A NEW METHOD OF SPECIFYING DATAFLOW ACTORS

by

Chris Chang, Johan Eker, Jörn W. Janneck, Yang Zhao

Memorandum No. UCB/ERL M02/28

27 September 2002

A NEW METHOD OF SPECIFYING DATAFLOW ACTORS

by

Chris Chang, Johan Eker, Jörn W. Janneck, Yang Zhao

Memorandum No. UCB/ERL M02/28

27 September 2002

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A New Method of Specifying Dataflow Actors

UCB/ERL M02/28
Chris Chang, Johan Eker, Jörn W. Janneck, Yang Zhao

EECS Department
University of California at Berkeley
Berkeley, CA 94720
U.S.A.
{cbc,johane,janneck,ellen_zh}@eecs.berkeley.edu

Abstract. In the design of embedded software, approaches such as synchronous dataflow or cyclo-static dataflow describe mechanisms for statically scheduling specialized classes of dataflow actors. This paper argues that these classes are often more specialized than is necessary in order to compute a static schedule, and introduces a more general notion of actor that allows for more flexible specifications of actor networks, while retaining the advantages of static analyzability and schedulability of compositions of actors. It also sketches the scheduling techniques by reducing the problem to well-known problems in the field of Petri nets.

1 Introduction

Modern embedded systems typically consist of several concurrently operating components that interact with each other and with the external environment. This is why *actors*—concurrent, interacting components—are a natural choice for modeling and implementing such systems. In contrast to objects, which interact via synchronous method calls, actors communicate by exchanging data via *ports* and *connections* between their ports. Hence, actors typically have no way of transferring control between each other.

The actor notion was introduced by [6] for modeling distributed knowledge-based algorithms. Since then, actors have become more widely used, cf. [2]. Several more or less specialized actor models have been used in the construction of software, and specifically embedded software, e.g. [13, 12, 9], and a variety of tools and design frameworks support an actor-oriented view of a system, e.g. Ptolemy [1], Moses [8], Grape II [10], Matlab/Simulink, LabVIEW, and SDL.

Not all environments that support software construction use a language geared towards the specification of actors—instead, many are based on the stylized use of 'traditional' software concepts, such as procedures and objects. This is especially the case for environments supporting *dataflow actors*, which are actors that receive and send data parcels called *tokens* from and to each other.

Dataflow actors are useful for describing embedded software due to the importance of issues such as data rates and optimizations in areas such as scheduling and buffer allocation. It is often desirable to compute a schedule of execution for the actors at design time, and approaches such as synchronous dataflow (SDF) and cyclo-static dataflow (CSDF) provide efficient scheduling mechanisms for specialized classes of actors.

This paper argues that these classes are often more specialized than is necessary in order to compute a static schedule, and introduces a more general notion of actor that allows for more flexible specifications of actor networks, while retaining the advantages of static analyzability and schedulability of compositions of actors.

The remainder of this paper is structured as follows. After summarizing some prerequisites in section 2, we discuss why traditional approaches lead to overspecification of actors and a consequent lack of reuse in section 3. This motivates a more flexible notion of actors, which we describe using the `Cal` actor language in section 4. In section 5 we show how the extra degree of freedom obtained by this notion addresses the reusability problem, by using a scheduling mechanism which we call *action-level scheduling* (ALS). However, this degree of freedom also potentially leads to ambiguous behavior, and section 6 discusses this issue. We address the ambiguity problem in sections 7 and 8, by first generalizing our notion of actors, and then extending action-level scheduling to handle this more general class of actors. We summarize and discuss our results in section 9.

2 Static scheduling of actor networks

This paper focuses on the implications of the proposed notion of actors for the *static* scheduling of actor networks, i.e. the design-time computation of a sequence in which the actors in a network of actors are executed. Doing this ahead of time, rather than during the execution of the system, is particularly interesting in the case of embedded systems, because it allows not only to remove the scheduling from the executing code, but also facilitates generation of memory- and time-efficient code from the actor network.

