Synchronous Reactive Modular Code Generation Project Report

Dai Bui
Mentor: Stavros Tripakis

May 15, 2009

Abstract

Opaque composite actors with monolithic fire functions can result in inefficient executions of Synchronous Reactive (SR) models because, although the causality interfaces of the composite actors may detect that there are no causality loops, the outside directors have no way of firing internal actors that their inputs are known, as a result SR models are executed several times until fixed points are reached.

To improve the performance, 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. We implement this with some clustering techniques. In addition, generating a composite actor so that other users do not need to know about the internal structure of the actor without sacrificing performance might be a good approach for intellectual property protection.

1 Introduction

In this project, we explore modular code generation from SR [1] domain in Ptolemy II meaning that we synthesize modular composite actors, similar to atomic actors, from composite actors. The modular code generation idea proposed in [3, 2] preserves the concept of hierarchy, which is often discarded due to flattening block diagrams, so-called composite actors in Ptolemy. The causality interfaces [4] analysis is used to derive dependency graphs for scheduling actors to minimize the number of firings in a composite actor.

Flattening composite actors increases the number of actors in a model, and thus increases the scheduling computation of external directors. Furthermore, flattening composite actors is not always possible when the composite actors are intellectual property (IP) composite actors. However, one monolithic 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 as shown in Section 4.

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 actor. 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.

2 Background

2.1 Synchronous Reactive

The Synchronous Reactive (SR) [1] domain in Ptolemy II models a system by composing multiple actors (blocks) into a model (diagram) and the actors are connected by zero-delay wires. Each zero-delay wire conveys a signal, which either has a known value or is absent. A SR director will fire actors in a model until all signals on the zero-delay wires in the model reach a fixed point.

2.2 Causality interfaces

Causality interfaces [4] in Ptolemy II construct a dependency graph between actors in a model. The dependency graph shows the order of firing actors in a model with a mechanism that if a downstream actor needs signals from upstream actors to fire, the downstream actor should not be fired before upstream actors that it depends on are fired.

3 Implementation

3.1 Usage scenarios

To use the modular code generation, users need to use a normal composite actor and open the composite actor. In the composite actor, users add a SR ModularDirector and a SRModularCodeGenerator, other actors, ports and then connect ports of actors as in Figure 2. After fishing a composite actor, one can

click on SRModularCodeGenerator, the code generation framework will create an entry for a newly created modular composite actor in User Library in the actor tree of Ptolemy. The entry will point to a Java class of the actor. The newly created modular composite actor is like an atomic actor.

3.2 Code generation mechanism

We have implemented a code generator actor that generates a modular composite actor with a Java class from a composite actor and stores it in the user library. The generated class has multiple interface functions with an internal director that can fire with different schedules according to which interface function is chosen.

The generated composite actor class also provides a standard interface, called *ModularInterface*, so that an outside director can detect its number of interface functions as well as ports belonging to each interface functions.

A new director, called SR ModularDirector, that can exploit new features of new modular generated composite actors was also implemented. This is implemented by a new causality interface that can construct a dependency graph for each interface function of an actor instead of one monolithic fire interface function of the actor. Whenever the new causality interface encounters an actor, it will ask if the actor implements the *ModularInterface* standard interface. If an actor implements the interface, the new causality interface will ask the actor about its interface functions. Based on the information the causality interface acquires, it will derive a static schedule for the director so that only interface functions that that have known values at their inputs are invoked. This avoids the unnecessary iterations as in monolithic fire interface function case. More details will be discussed in the Section 4.

Whenever a modular composite actor is fired, the outside director can use the $ModularInterface$ that provides a function to set an index parameter, called *interface function index*, so that the respective interface function will fire. In case the external director is not aware of the interface and does not set the index parameter, the conventional fire function will be invoked.

