Interface Automata and Actif Actors

H. John Reekie

Dept. of Electrical Engineering and Computer Science
University of California at Berkeley
johnr@eecs.berkeley.edu

Abstract

This technical note uses the Ptolemy II interface automaton editor to compose networks of dataflow actors and buffers. The automata are first described as Actif actors; the composition of the automata together with a *controller* automata reveals the combined behavior of the network.

This technical note can be referenced as:

> "Interface Automata and Actif Actors," H John Reekie, UC Berkeley, November 2002, online at: `http://ptolemy.berkeley.eecs/~johnr`.

Or use the following BibTex citation:

```
@unpublished{Reekie0201,
  author = {H. John Reekie},
  title = {Interface Automata and Actif Actors},
  month = {November},
  year = {2002},
  note = {Online at \href{http://ptolemy.eecs.berkeley.edu/~johnr/}
    {http://ptolemy.eecs.berkeley.edu/~johnr/notes/atomicbuffer.pdf}}
}
```

1 Introduction

This technical note examines the use of interface automata [1] to compose dataflow actors, buffers, and a controller. This work was inspired by Lee and

Xiong's work on behavioral types [3], in which interface automata are used to construct a hierarchy of types based on the behavior of actors and the "domains" that execute the actors. Lee and Xiong have identified some issues with this approach, related to the atomicity of reading and writing tokens from and to channels [4]. I propose here that token reads and writes can be made atomic easily enough, provided that an actor does not need to check for whether tokens are present.

1.1 Dataflow actors

I use Actif [5] to describe actors. In Actif, an actor consists of a set of actions, and has sets of input channels and output channels. An action can *fire* if it has sufficient tokens available on the input channels. Figure 1 gives a textual representation of several actors.

- *source* produces a single output token each time it fires. The value is unimportant here.
- *sink* consumes a single input token each time it fires.
- *ndsource* will produce either one token on its output, or two tokens on its output, each time it fires. Which, is chosen non-deterministically.
- *ndmerge* merges two input streams. If a token is present on the first channel, it reads it and writes it to the output channel; if a token is present on the second channels, it reads it and writes it to the output. If tokens are present on both input channels, it non-deterministically chooses one of them.

Figure 2 gives the so-called "firing automata" of these actors. The firing automata indicate in what order the actions of the actor can fire. In the case of *source* and *sink*, there is only a single action, so the automata are trivial. In the case of *ndsource* and *ndmerge*, there are two actions. Each can fire at any time, provided that the requisite input tokens are present.

1.2 Interface automata

An interface automaton has a single action associated with each transition. These are input, output, or internal transitions, indicated by the suffix "?", "!", or ";" respectively. Figure 3 shows the interface automata of the actors above. Note that, since automata compose by the names of the actions on the transitions, the name of the automata is encoded into the actions. For example, the source actor is known as "A" and has actions *FireA* and *ReturnA*, representing a call to fire it, and the return from that call. Similarly, the actor writes to buffer "1", and hence has the output action *Put1*.

There is a straight-forward mapping from the actor's firing automaton to its interface automaton. Apart from the renaming mentioned above, each action in the Actif firing automaton is converted into a series of transitions consisting of a *Fire* action, a series of *Get* and *Put* actions, and a *Return* action.

```
actor source () (float out) {
  action a (() -> [1]);
}
actor sink (float in) () {
  action a ([1] -> ());
}
actor ndsource () (float out) start a,b {
  action a (() -> [1])   next a,b;
  action b (() -> [1,2]) next a,b;
}
actor ndmerge (float in1, float in1) {float out)
  start a,b {
  action a ([x][] -> [x])   next a,b;
  action b ([][y] -> [y]) next a,b;
}
```