When statically scheduling an actor network, one is usually looking for *cyclic* schedules. In a network of dataflow actors (which are connected by FIFO buffers), a cyclic schedule is a sequence of actor activations (firings) that results in the same number of tokens being written to each buffer as were read from it—thereby leaving the buffer lengths unchanged.

In order to be able to compute a cyclic schedule ahead of time one needs to constrain the possible behaviors that the actors in the network may exhibit. For example, actors that produce or consume different numbers of tokens depending on the *values* of their input tokens are usually disallowed, because it would be necessary to know about token values in order to compute a schedule, which is impossible or infeasible in most cases.

In *synchronous dataflow* (SDF) [12] networks, actors are constrained to constant token rates—at each firing, an actor must consume and produce the same number of tokens at each input and output port (although these numbers may be different for different ports). With this constraint, actor networks can be efficiently checked for schedulability, and techniques have been developed to generate code that optimizes characteristics such as buffer size or code size.

The definition of a *cyclo-static dataflow* (CSDF) [5] actor relaxes the constraint on the token rates a little by requiring the numbers of tokens consumed or produced on each port to cycle through a series of values during successive firings of the actor. Schedulability can still be determined statically, and a schedule can be computed ahead of time.

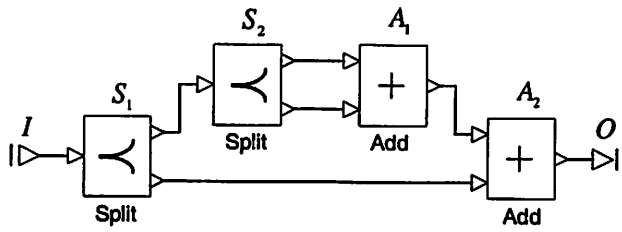

Fig. 1. A small example system.

Further generalizing SDF and CSDF actors, which we will call *constant* and *cyclic* in the following, this paper introduces two new classes of actors (which we call *free* and *regular*) and show how networks of these, too, can be statically scheduled.

3 Problem statement

Consider the network of actors shown in figure 1. Here we are using two different kinds of actors:

1. `Add` actors (A_1 and A_2), which consume one token from each input port, outputting a token which is their sum.
2. `Split` actors (S_1 and S_2), with two possible actions:
   (a) *Up*: The actor consumes a token from its input port and sends it to its upper output port.
   (b) *Down*: The actor consumes a token from its input port and sends it to its lower output port.

The above does not uniquely specify the behavior of the `Split` actors, because it does not define a policy for choosing between their two actions. For example, the sequence

$$S_1^{Up}—S_1^{Up}—S_1^{Down}—S_2^{Up}—S_2^{Down}—A_1—A_2$$

is a cyclic schedule of the actions of the actors, as are

$$S_1^{Down}—S_1^{Up}—S_2^{Down}—S_1^{Up}—S_2^{Up}—A_1—A_2$$

and

$$S_1^{Down}—S_1^{Up}—S_1^{Up}—S_2^{Down}—S_2^{Up}—A_1—A_2$$

In the first schedule, the actions of S_1 are executed in the order *Up—Up—Down*, whereas in the other two schedules they are executed in the order *Down—Up—Up*.

In this case, a cyclo-static dataflow description of a `Split` actor would include the order in which its actions are executed, its *internal schedule*. Conceptually, the internal

schedule is part of the actor specification and is fixed even before the actor is used in a network. If it is different for two actors, then the result will be two different actors.

Since the internal schedules are fixed ahead of time, the schedulability of the overall network not only relies on having the "correct" `Split` actors available but also on the user to manually choose them. For example, choosing the schedule *Up—Down—Down* for S_1 would result in a network that has no cyclic schedule.

This example illustrates two important points: first, the manner in which a `Split` actor is specified, namely, as a *fixed* sequence of actions, reduces the possibilities for reusing the actor in different contexts. Second, constructing the network becomes error-prone, since the user has to choose actors with the correct internal schedules.

