AIF and Modular Compilation

SYNCHRON 2008

Jens Brandt

Embedded Sytems Group
Department of Computer Science
University of Kaiserslautern

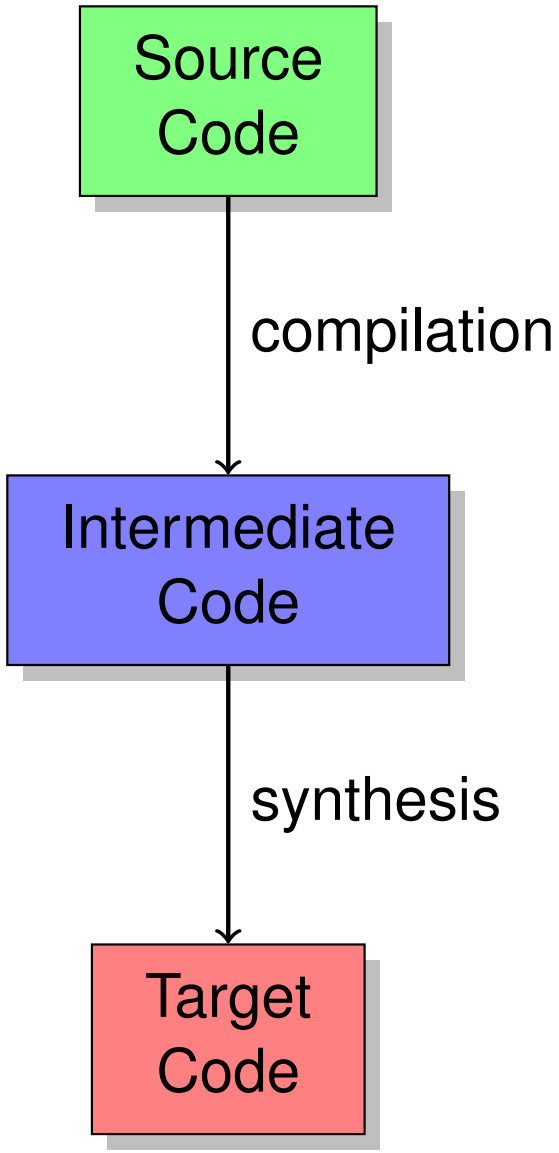
04 December 2008

Why a modular intermediate code?

