

# Separate Compilation for Synchronous Programs

## SCOPES 2009

Jens Brandt and Klaus Schneider

Embedded Systems Group  
Department of Computer Science  
University of Kaiserslautern

April 2009

# Outline

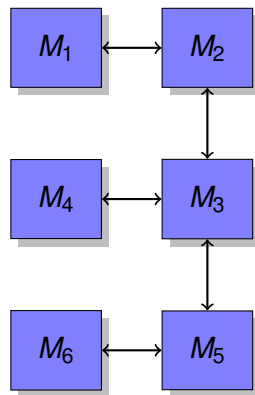
- 1 Outline
- 2 Contribution
  - Separate
  - Compilation
  - for Synchronous Programs
- 3 Challenges
  - Context
  - Delayed Assignments
  - Schizophrenia
- 4 Summary

# Contribution (1/3)

## Separate Compilation for Synchronous Programs

# Symmetric Linking

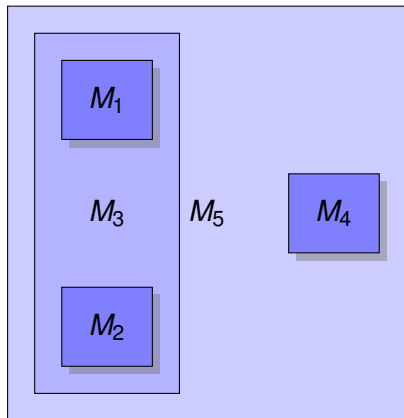
- modules: **independent components**
- defined by **behavioural** part assembled by **structural** part
- modules run in the same context



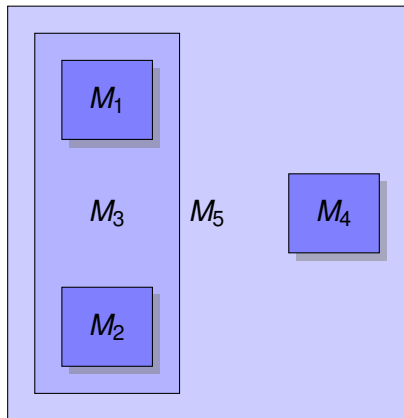
⇒ VHDL, SDL and threads in all classical imperative programming languages

# Asymmetric Linking

- modules have contexts
- module calls in behavioural part of the language
- behavioural hierarchy



# Incremental vs. Separate Compilation



## incremental compilation

- use previously compiled inner module when outer one is compiled

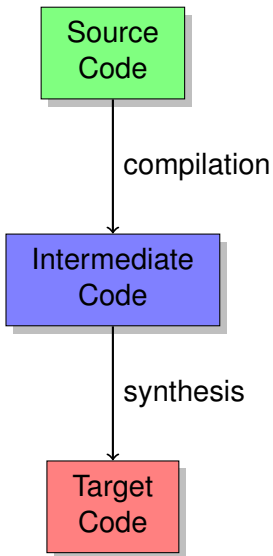
## separate compilation

- compile module without any knowledge about called modules
- compile all modules in **arbitrary order** and link them afterwards

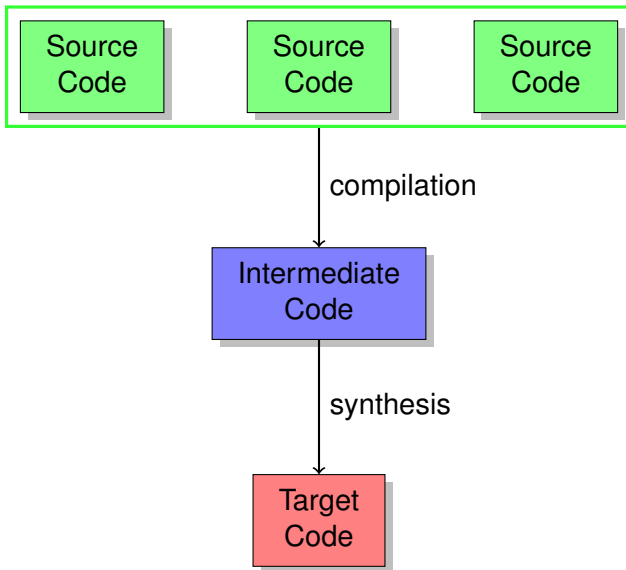
## Contribution (2/3)