Figure 1: Dataflow actors used in the examples

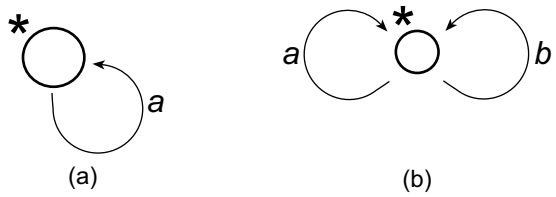


Figure 2: Firing automata of the actors used in the examples: a) *source* and *sink*, b) *ndsource* and *ndmerge*

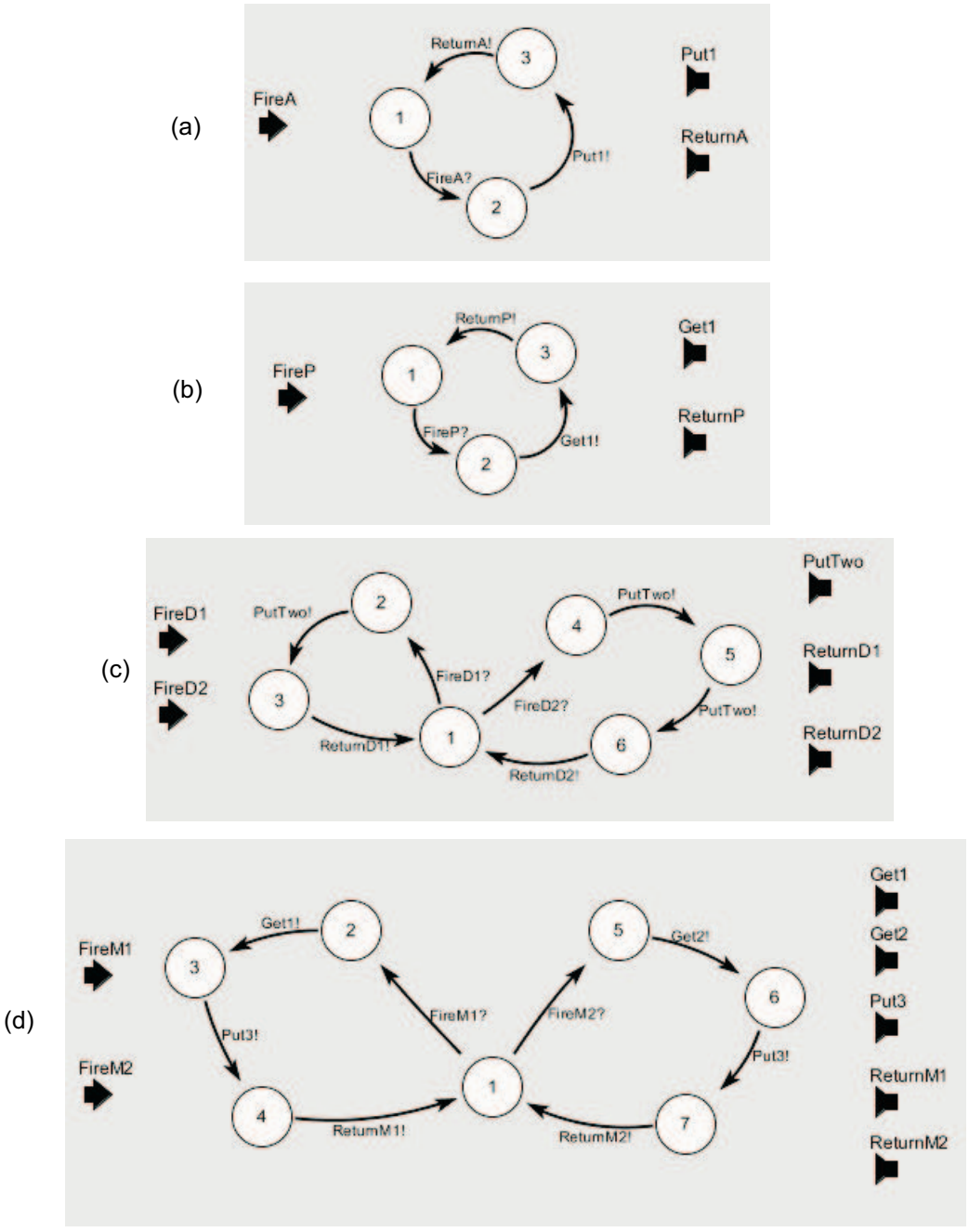
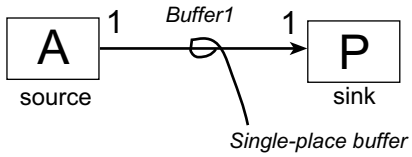
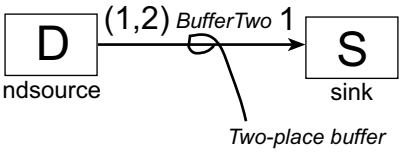