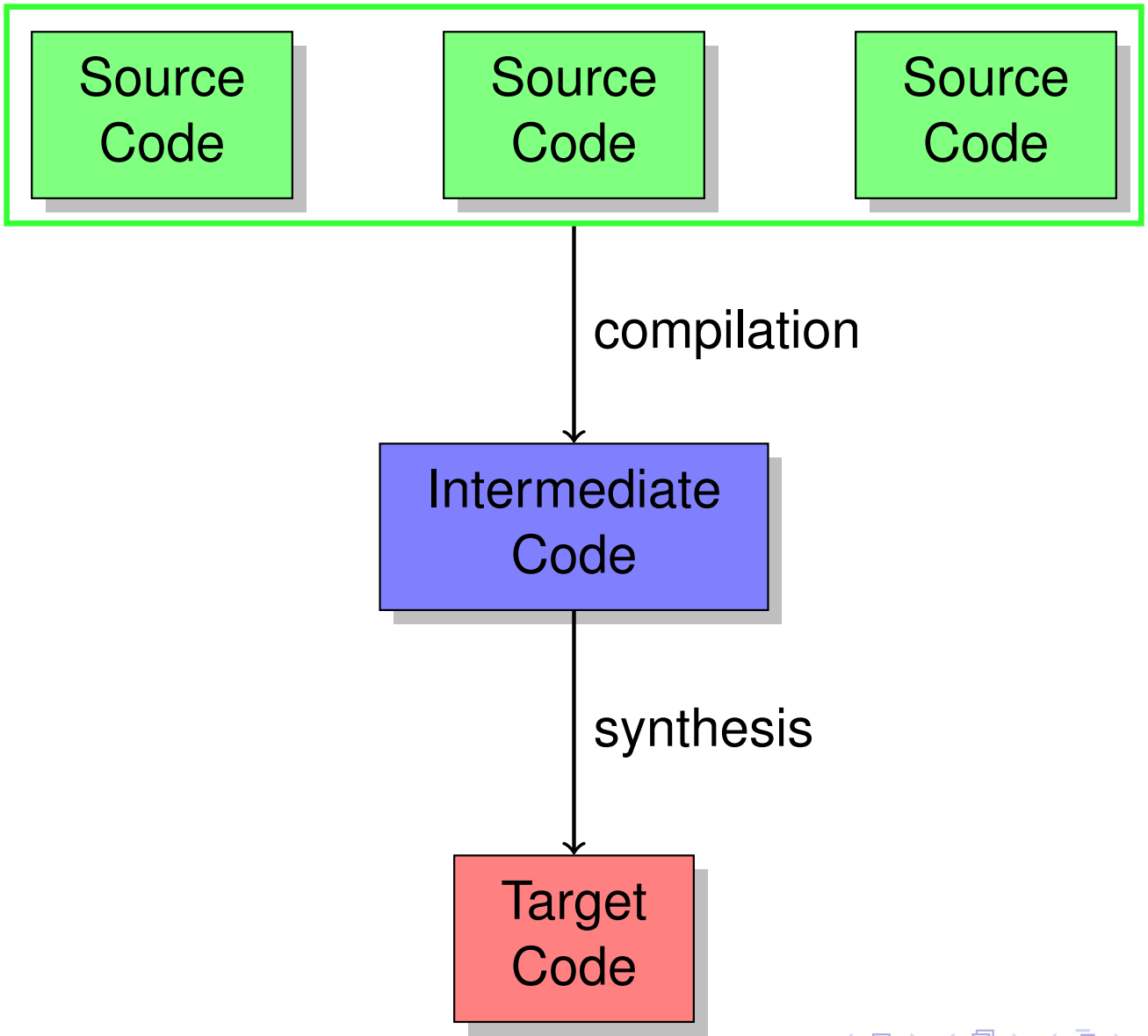
- split the overall compilation process into several phases
- first steps are independent of the realisation
- expensive optimisations may already be performed on the intermediate results
- complex statements reduced to simpler statements
- connect other languages and tools
- intermediate results can be stored
- intermediate code can be distributed in libraries without revealing its proprietary source