We can generalize to any cyclo-static actor: its specification requires fixing its internal schedule, and although sometimes this is precisely what the user wants, other times it may limit the usefulness of an actor by essentially overspecifying actor behavior. Actors which affect the flow of tokens, such as the `Split` actors, are typical examples where a fixed sequence is often undesirable. We therefore need more flexible actor descriptions, which can be used to automatically compute different internal schedules depending on the actor's context.

By analogy with polymorphism in type systems, the two instances of the `Split` actor above perform the same function but have different *types*. In each case, the user is forced to specify the type (ordering of actions) explicitly. Would not a means for *inferring* these types be desirable? We will show that this is what our system achieves, while simultaneously allowing the user to explicitly "declare types" as needed.

To accomplish these goals, we propose a different method for specifying actors: first, we relax the need to specify a precise *sequence* for the actions and second, we provide concepts and notation for describing families of internal schedules, some of which will also require the actors to have state.

Hence, our actor specification will consist of two parts: a *set* of actions and some state. This is precisely how an actor is specified in the `Cal` language, which will be the topic of the next section.

4 Introduction to the `Cal` actor language

`Cal` [4] is an actor language developed in the Ptolemy project [3]. We will focus on those aspects of the language relevant to the problem discussed here, viz. a more flexible representation of dataflow actors.

We begin with two examples—the `Cal` versions of the `Add` and `Split` actors from the previous section:

```
1 actor Add [T < Number] ()
2   T Input1, T Input2 ==> T Output :
3
4   action [a], [b] ==> [a + b] end
5 end
```

```
1 actor Split [T] ()
2   T Input ⟹ T Output1, T Output2 :
3
4   Up :    action [a] ⟹ [a], [] end
5   Down :  action [a] ⟹ [], [a] end
6 end
```

The essence of each actor is contained in its set of *actions*. An action is what an actor executes when it *fires*, and it is atomic, in the sense that no two actions of the same actor may execute at the same time (which could create race conditions on the actor state). Each action consumes a number of input tokens, produces a number of output tokens, and possibly changes the state of the actor.

Each action has a pattern on the left which is matched to the input. For example, the action in line 4 of the `Add` actor consumes a token from its first input port, calls it `a`, then consumes a token from its second input port, calls it `b`, and finally sends the value of `a+b` to its output port.

An action may only be executed if its input patterns match the actual input sequences; for example, the two actions of the `Split` actor each require at least one token to be present at the input port. If such a token is present, both actions can be executed—the order in which they occur in the actor does not imply any precedence among them.

We call actors *free* if the executability of their actions only depends on the presence of a constant (for each port and action) number of tokens. Non-free actors add further preconditions to their actions. Combined with state this will allow us to constrain the internal schedules of these actors, which we will discuss in section 7.

In the following section we will first consider the problem of computing a schedule for a network of free actors, before generalizing the technique for a more comprehensive class of actors.

5 Action-level scheduling

Given a network of free actors, we will employ techniques developed in the context of Petri nets [14] to compute the set of valid cyclic schedules in three stages:

1. Map the network of actors into a Petri net.
2. Compute the *firing vectors* of the Petri net.
3. Search for *legal firing sequences* of the Petri net.

The result of these steps is a set of possible sequences in which the actions of the actor network may be fired.

5.1 Mapping an actor network to a Petri net

The Petri net representing the actor network in figure 1 is shown in figure 2. It is constructed as follows:

- Actions are represented by *transitions*, e.g. the rectangles labeled *Up* and *Down* in the dashed box labeled S_1 (called S_1^{Up} and S_1^{Down} in the following).

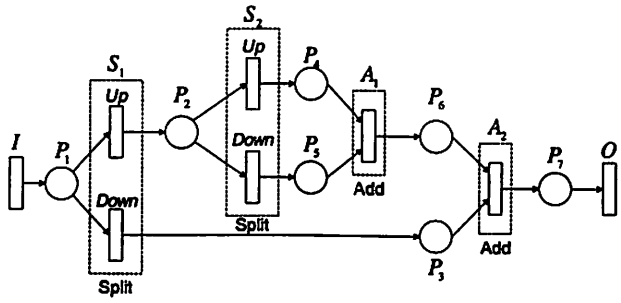

Fig. 2. The Petri net representing the actor network in figure 1.