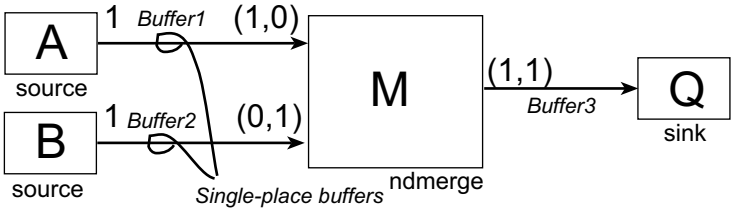
Figure 3: Interface automata of the actors used in the examples: a) *source*, b) *sink*, c) *ndsource*, and d) *ndmerge*



(a) Example 1*: single-action actors, single-place buffer*



(b) Example 2: *non-deterministic source with two actions, two-place buffer*



(c) Example 3: *non-deterministic merge, single-place buffers*

Figure 4: Networks uesd for composition examples

2 Composing interface automata

Interface automata can easily be composed using the interface automaton editor in Ptolemy II [2]. In this section I show the results of using this editor to compose the automata representing the components in the networks of Figure 4.

Actors communicate through channels. These can take varying forms, depending on the "model of computation" under which the actors are composed. Often, they take the form of FIFO queues; single-place buffers are also common. In these examples, we will use single-place buffers and two-place buffers. The interface automata of these buffers are shown in Figure 5. Each buffer receives events that get a token from or put a token into the buffer. In contrast to Lee and Xiong [3], this approach models a token put or get as a single atomic operation—there is no call and return, and there is no need to check for presence of a token. (This latter is because we are using an Actif controller to further constrain the actors, see below.)

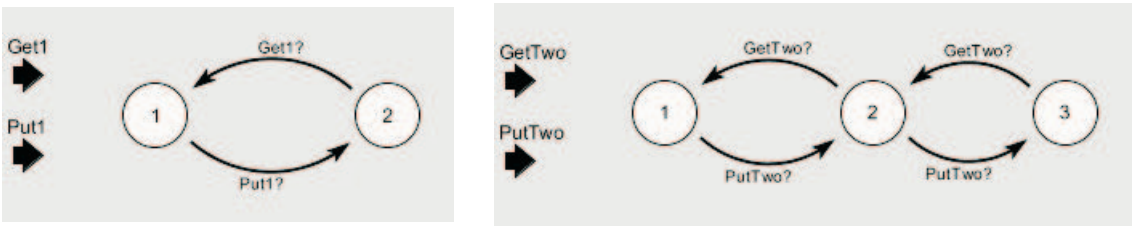


Figure 5: Interface automata for buffers: a) single-place; b) two-place

2.1 Example 1

Example 1 consists of a *source* actor, a *sink* actor, and a single-place buffer connected as in Figure 4a. Composing them yields the automaton in Figure 6a. Considering how simple this network is, this automaton turned out to be rather "messy." Still, this is what composition of the three interface automata yields.