Outline

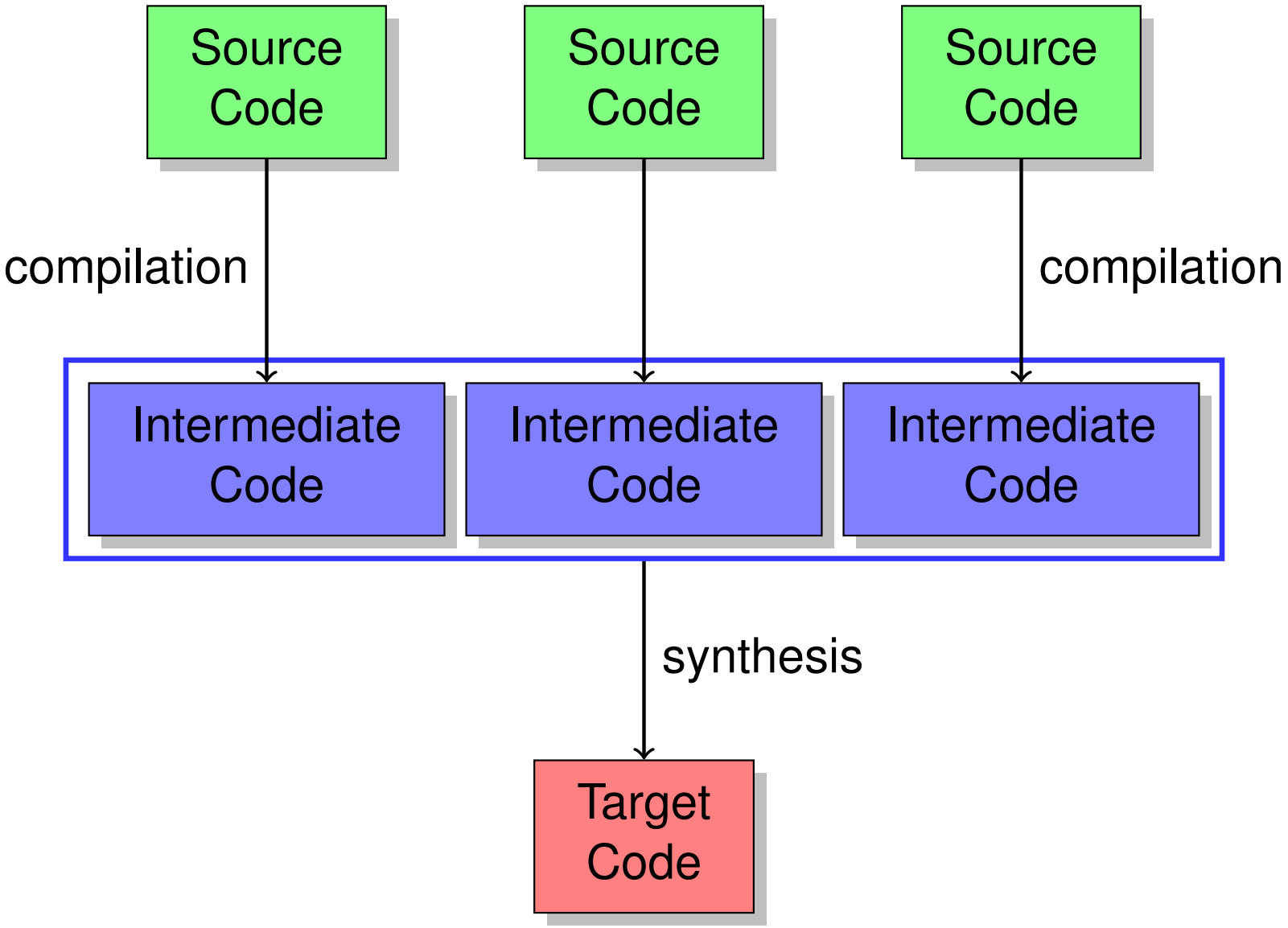
Compilation and Synthesis



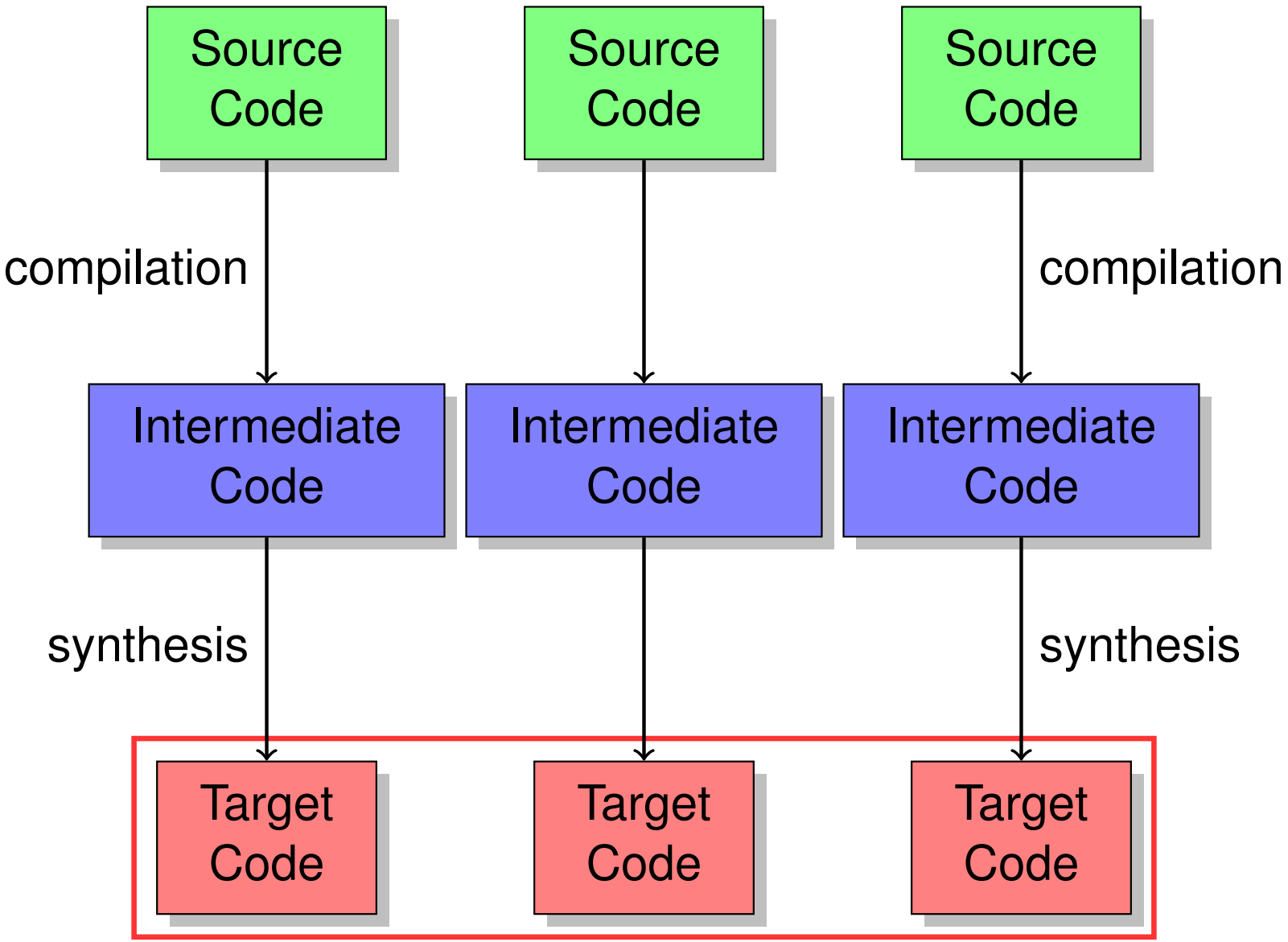
