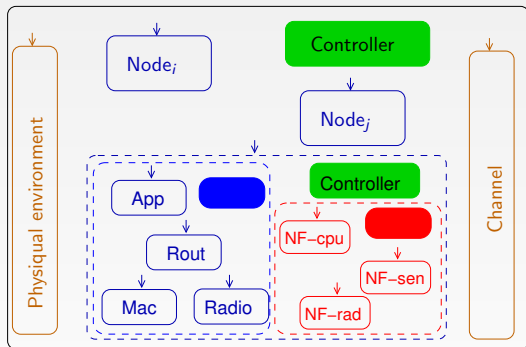
Integrating Functional/Non-Functional Model (option 2)



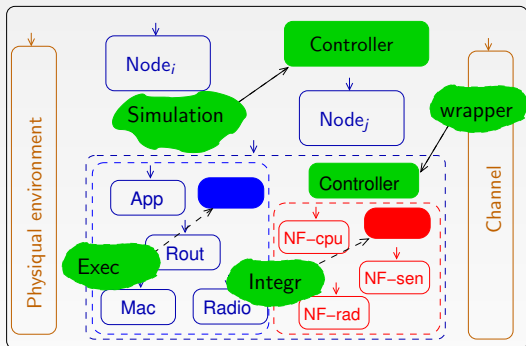
- Very clear separation **Func/N-Func**.
- Duplication of hierarchy (hard to maintain).
- To many wires to manage.
- Would be well suited for other applications.



Integrating the Functional/Non-Functional Model (option 3)



Integrating the Functional/Non-Functional Model (option 3)



- Identification of **Func/N-Func** models.
- **Functional** models still unchanged.
- This structuring is well suited for WSN applications.
- Not adequate for applications where **Energy** models share resources.