One way that we can simplify this automaton is to now compose it with a *controller*. The controller is Actif parlance for the "environment" (in interface automata theory); it also performs the same function as a "director" in Ptolemy II, but specialized for a particular network. A good schedule for Figure 4a is fairly obviously to fire the source actor A, then fire the sink actor P, then repeat. The interface automaton of this controller is shown in Figure 7a; composing it Figure 6a yields the (closed) system shown in Figure 6c.

Note that in Actif, an actor never has to check that tokens are present on its inputs. This task is done, in effect, by the controller—that is, the controller is constructed such that an action is fired only if the required tokens are already present on its input.

It turns out that Figure 6a has a lot of states that will *always* be removed when a network is composed with a single-threaded controller. If you examine this figure, you will see that the first two transitions are *FireA?* and *Put1*. The next state has two output transitions, *ReturnA!* and *FireP?*. In a single-threaded system, *FireP?* cannot occur at this point, as actor *A* must return before another actor can be fired. This is the same limitation Lee and Xiong noted, but at the actor firing level rather than the token read/write level.

A simple solution is to simply remove the unwanted transitions; if all these unwanted transitions are removed and unreachable states pruned from the graph, the result is as shown in Figure 6b. Composing this automaton yields of course the exact same result as before. Note that:

- This extra pruning step reduces the size of the state space when composing actors and buffers.
- The input and output actions remaining in Figure 6b are the exact inverse of the controller. In *some* cases, then, we can derive the controller for a

network of Actif actors simply by composing the automata of the actors and buffers.

2.2 Example 2

Example 2 (Figure 4b) is similar to example 1, but uses a source actor that non-deterministically outputs either one or two tokens. A suitable controller for this network is shown in Figure 7b. As for example 1, the same comments apply in respect to pruning the composed automata of impossible call-return sequences. The result of composing the source, the sink, and the buffer with the controller is shown in Figure 8a.

2.3 Example 3

Example 3 uses a non-deterministic merge actor to merge the output from two source actors (Figure 4 c). This network is sufficiently complex that the automata of the actors and buffers cannot be composed *without* using the controller to constrain the size of the result automata—computing the composition simply takes too long. By composing the actors with the controller first, the complete composition is simple enough to compute, and is shown in Figure 8b.

3 Concluding remarks

It seems that for Actif actors that do not have a *choice* vertex, interface automata are easily derived and composed. Although I have not shown it in this report, Actif controllers have a similar state-machine structure to actors, and so could easily have interface automata derived for them.

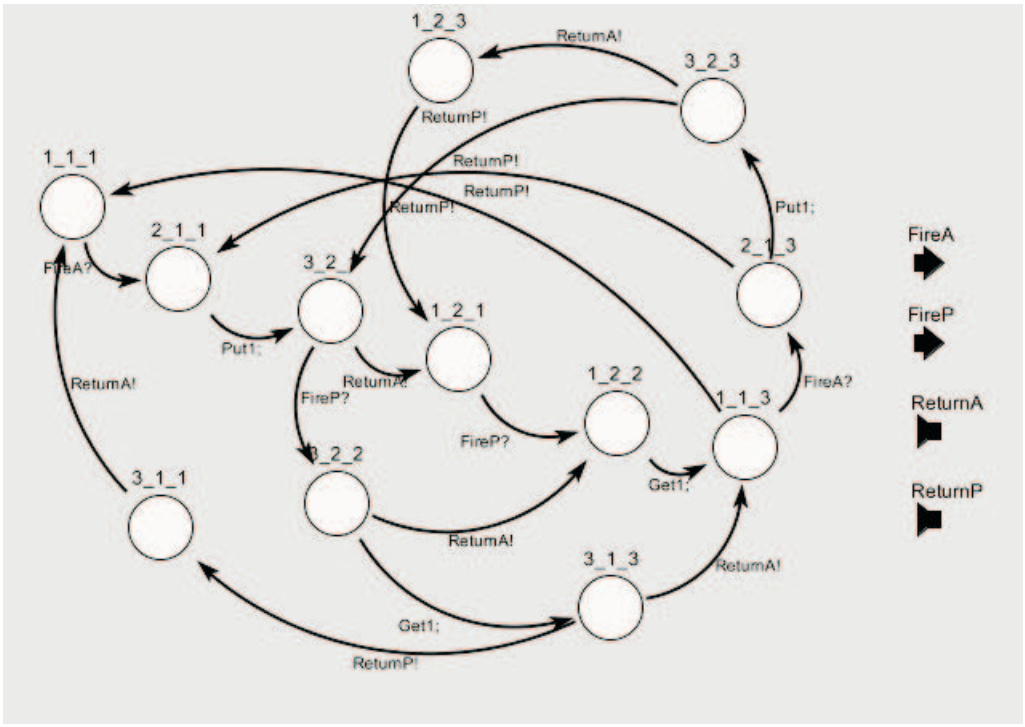
If an actor has choice, then an interface automaton cannot be easily derived for it. I need to look into this more. My intuition is that choice vertices can be inserted into interface automata and propagated through composition, so that the result automaton simply preserves the product of the states before and after the choice.