Non-Modular Compilation



Modular Compilation



Modular Synthesis



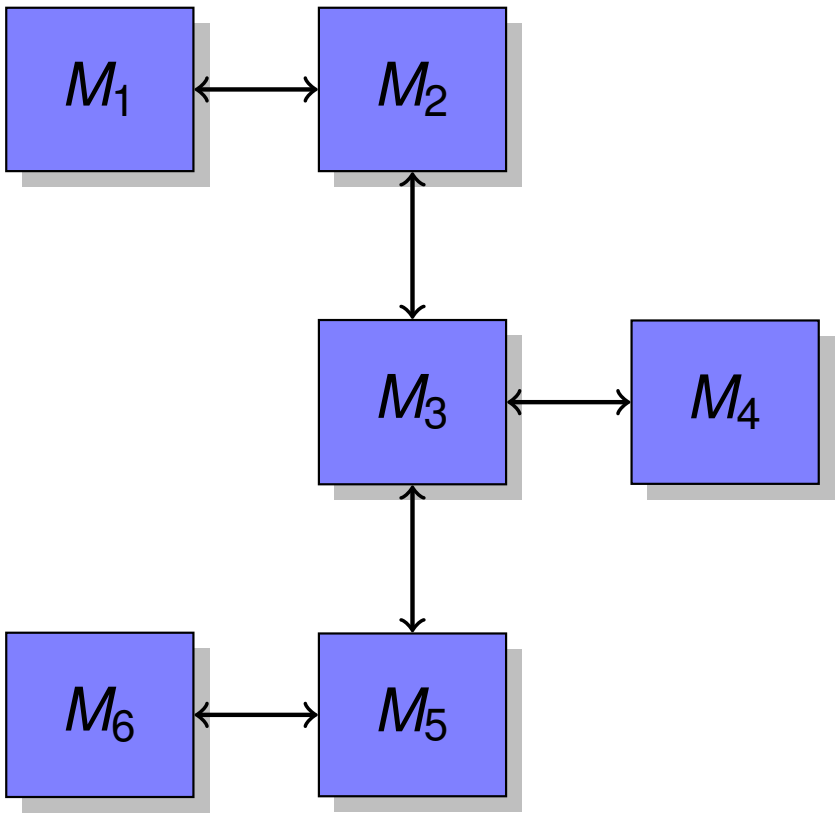
Symmetric Linking (1/2)