Separate **Compilation**  
for Synchronous Programs

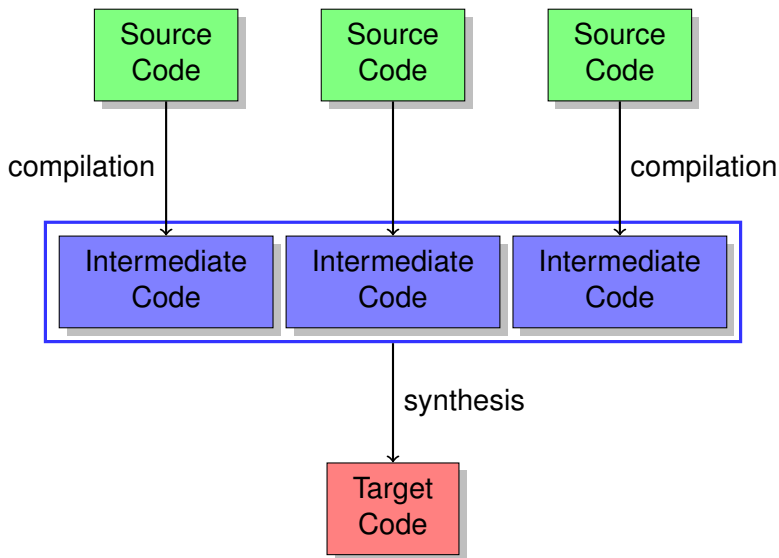
# Compilation and Synthesis



# Non-Modular Compilation

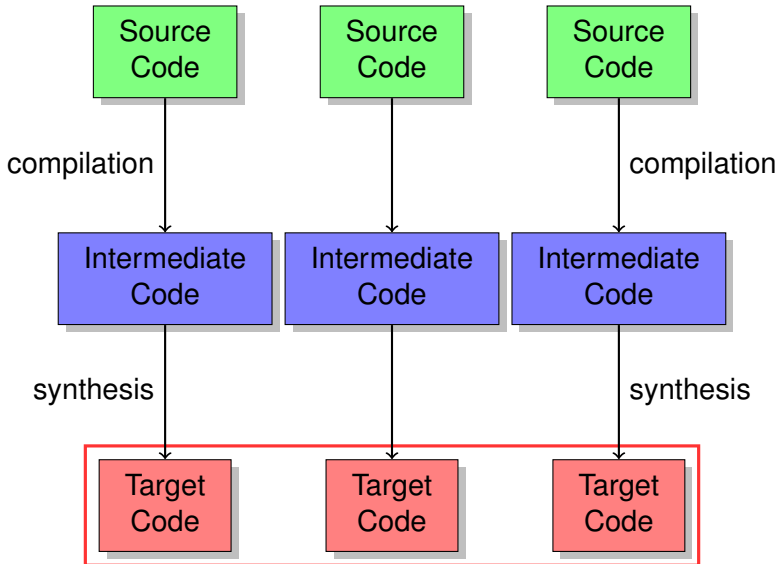


# Modular Compilation

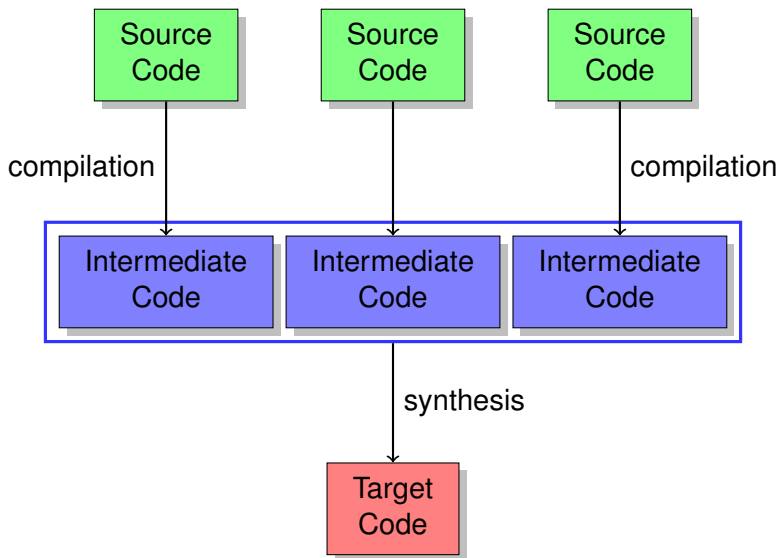




# Modular Synthesis



# Modular Compilation



# Contribution (3/3)

## Separate Compilation for Synchronous Programs

# Synchronous Programs

## Example

```

{
  b = true;
  l1 : pause;
  if(a) b = false;
  l2 : pause;
} || {
  l3 : pause;
  if( $\neg b$ ) c = true;
  a = true;
  l4 : pause;
  b = true;
}

```

- synchronous languages
  - Esterel
  - Lustre
  - Quartz
  - ...
- step 0  $\{ \}$ :  
 $a = \text{false}, b = \text{true}, c = \text{false}$
- step 1  $\{l_1, l_3\}$ :  
 $a = \text{true}, b = \text{false}, c = \text{true}$
- step 2  $\{l_2, l_4\}$ :  
 $a = \text{true}, b = \text{true}, c = \text{true}$