Independently of Actif, this report illustrates one way in which additional infeasible states can be pruned while composing automata, by disallowing action sequences that conflict with some known property of the desired result automaton. (In these examples, the property is that calls to *Fire* cannot be nested.)

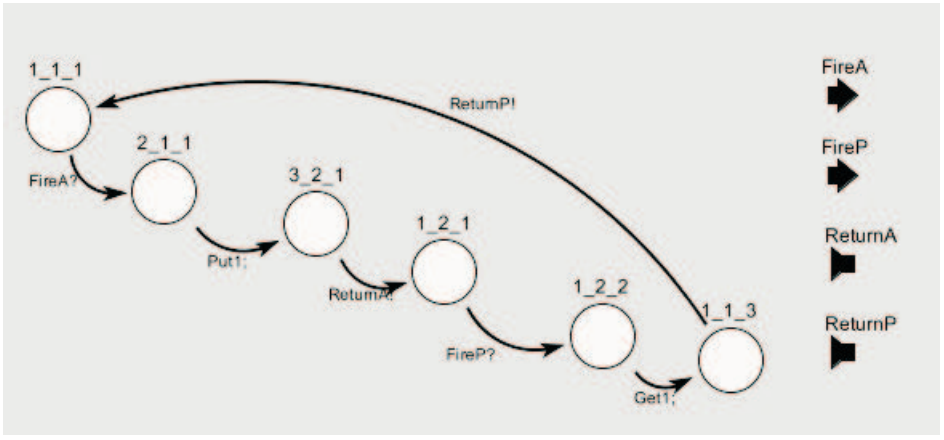
References

[1] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001. Online at http://www-cad.eecs.berkeley.edu/ tah/.

(a)



(b)



(c)

