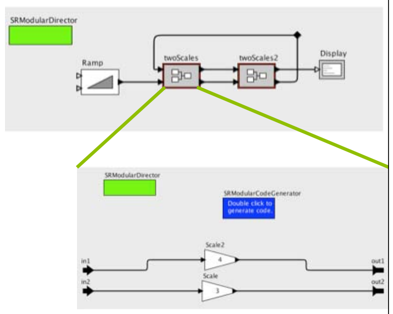
Synchronous Reactive Modular Code Generation

Dai Bui
Mentor: Stavros Tripakis
EE290N – Concurrent Models of Computations

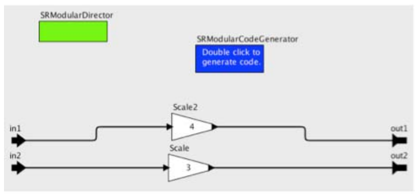
Introduction

- Synthesize an atomic actor from a composite actor
- Why?
  - IP protection: hide internals of composite
  - Efficiency: static scheduling
- Generated code should be modular
  - Independent from context



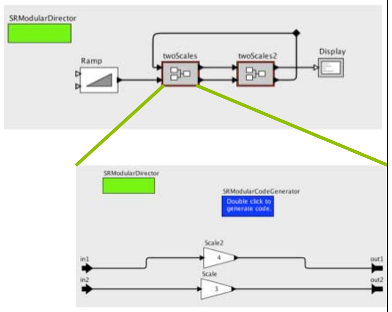
Usage scenarios

- Creating a new modular composite actor
  - Add SRModularDirector
  - Add SRModularCodeGenerator
  - Add other actors and connect them
- Generate code for each actor by pressing the SRModularCodeGenerator actor
  - Create an entry in User Library in Ptolemy actor tree like other atomic actors
  - This entry points to the generated Java class similarly to atomic actors



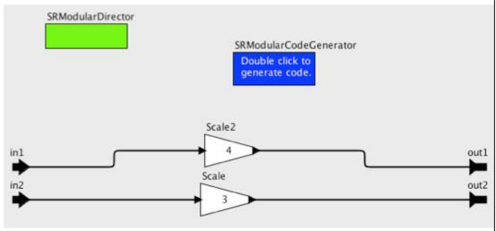
Introduction

- Synthesize an atomic actor from a composite actor
- Some issues with current approach
  - Flattening composite actors increases the number of actors in a model, and thus increases the scheduling computation of external directors
  - Flattening composite actors is not always possible when the composite actors are intellectual property (IP) composite actors -> composite actors should have their own fire functions
    - However, one monotholic fire function approach currently used in Ptolemy can reduce performance of SR models since composite actors might have to fire several times before the models reach fixed points



Key idea: Interfaces

- Instead of "monolithic" interface: single fire/ postfire functions
- Multi-function interface
- E.g.: fire1(), fire2()

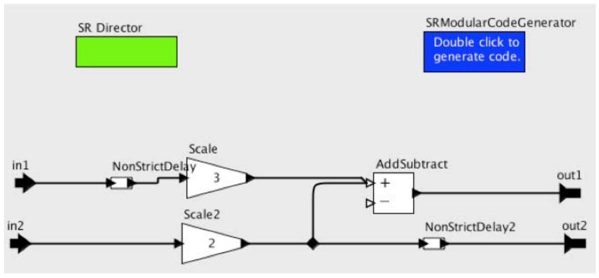


Solution

- We employ a new technique, called modular interface, in which each composite actor could have multiple fire interface functions so that an outside director can invoke appropriate interface functions based on the presence of respective inputs.
- The modular code generated for each composite actor should be independent from context the composite actor can be used. This can be achieved by generating a set of fire interface functions for each composite actors.
- The provided information about the interface functions is used by outside directors to make decisions on which fire interface function should be invoked based on the outside directors' scheduling algorithm.