- modules: independent components communicating over their interfaces
- often explicit distinction between behavior and structure
- modules defined by behavioural part of the language
- subsequently assembled by the structural part

$\Rightarrow$ modules run in parallel, in the same context

- VHDL, SDL or even threads in all classical imperative programming languages

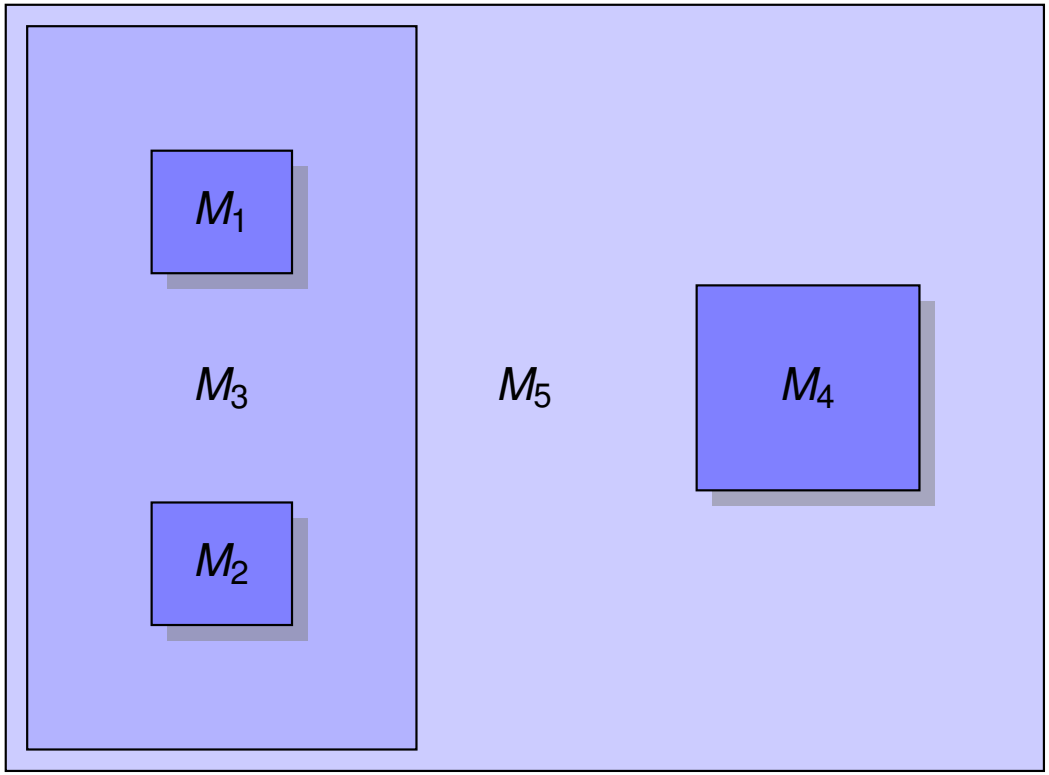
Symmetric Linking (2/2)



Asymmetric Linking (1/2)

- module instanciations: orthogonal to all other statements of the language
- module instantiations can be used at any place in the synchronous program
- often denoted as module calls
- the outer modules defines the context of the inner one
- behavioral hierarchy

Asymmtric Linking (2/2)