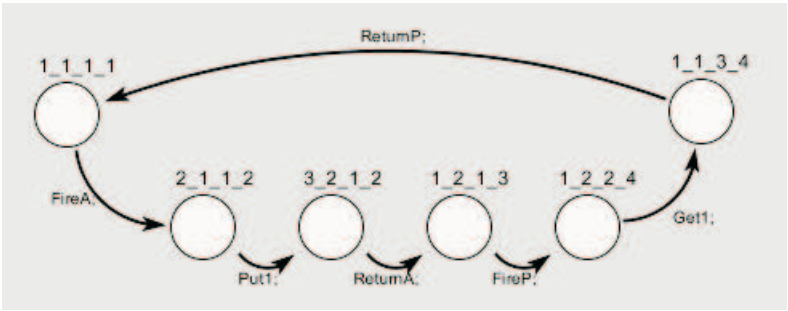
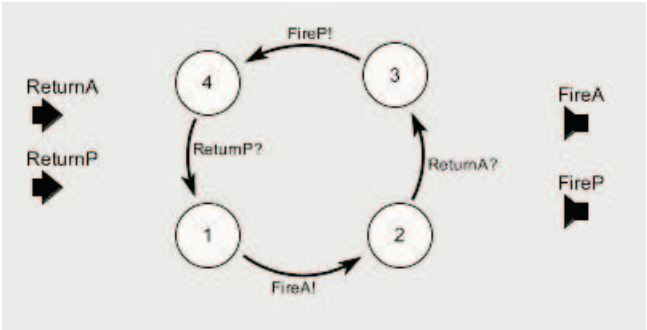
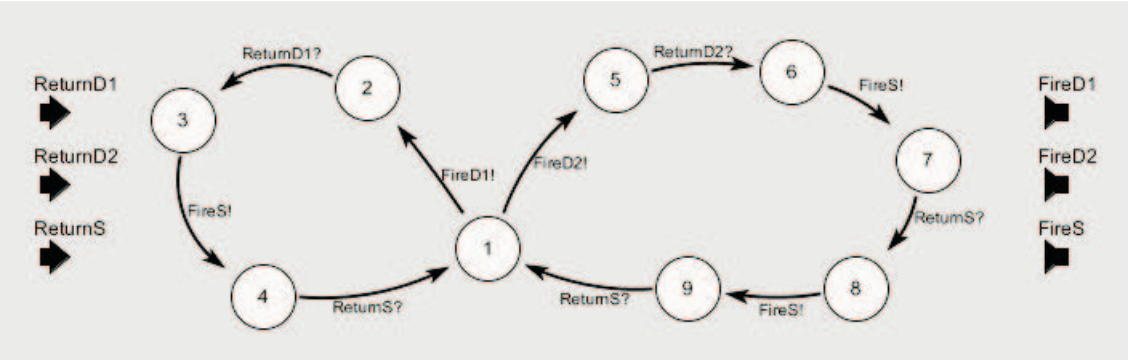


Figure 6: Composed automata for Example 1. a) composition of two actors and a buffer, without fire-return pruning; b) with fire-return pruning; c) after composing either (a) or (b) with a controller

(a)



(b)



(c)

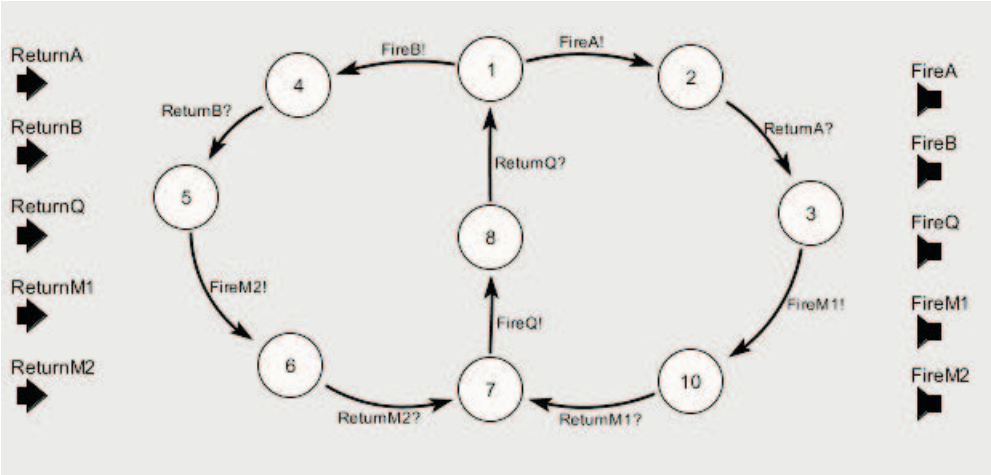
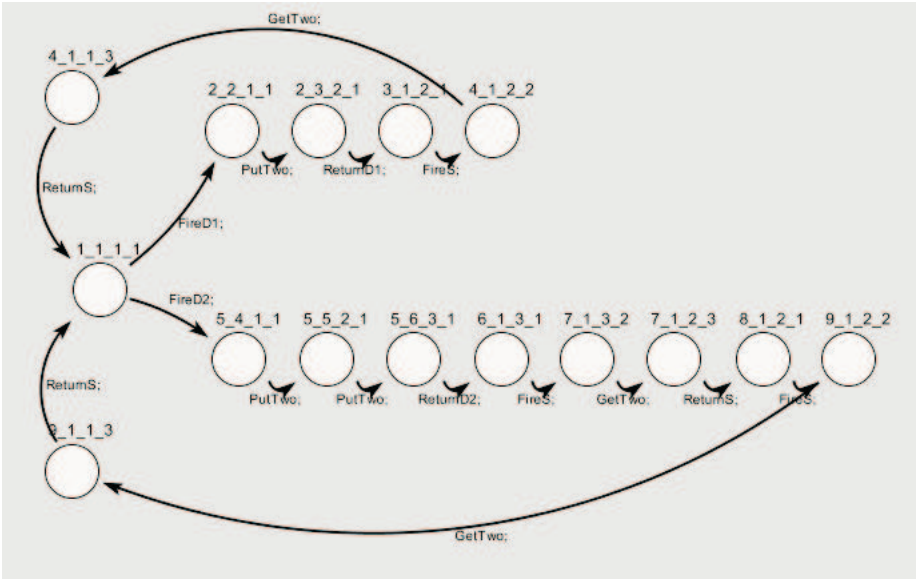