A composite actor is stored in user library pointing to a Java class similar to an atomic actor. For example, the *twoScales* actor in Figure 3 will have a *twoScales* class in a *twoScales.java*. When the class is instantiated, in the constructor of the Java class, subactors and connections between subactors are created, however, they are hidden from users as in Figure 1, and users cannot delete internal actors. With this mechanism, whenever a new Java class is generated when updating a composite actor, any model that uses the modular composite actor generated from the composite actor will be updated automatically since the structure of the modular composite actor is constructed by the Java class.

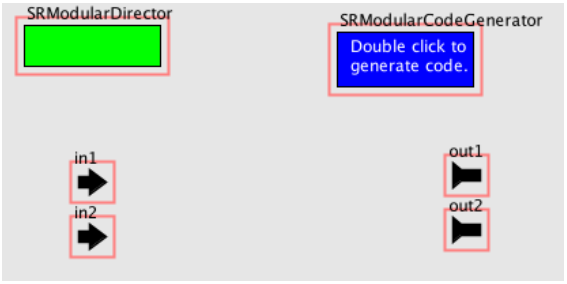

Figure 1: Actors are hidden from users

3.3 Clustering

We implement clustering algorithms proposed in [2, 3]. The set of outputs of a composite actor C is partitioned into the minimal number of k disjoint subsets $C_0, \ldots, C_{k-1}$ such that $\forall i = 0, \ldots, k-1$, all outputs in C_i depend on the same set of inputs of the composite actor. Each interface function now is associated one subset C_i. For example, as in Figure 5, two interface functions are created for two clusters of output ports, each interface function contains one Scale actor. The two interface functions can fire independently.

Each modular composite actor by default implements a postfire function that invokes the postfire function of all subactors.

For each cluster of output ports, the upstream actors of each cluster will form a firing sequence that creates the interface function for that cluster. The upstream actors of a clusters are actors that their output signals influences any output port of that cluster in one tick. Therefore, an upstream actor of a $NonStrictDelay$ actor of a cluster, does not belong to the interface function of that cluster if it does not connect to any other actor of that cluster. For example, as in Figure 4, actors $Scale, NonStrictDelay2, Scale3$ do not belong to the only interface function of the modular composite actor. We put all actors that do not belong to any cluster in the postfire function of the modular composite actor. The postfire function of the modular composite actor in Figure 4 would have a firing sequence like:

```
Fire: Scale
Fire: NonStrictDelay2
Fire: Scale3
Postfire of all fired actors
```

3.4 Compatibility

Newly generated modular composite actors can still fire normally when the external director is a conventional SR director.

4 Examples

4.1 Double parallel scales inside a modular composite actor

We create a model as in Figure 3, in which modular composite actors Scales have internal structure as in Figure 2. The internal structure of the Scales actor will be mapped into two interface functions, each has one Scale actor. The two interface functions can be invoked separately from outside directors.

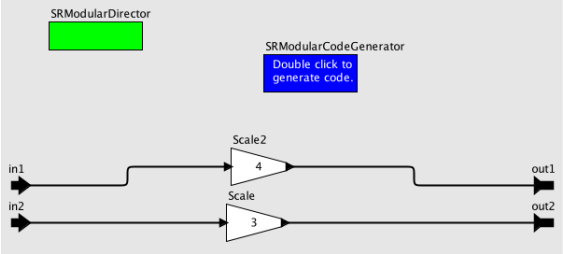

Figure 2: The twoScales composite actor

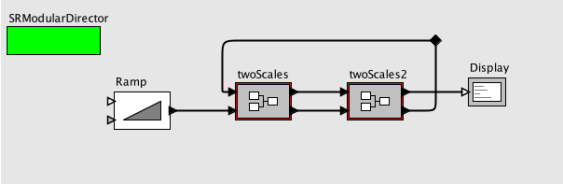

Figure 3: Using the "twoScales" modular composite actor (automatically generated from Figure 2

The firing sequence of the model at one tick is as follows:

```
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
```

The interface function -1 denotes the default conventional fire interface function. The above firing sequence shows that the fixed point is found after 1 iteration. No actor has to fire twice as in conventional SR case below, in which the two actor *Scales* and *Scales*2 have to fire twice until a fixed point is reached.