Background

- Synchronous/Reactive
- Causality interfaces



New stuff

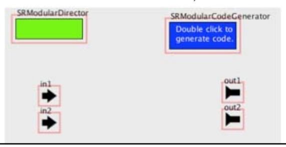
- New standard interface ***ModularInterface***:
  - Number of fire interface functions
  - Output ports belongs to each fire interface functions
- New ***SRModularDirector***:
  - If this director is inside a modular composite actor, it can fire multiple schedules according to which fire interface function of the modular composite actor is invoked
  - If the director is used as an external director, it can exploit the new features of modular composite actors

Code generation

- Each generated modular composite actor implements a standard interface called *ModularInterface* with standard functions that provides information about fire interface functions
  - Number of fire interface functions
  - Output ports belongs to each fire interface functions
- New SRModularDirector is implemented
  - If this director is inside a modular composite actor, it can fire multiple schedules according to which fire interface function of the modular composite actor is invoked
  - If the director is used as an external director, it can exploit the new features of modular composite actors

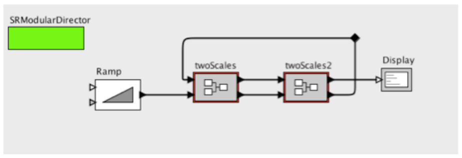
Store, Reconstruct and Hide internals

- The internal structure of a modular composite actors is store in the actor Java file in some form of XML structure
- Internal structures are constructed in constructors of Java class
- Compiling and preprocessing information of each modular composite actors, i.e. port dependency, number of interface functions, clusters of actors, …, are initialized in Preinitialization
- Internal structures are hidden so that users can not see, i.e. for IP protection



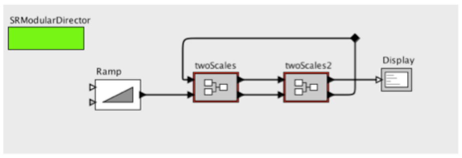
SRModularDirector

- External SRModularDirector uses a causality interface for modular director to derive schedules
- Exploit reflection mechanism in Java



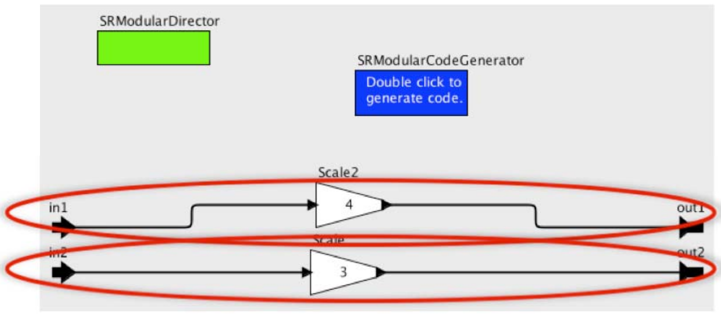
Operating mechanism

- External SRModularDirector uses a causality interface for modular director to derive schedules
- Exploit reflection mechanism in Java



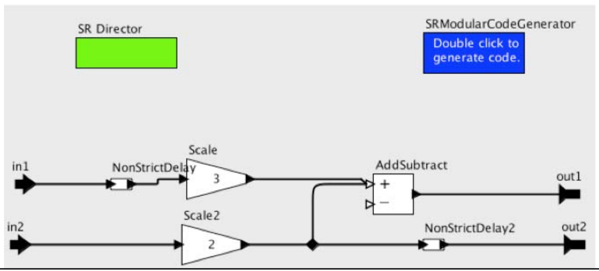
Clustering (1)

- How many interface functions to generate? Which ones?
- Clustering: different methods

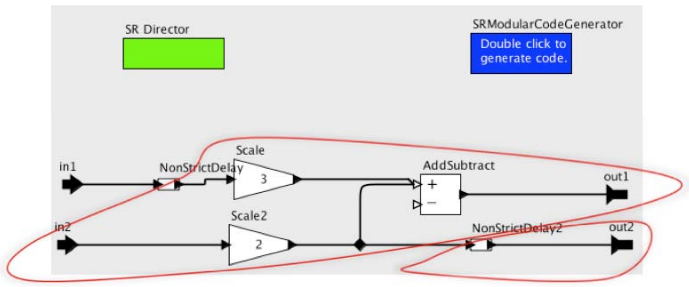


Clustering (1)