- The input ports of actors are represented by *places*, e.g. circles labeled P_4 and P_5 representing the input ports of actor A_1.
- Connections from output ports to input ports are represented by arcs from transitions to places, e.g. the arc going from S_1^{Down} to P_3. The arc weight is equal to the number of tokens produced by the action represented by the transition on the corresponding output port.
- If an action consumes tokens from an input port, an arc is drawn between the corresponding place and the transition representing the action, e.g. the arc between P_3 and A_2. The arc weight is equal to the number of tokens consumed by the action from the input port.
- Input ports of the network are represented by transitions, e.g. the transition labeled I. They have outgoing arcs with weight 1 to every place representing an actor input port that the network input port is connected to, e.g. the arc from I to P_1.
- Output ports of the network are represented by a place and a transition like P_7 and O. Every transition that represents an action that sends tokens to that network output port has an outgoing arc of the corresponding weight to the place, e.g. the arc from A_2 to P_7.

The *initial marking* of the generated Petri net has zero tokens on every place (i.e. the input buffers of every actor are empty).[1]

Because each action is directly represented by a transition in the Petri net, any sequence of transition firings that return the Petri net to its original state corresponds to a cyclic schedule of actions. Furthermore, the number of firings of a transition representing a network input gives us the number of input tokens required by the schedule, and the number of firings of a transition representing a network output equals the number of tokens produced at that output.

[1] This can easily be generalized to situations where there are initial tokens in the network. Furthermore, our technique can also be adapted to compute the minimal initial marking required to find a cyclic schedule.

5.2 Computing the firing vectors

As a prerequisite to finding a sequence of transition firings, we will now compute the *firing vectors* of the Petri net. A firing vector contains a number of firings for each transition—consequently, its entries are non-negative integers, and its dimension is $n \times 1$, where n is the number of transitions. Each sequence of transition firings uniquely defines a firing vector which contains for each transition the number of its occurrences in that sequence. The inverse is not true—given a firing vector, there may be many legal firing sequences, including zero, that correspond to it. Later we will use this vector to search for a legal sequence of transition firings.

In order to compute the firing vectors of a Petri net, we need to construct its *incidence matrix*. For a Petri net with n transitions and m places, the incidence matrix A is an $n \times m$ matrix of integers and its entries are given by

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-} \tag{1}$$

where $a_{ij}^{+} = w(i,j)$is the weight of the arc from transition i to its output place j, and $a_{ij}^{-} = w(j,i)$ is the weight of the arc to transition i from its input place j. If there is no arc between i and j, then the corresponding weight is 0.

For example, the incidence matrix for the Petri net in figure 2 is

$$A = \begin{array}{c} \begin{array}{ccccccc} P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 \end{array} \\ \left[\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{array}\right] \end{array} \begin{array}{l} I \\ S_1^{Up} \\ S_1^{Down} \\ S_2^{Up} \\ S_2^{Down} \\ A_1 \\ A_2 \\ O \end{array} \tag{2}$$

Given an initial marking M_0, a firing vector $\mathbf{x}$ is related to the marking M resulting from a sequence of transition firings corresponding to it by the following state equation:

$$M = M_0 + A^T \mathbf{x} \tag{3}$$

Since we are looking for cyclic schedules, $M_0 = M$, and thus the firing vector becomes the (integer) solution of the equation:

$$A^T \mathbf{x} = 0 \tag{4}$$

Solving equation 4 for the matrix in equation 2, we get the firing vector

$$\mathbf{x} = \alpha \cdot [3,2,1,1,1,1,1]^T \tag{5}$$

where α is a scalar. Since we are only interested in non-negative integer solutions, α must be a non-negative integer.

Although the solution space of equation 4 is one-dimensional in our example, it can be multi-dimensional in the general case. This would give rise to multiple firing vectors that would be used as a starting point for finding a legal firing sequence.