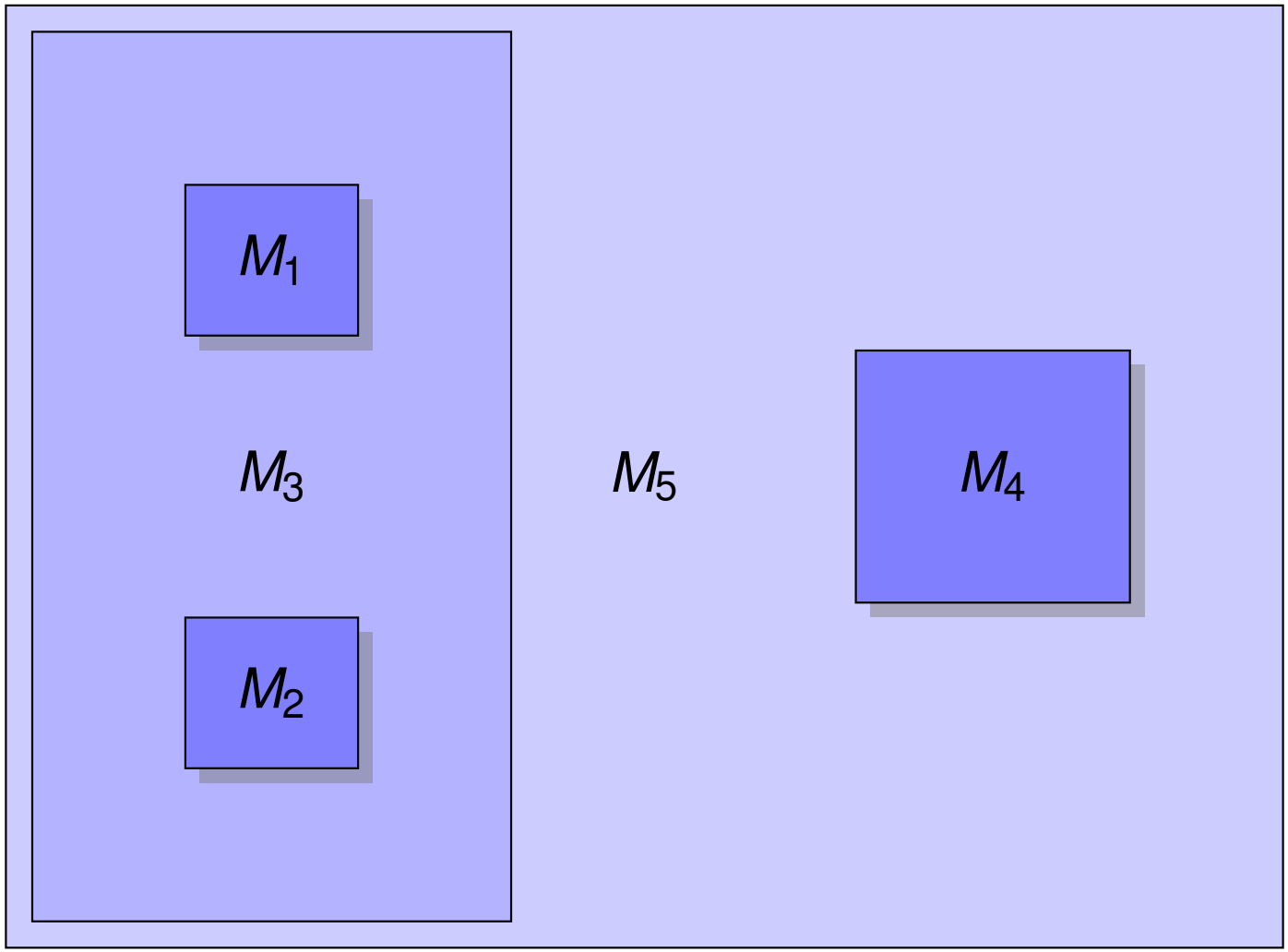


- **module hierarchy**
- Module M_5
  - Module M_4
  - Module M_3
    - Module M_2
    - Module M_1

Incremental vs. Separate Compilation (1/2)

- **incremental compilation**
  - compiled code for the inner module is available when the outer one is compiled
  - simply storing all the compilation results of a compiled module in a file
  - importing these results whenever a module instantiation of this module is compiled
- **separate compilation**
  - compile a module without having any knowledge about the called modules
  - compile all modules of the system in an arbitrary order and to link them afterwards
  - change internals of some module without recompiling all parts of the system that make use of it

Incremental vs. Separate Compilation (2/2)



Quartz

- imperative synchronous language,
  started as an Esterel variant
- **Averest System**
- compile, simulate, verify, synthesise Quartz programs
- target audience: academia
- available at `http://www.averest.org`
- major revision 2.0 almost available
  - new type system
  - significantly improved synthesis
  - modular compilation
  - new specifications: (observers, regular expressions, . . .)
  - . . .