- A set of output ports depending on the same set of input ports is called a cluster
- Upstream actors of a cluster form a fire interface function

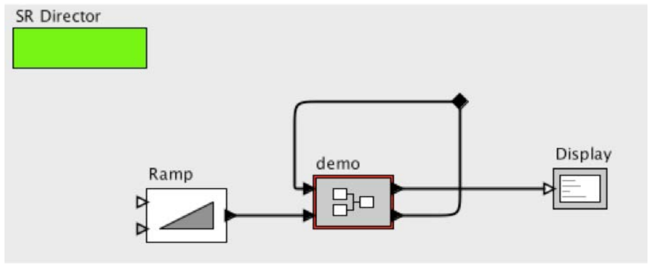


Clustering (2)



Compatibility

- Modular composite actors are compatible with conventional external SR Director



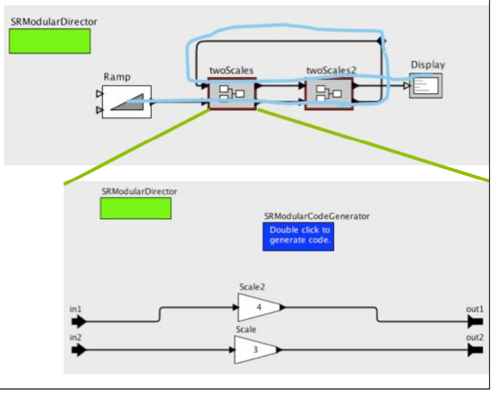
Example 1

Modular firing trace

```
Firing: Ramp interface function -1
Firing: twoScales interface function 1
Firing: twoScales2 interface function 1
Firing: twoScales interface function 0
Firing: twoScales2 interface function 0
Firing: Display interface function -1
```

Conventional firing trace

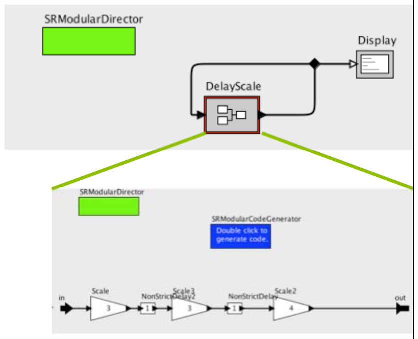
```
Firing: Ramp
Firing: Scales
Firing: Scales2
Firing: Scales
Firing: Scales2
Firing: Display
```