If the initial marking M_0 of a Petri nets is fixed, then the existence of a firing vector $\mathbf{x}$ satisfying (4) is only a necessary but not a sufficient condition for the existence of a cyclic schedule [14].[2]

5.3 Finding legal firing sequences

A sequence of transition firings is *legal* if it does not produce a negative number of tokens on any place at any step. For example, the sequences

$$I—I—I—S_1^{Down}—S_1^{Up}—S_1^{Up}—S_2^{Down}—S_2^{Up}—A_1—A_2—O—O$$

and

$$I—I—I—S_1^{Down}—S_2^{Up}—S_2^{Down}—S_1^{Up}—S_1^{Up}—A_1—A_2—O—O$$

both correspond to the firing vector $[3, 2, 1, 1, 1, 1, 1, 1]^T$ that is a solution to the equation 4. However, because the second sequence would temporarily produce in a negative number of tokens on place P_2. This is why in general the existence of a firing vector does not imply the existence of a legal firing sequence.

The problem of finding legal firing sequences given a firing vector has been studied extensively—see e.g. [16]. It is known to be NP-complete in the general case, but polynomial-time solutions exist for special classes of Petri nets, and heuristics have been developed for finding legal firing sequences in acceptable time in more general networks (for example [17]). Which of the many algorithms is the most appropriate for our class of Petri nets, and whether we can make use of special structural properties of our Petri nets is still an open problem.

Once we have computed a legal firing sequence, we need to remove the input and output transitions from it to get a valid cyclic action schedule. The first sequence above would thus become the schedule

$$S_1^{Down}—S_1^{Up}—S_1^{Up}—S_2^{Down}—S_2^{Up}—A_1—A_2$$

6 Ambiguous schedules

In many cases, the procedure described in section 5 produces more than one possible schedule, i.e. it is ambiguous. There are basically two kinds of ambiguity: that which may affect the result computed by the network, and that which is guaranteed not to. Note that in our example, all schedules yield the same result, because addition is a commutative operation, and exactly how the two *Split* actors distribute the tokens does not affect the outcome. But suppose that the actor A_2 in figure 1 subtracted the lower input token from the upper input token instead. In that case, the schedule

[2] However, if we are able to choose M_0, then we can put enough tokens in each place to guarentee that transitions can fire as specified by the firing vector, and the existence of such a $\mathbf{x}$ becomes sufficient. We can then even compute the *minimal initial marking* required for a cyclic schedule. [15]

$$S_1^{Up}—S_1^{Up}—S_1^{Down}—S_2^{Up}—S_2^{Down}—A_1—A_2$$

would compute the value $x_1 + x_2 - x_3$ (where x_i is the i-th input token to the network), while

$$S_1^{Down}—S_1^{Up}—S_1^{Up}—S_2^{Up}—S_2^{Down}—A_1—A_2$$

would compute $x_2 + x_3 - x_1$ instead. On the other hand, the schedule

$$S_1^{Up}—S_1^{Up}—S_2^{Up}—S_2^{Down}—A_1—S_1^{Down}—A_2$$

always computes the same result as the first schedule. The reason is that each actor has the same *internal* schedule in both cases, i.e. S_1 is fired in the sequence *Up—Up—Down* and S_2 in the sequence *Up—Down* (for the other two actors the internal schedules are not very interesting as they only have one action).

We call two schedules *equivalent* if and only if they imply identical internal schedules for each actor. For example, the first and the second schedule above are not equivalent, because S_1 has the internal schedule *Up—Up—Down* in the first, and *Down—Up—Up* in the second schedule. Two equivalent schedules are guaranteed to result in the same output of the network.[3]

If the search for legal firing sequences only produces different equivalent schedules, the choice between them may still affect the efficiency of the generated code, e.g. with respect to the size of the buffers between actors. However, since in this case the internal schedules are entirely determined, we can use the techniques developed for cyclo-static dataflow networks to handle this case.