# The Synchronous Language Quartz

## Core Statements

- `nothing;`
- `x =  $\tau$ ; and next(x) =  $\tau$ ;`
- `l : pause;`
- `if ( $\sigma$ ) S1 else S2;`
- `S1 S2`
- `do S while( $\sigma$ );`
- `during S1 do S2`
- `S1 || S2`
- `[weak] abort S when [immediate]( $\sigma$ );`
- `[weak] suspend S when [immediate]( $\sigma$ );`
- `{ $\alpha$  x; S}`
- `name( $\tau_1, \dots, \tau_n$ );`

# Intermediate Code

## Guarded Actions

$\gamma \Rightarrow x = \tau$  (immediate action)

$\gamma \Rightarrow \text{next}(x) = \tau$  (delayed action)

- intermediate code: set of synchronous **guarded actions**
  - in each macro step, check all guards are simultaneously
  - if a guard is true, its action is immediately executed
  - value is assigned in current or next step
  - if no action can be fired, execute **absence reaction**
  - generated for data flow and control flow

# Intermediate Code: Example

## Example

```

module  $M_1$ (bool ? $i$ , bool ! $o$ ){
  if( $i$ ) {
     $\ell$  : pause;
     $o = i$ ;
  } else {
    next( $o$ ) = true;
  }
}

```

- guarded actions for  $M_1$ :  
(simplified)

$$start \wedge i \Rightarrow (\text{next}(\ell) = \text{true})$$

$$\ell \Rightarrow (o = i)$$

$$start \wedge \neg i \Rightarrow (\text{next}(o) = \text{true})$$

$$\text{abs}_{\text{init}}(o) = \text{false}$$

$$\text{abs}_{\text{trans}}(o) = o$$

# Contribution

- first separate compilation procedure for synchronous programs
- implemented within our **Averest System**
- available at `http://www.averest.org`
- major revision 2.0 almost available

# Outline

- 1 Outline
- 2 Contribution
  - Separate
  - Compilation
  - for Synchronous Programs
- 3 Challenges**
  - Context
  - Delayed Assignments
  - Schizophrenia
- 4 Summary

# Challenge 1: Context

## Example

```
abort
  if(i) {
    ℓ : pause;
    o = i;
  } else {
    next(o) = true;
  }
when(b);
```

- guards of inner module depend on starting and preemption conditions of outer module
- guards of outer module depend on termination of inner module

# Challenge 1: Context

## Example

```
abort
   $M_1(i, b)$ ;
when( $b$ );

module  $M_1(\text{bool } ?i, \text{bool } !o)\{
  \text{if}(i) \{
    \ell : \text{pause};
     $o = i$ ;
  \} \text{else} \{
    next( $o$ ) = true;
  \}
}$ 
```

- guards of inner module depend on starting and preemption conditions of outer module
- guards of outer module depend on termination of inner module

# Context Interface

## Inputs

- $\text{go}(M)$ : start module
- $\text{abrt}(M)$ : abort control-flow
- $\text{susp}(M)$ : suspend control-flow
- $\text{prmt}(M)$ : preempt data-flow

## Outputs

- $\text{inst}(M)$ :  $M$  is instantaneous now
- $\text{insd}(M)$ :  $M$  is active now
- $\text{term}(M)$ :  $M$  terminates voluntarily now

# Context Interface: Example

## Example

```

module  $M_1$  (bool ? $i$ , bool ! $o$ ) {
  if ( $i$ ) {
     $l$  : pause;
     $o = i$ ;
  } else {
    next ( $o$ ) = true;
  }
}

```

$$\begin{aligned}
\text{go}(M_1) \wedge i &\Rightarrow \text{next}(l) = \text{true} \\
\text{go}(M_1) \wedge \neg i &\Rightarrow \text{next}(o) = \text{true} \\
l \wedge \text{susp}(M_1) &\Rightarrow \text{next}(l) = \text{true} \\
l \wedge \neg \text{prmt}(M_1) &\Rightarrow o = i
\end{aligned}$$

$$\begin{aligned}
\text{inst}(M_1) &= \neg i \\
\text{insd}(M_1) &= l_1 \\
\text{term}(M_1) &= l_1
\end{aligned}$$

## Challenge 2: Delayed Assignments

### Example

```
{  
  bool b;  
  :  
  if(i) {  
    ℓ : pause;  
    b = i;  
  } else {  
    next(b) = true;  
  }  
}
```

- problem: delayed assignment to output variable in last step must be deactivated

## Challenge 2: Delayed Assignments

### Example

```
{
  bool b;
  ⋮
   $M_1(i, b)$ ;
}

module  $M_1(\mathbf{bool} ?i, \mathbf{bool} !o)\{
  \mathit{if}(i) \{
    \ell : \mathit{pause};
    o = i;
  } \mathit{else} \{
     $\mathit{next}(o) = \mathit{true}$ ;
  }
}$ 
```