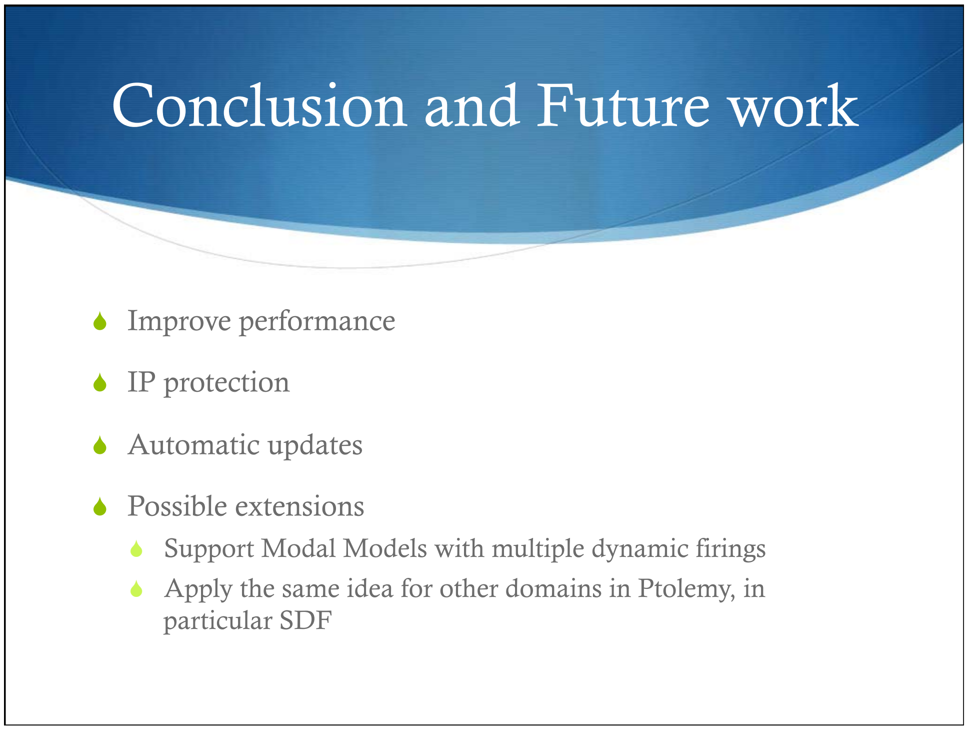


Example 2

Modular firing trace

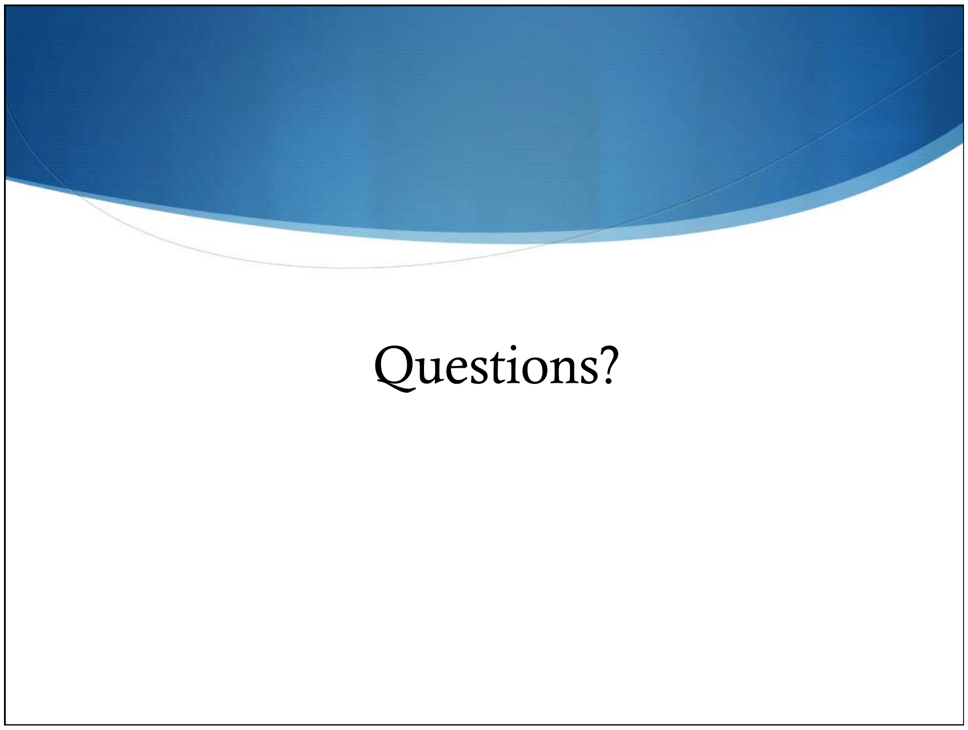
```
Firing: DelayScale interface function 0
Firing: Display interface function -1
```





Conclusion and Future work

- Improve performance
- IP protection
- Automatic updates
- Possible extensions
  - Support Modal Models with multiple dynamic firings
  - Apply the same idea for other domains in Ptolemy, in particular SDF



Questions?