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

In this case the causality interface can detect that there is no causality loop even with only one fire interface function. However, it takes two iterations to reach a fixed point.

The preinitialize function of the code generated from the composite actor is as follows:

```
public void preinitialize()
  throws IllegalActionException {
  super.preinitialize();
  removeDependency(_nameToPort.get("in2"),
    _nameToPort.get("out1"));
  removeDependency(_nameToPort.get("in1"),
    _nameToPort.get("out2"));
  SRModularDirector director =
    (SRModularDirector)getDirector();
  director.
    addActorFireFunctionIndexToInterfaceFunction(
    _nameToActor.get("Scale"), -1, 0);
  director.
    addActorFireFunctionIndexToInterfaceFunction(
    _nameToActor.get("Scale2"), -1, 1);
  addInterfaceFunctionsOutputPort(0,
    _nameToPort.get("out2"));
  addInterfaceFunctionsOutputPort(1,
    _nameToPort.get("out1"));
  _numInterfaceFunctions = 2;
};
```

We only quote the preinitialize function because we do most of the initialization of the actor in this function. In which, the dependencies between input and output ports are specified. If one output port does not depend on some input port in one tick, the dependency between the two ports is removed. The removed dependency helps the causality interface detect false loops. After that, the interface functions of this actor are initialized by adding each subactor's interface function to the sequence of firings of that respective interface function. Then, the set of clustered ports of each interface function is initialized so that external director can ask the list of ports of each interface function when deriving a scheduling sequence. Then the number of interface functions of this modular composite actor is specified.

4.2 NonStrictDelay to break the loop inside a modular composite actor

Figure 5 shows the model of this example. The internal structure of the composite actor is as in

Figure 4 in which the *NonStrictDelay* actor will break the causality loop. The *NonStrictDelay* and *Scale*2 actors will be fired in interface function 0. *NonStrictDelay*2 and *Scale*1 and Scale actors will be fired in postfire of the modular composite actor.

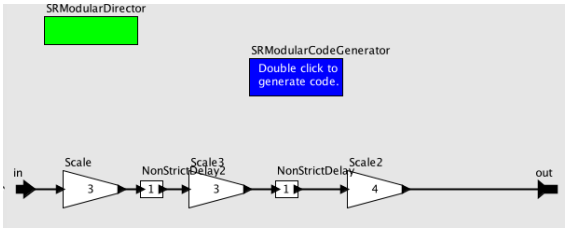

Figure 4: The DelayScale composite actor

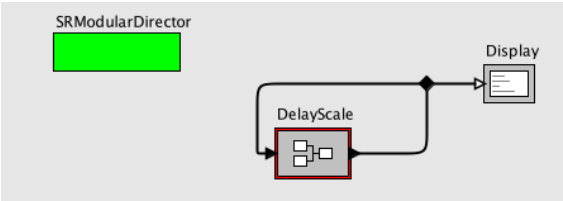

Figure 5: Using the "DelayScale" modular composite actor (automatically generated from Figure 4

When the model in Figure 5 runs, each actor also only needs to fire once every clock tick.

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

In conventional SR, the composite actor has to fire twice to reach a fixed point.

5 Conclusion and future work

The implementation of modular composite actors can improve the performance of SR models. The generation of modular composite actors also provides a good way of updating actors in a model whenever the actor library changes. Possible extensions of this work are: (1) to support modal models with multiple dynamic interfaces, and (2) to study modular interfaces for other Ptolemy domains, in particular SDF.

References

[1] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.

[2] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 78–89, New York, NY, USA, 2009. ACM.

[3] Roberto Lublinerman and Stavros Tripakis. Modularity vs. reusability: code generation from synchronous block diagrams. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 1504–1509, New York, NY, USA, 2008. ACM.

[4] Ye Zhou and Edward A. Lee. Causality interfaces for actor networks. *Trans. on Embedded Computing Sys.*, 7(3):1–35, 2008.