Figure 7: Controllers for the three example networks

(a)



(b)

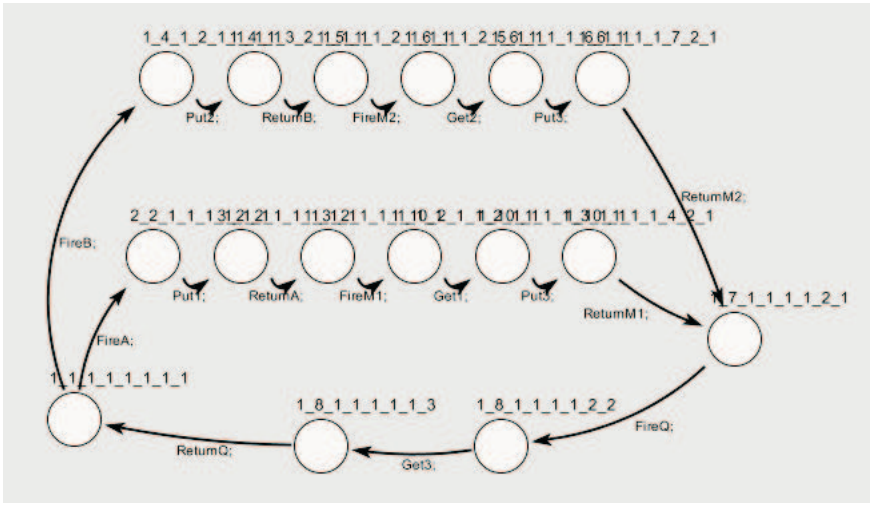


Figure 8: More composed networks: a) Example 2; b) Example 3

[2] John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL M01/12, Electronics Research Laboratory, Dept of EECS, University of California at Berkeley, March 2001. Online at http://ptolemy.eecs.berkeley.edu/.

[3] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. In T.A. Henzinger and C.M. Kirsch, editors, *EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[4] Edward A. Lee and Yuhong Xiong. Behavioral types for component-based design. Technical Memorandum UCB/ERL M02/29, Electronics Research Laboratory, Dept of EECS, University of California at Berkeley, September 2002. Online at http://ptolemy.eecs.berkeley.edu/.

[5] H. John Reekie and Edward A. Lee. Lightweight components models for embedded systems. Technical Report UCB ERL M02/30, Electronics Research Laboratory, University of Californa at Berkeley, November 2002. Online at http://www.gigascale.org/pubs/344.html.