- problem: delayed assignment to output variable in last step must be deactivated

# Deactivation of Delayed Assignments

- delayed assignments in last step of scope must be deactivated
- deactivation cannot be made when compiling the module
- outer module does not know the actions of the inner one
- [defer deactivation to the linker](#)
- endow all argument of calls by activation conditions
- will be finally added to the linked module

## Challenge 3: Schizophrenia

### Example

```
loop {  
  int n;  
  if(i) n = 1;  
  ℓ : pause;  
  if(i) n = 2;  
  o = n;  
}
```

- synchronous paradigm:  
variable has a single value  
in each step
- scope of variables can be  
left and reentered in the  
same macro step
- several incarnations of  
same variable needed

## Challenge 3: Schizophrenia

### Example

```
loop {  
   $M_2(i, n)$ ;  
   $o = n$ ;  
}
```

```
module  $M_2(\text{bool } ?i, \text{int } !o)\{\$   
   $\text{if}(i) o = 1$ ;  
   $\ell : \text{pause}$ ;  
   $\text{if}(i) o = 2$ ;  
}
```

- synchronous paradigm:  
variable has a single value  
in each step
- scope of variables can be  
left and reentered in the  
same macro step
- several incarnations of  
same variable needed

## Challenge 3: Schizophrenia (continued)

- split module into **surface** (initial step) and **depth** (remaining steps)
- required reincarnations are made by generating copies of the module's surface
- multi-stage numbering of the incarnations needed
- encode **transfers** into absence reactions

# Example: Schizophrenia

## Example (Schizophrenia)

```

module  $M_3$ (bool ? $i$ , int ! $o$ ){
  if( $i$ )  $o = 1$ ;
   $l$  : pause;
  if( $i$ )  $o = 2$ ;
}

```

```

module  $M_4$ (bool ? $i$ , int ! $o$ ){
  loop {
    int  $n$ ;
    weak abort
       $M_3(i, n)$ ;
    when( $n < 1$ );
     $l$  : pause;
    if( $i$ )  $n = 3$ ;
    next( $o$ ) =  $n$ ;
  }
}

```

- $\text{surface}(M_3)$ :

$$\begin{aligned} \text{go}^s(M_3) \wedge i &\Rightarrow o = 1; \\ \text{go}^d(M_3) &\Rightarrow l = \text{true}; \end{aligned}$$

- $\text{depth}(M_3)$ :

$$l \wedge i \Rightarrow o = 2;$$

- $\text{surface}(M_4)$ :

$$\text{go}^d(M_4) \Rightarrow \text{next}(l) = \text{true}$$

- $\text{depth}(M_4)$ :

$$\begin{aligned} l \wedge \neg \text{abrt}(M_4) \wedge \text{term}(M_3) \vee (n@1 < 1) \\ \Rightarrow \text{next}(l) = \text{true} \end{aligned}$$

$$l \wedge i \wedge \neg \text{prmt}(M_4) \Rightarrow n = 3$$

$$l \wedge \neg \text{prmt}(M_4) \Rightarrow \text{next}(o) = n$$

$$\text{abs}_{\text{trans}}(n) = \text{go}^d(M_4) \text{ ? } n@1 : 0$$

# Outline

- 1 Outline
- 2 Contribution
  - Separate
  - Compilation
  - for Synchronous Programs
- 3 Challenges
  - Context
  - Delayed Assignments
  - Schizophrenia
- 4 Summary

# Summary

## Separate Compilation of Synchronous Programs

- **modular compilation** vs. modular synthesis
- symmetric vs. **asymmetric linking**
- incremental vs. **separate compilation**

## Challenges

- module context
- delayed assignments
- schizophrenia

# Intermediate Format

## Averest Intermediate Format

- name (module name)
- interface (list of variables exposed at the data interface)
- locals (list of locally declared variables)
- context
  - context
  - surfaceCalls
  - surfDepthCalls
  - depthCalls
- surface
  - ctrlFlow (guarded actions of the control flow)
  - dataFlow (guarded actions of the data flow)
- depth
  - ctrlFlow (guarded actions of the control flow)
  - dataFlow (guarded actions of the data flow)
- absReacts (absence reactions)

# Context Interface

## Inputs

- $go^s(M)$ : execute surface
- $go^d(M)$ : execute surface and enter depth
- $abrt(M)$ : abort control-flow
- $susp(M)$ : suspend control-flow
- $prmt(M)$ : preempt data-flow

## Outputs

- $inst(M)$ :  $M$  is instantaneous now
- $insd(M)$ :  $M$  is active now
- $term(M)$ :  $M$  terminates voluntarily now