The Synchronous Language Quartz

Core Statements

- **nothing**;
- $x = \tau$; and **next**$(x) = \tau$;
- ℓ : **pause**;
- **if** (σ) S_1 **else** S_2;
- S_1 S_2
- **do** S **while**(σ);
- **during** S_1 **do**S_2
- $S_1 \parallel S_2$
- **[weak] abort** S **when [immediate]**(σ);
- **[weak] suspend** S **when [immediate]**(σ);
- $\{\alpha\ x;\ S\}$
- $name(\tau_1, \ldots, \tau_n)$;

Guarded Actions

- AIF is based on **guarded actions**
- either the form $(\gamma, x = \tau)$ or $(\gamma, \textbf{next}(x) = \tau)$
- generated for both, the data flow and the control flow.
  - data flow: all assignments of the program
  - control flow: $(\gamma, \textbf{next}(\ell) = \text{true})$ where γ is condition to move to location ℓ
- semantics of guarded actions:
  - in each macro-step, check all guards are simultaneously
  - if a guard is true, its action is immediately executed

Intermediate Code Formats

- Imperative/Intermediate Code (IC)
- Object Code (OC)
- (Sorted) Sequential Circuit Code (SC and SSC)
- Declarative Code (DC)
- Event-Triggered Graphs
- Concurrent Control Flow Graphs (CCFG)
- Graph Code (GC)
- ...

Why Guarded Actions?

- guarded actions are exactly at the right level of abstraction
- good balance between
  - removal of complex constructs from the source code level, independence of a specifig source
  - independence of a specific synthesis target
- efficient translation to both software and hardware is possible (see OC and SC)
- guarded actions are **modular**.

$\Rightarrow$ linking: union of sets of the guarded actions

Example

Example (Inner Module)

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

$$i \Rightarrow (\textbf{next}(\ell) = \text{true})$$
$$\ell \Rightarrow (o = i)$$
$$\neg i \Rightarrow (o = \text{true})$$

Example

Example (Inner Module)

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

$$\ell_0 \wedge i \Rightarrow (\textbf{next}(\ell) = \text{true})$$
$$\ell \Rightarrow (o = i)$$
$$\ell_0 \wedge \neg i \Rightarrow (o = \text{true})$$

Example

Example (Inner Module)

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

- surface

  $i \Rightarrow$ **next**(ℓ) = true

  $i \neg i \Rightarrow o$ = true
- depth

  $\ell \Rightarrow o = i$

Absense Reactions

- guarded actions are complemented by reaction to absence

$\Rightarrow$ if no action has determined the value in the current step

- absence reactions of each variable may only depend on constants or previous values of the same variable
- in principle, absence reactions can also be expressed as guarded actions
- modular compilation needs to distinguish them from regular guarded actions
- example: $(b, 0, \mathrm{pre}(b))$

First Look at AIF

AIF

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

Example

Example (Outer Module)

module M_2(**bool** ?i, **bool** !o){
  loop {
    bool b;
    ℓ : **pause**;
    weak abort
      $M_1(i, b)$;
    when(b);
    $o = b$;
  }
}

- modules are called in a context
- 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

- $\text{go}^{\text{surf}}(M)$: execute surface
- $\text{go}^{\text{depth}}(M)$: execute surface and enter depth
- $\text{abrt}(M)$: 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

Data-Flow Preemption

- strong/weak data preemption in the surface
  handled by $\text{go}^{\text{surf}}(M)$ and $\text{go}^{\text{depth}}(M)$
- strong/weak data preemption in the depth
  handled by $\text{prmt}(M)$
- added to all guards of the module

Example

Example

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

- surface
  - context: $\text{inst}(M_1) = \neg i$
  - $\text{go}^{\text{depth}}(M_1) \wedge i$
    $\Rightarrow$ **next**(ℓ) = true
  - $\text{go}^{\text{surf}}(M_1) \wedge \neg i$
    $\Rightarrow o = \text{true}$
- depth
  - context: $\text{insd}(M_1) = \ell_1$
    $\text{term}(M_1) = \ell_1$
  - $\ell \wedge \text{susp}(M_1)$
    $\Rightarrow$ **next**(ℓ) = true
  - $\ell \wedge \neg\text{prmt}(M_1)$
    $\Rightarrow o = i$

Example

Example

module M_2(**bool** ?i, **bool** !o){
  loop {
    bool b;
    weak abort
      $M_1(i, b)$;
    when(b);
    ℓ : **pause**;
    $o = b$;
  }
}

- surface

  $\text{inst}(M_2) = \text{false}$

  $\text{go}^{\text{depth}}(M_2) \Rightarrow \textbf{next}(\ell) = \text{true}$

- depth

  $\text{insd}(M_2) = \text{insd}(M_1) \vee \ell$
  $\text{term}(M_2) = \text{false}$

  $\neg\text{abrt}(M_2) \wedge \ell \wedge \text{term}(M_1) \vee$
  $b@1 \Rightarrow \textbf{next}(\ell) = \text{true}$

  $\ell \wedge \neg\text{prmt}(M_2) \Rightarrow o = b$

  $(b, \text{go}^{\text{depth}}(M_2)\ \textbf{?}\ b@1\ \textbf{:}\ \text{false})$

Second Look at AIF

AIF

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

Problem 1: Data-Flow Preemption

- previously implemented by reprocessing the computed guarded actions
- this was done each time a strong preemption context was compiled

→ required that all guarded actions are known

- use $\text{go}^{\text{surf}}(M)$, $\text{go}^{\text{depth}}(M)$, $\text{prmt}(M)$ to handle preemption
- added to all guards of the module

Problem 2: Causality

- causality checking (at least for global variables) can only be done in the complete system and not in its individual components

→ not part of the compilation, but part of the linker

- if causality analysis is integrated to the modular compilation, an explicit causality interface must be provided
- such interfaces would restrict programs to acyclic dependencies

Problem 3: Schizophrenia

- guarded actions were computed in a recursive traversal of the structure of the source program
- removal of schizophrenia was deferred to the very end
  - count all incarnations of a module
  - create new incarnations depending on this index
- number of incarnations for any other module is not accessible in modular case

Problem 3: Schizophrenia (continued)

- handle local variable declarations completely locally
- module calls inside loops surfaces lead to reincarnations of the local variables
- required reincarnations are made by generating copies of the called modules in the different surfaces of a loop
- duplicate not single variables, but instead, the surface of a module
  (including other surfaces)
- multi-stage numbering of the incarnations needed:

$\Rightarrow$ qualified names

- x, x@1, x@2, x@3
- instA.instB.x, instA.instB@1.x, instA@1.instB.x, instA@1.instB@1.x, instA@1.instB@2.x

Problem 4: Delayed Assignments

- delayed assignments executed in the last step of a local variable scope must be deactivated
- very tricky problem related to modular compilation:
  - delayed assignment to output variable in last step
  - bound to a local variable in the outer module
  - module call in its last step
- deactivation cannot be made when compiling the module
- outer module does not know the actions of the inner one
- defer deactivation to the linker
- compiler endows all argument expressions of the module calls by appropriate activation conditions
- will be finally added to the guards of the linked module

Example

Example (Inner Module)

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

Example (Outer Module)

module M_2(**bool** ?i, **bool** !o){
  loop {
    bool b;
    $o = b$;
    ℓ : **pause**;
    weak abort
      $M_1(i, b)$;
    when(b);
    next(b) = true;
  }
}

Summary

- modular compilation vs. modular synthesis
- symmetric vs. asymmetric linking
- incremental vs. separate compilation
- Quartz
- Averest Intermediate Format
- modular compilation