If the procedure in the previous section produces several non-equivalent schedules, we have the following ways of dealing with the situation:

1. Flag an error.
2. Display the possible classes of schedules and let the user choose one of them.
3. Let the user add constraints to the actors.

The first solution would be appropriate if the user declares that a unique schedule (up to equivalence) is required, and the second solution is only practical if the number of non-equivalent schedules is small. Picking a class of schedules effectively fixes all internal schedules for all actors, thus again reducing the problem to a cyclo-static scheduling problem.

The following sections will address the third solution.

7 Describing regular scheduling constraints in `Cal` actors

This section will explain how to express constraints on an actor's schedule in the `Cal` language. In the limit, such a constraint explicitly defines a full internal schedule—effectively yielding a cyclic actor. Doing this for every actor in the network would result in a cyclo-static network. Say, for example, we want to constrain S_1 to start with one execution of *Down*, we could do this by explicitly giving its schedule as

[3] Because fixing the internal schedule of an actor turns it into a determinate prefix-monotonic function on its input streams—cf. [11].

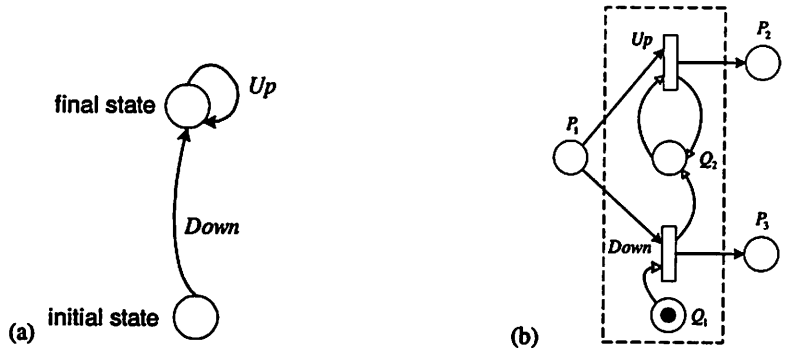

Fig. 3. The finite automaton (a) and its translation into the Petri net (b) of the constraint `Down Up*` for the `Split` actor.

```
Down Up Up
```

But we may want to be less restrictive than that. If we do not know how many tokens it will send up (and in more complex situations this might very well be a non-obvious result of the schedule), we may want to say that one execution of *Down* is followed by any number of executions of *Up*, e.g. like this:

```
Down Up*
```

A constraint that has the form of a regular expression (of action names) is called a *regular constraint*, and an actor that contains such a constraint is called a *regular actor*. In `Cal`, the `Split` actor with the regular constraint above would be written as follows:

```
1 actor SplitConstrained [T] ()
2   T Input ⟹ T Output1, T Output2 :
3
4   Up :      action [a] ⟹ [a], [] end
5   Down :    action [a] ⟹ [], [a] end
6   selector Down Up* end
7 end
```

In lines 4 and 5 the actions are *named*, and these names are used in the selector clause (line 6) as symbols in the regular expression.

Note that regular actors are a proper superset of both free and cyclic actors. If $A_1, ..., A_n$ are the action labels, then any fixed sequence in the A_i (corresponding to the internal schedule of a cyclic actor) is a valid regular expression, and the regular expression to describe a free actor would be

$$(A_1 \mid ... \mid A_n)*$$

The following section will describe the modifications to the ALS procedure from section 5 that are necessary to handle regular actors.

8 Regular action-level scheduling

Even though regular actors are significantly more expressive than free actors, the basic mechanism for finding a schedule is hardly any different. As in the free case, the actor network is mapped into a Petri net, for which we then compute a firing vector and search for legal firing sequences. The only differences are that the mapping of regular actors into pieces of the Petri net is slightly different from the free case, and also that the firing vector and the legal firing sequence are computed for different initial and final markings.

The basic idea of the mapping is still the same as in section 5—actions are represented by transitions, input buffers by places. In the regular case, however, we need to represent the regular constraint. As a first step, we transform the regular expression into a finite automaton that accepts the same sequence of labels (e.g. [7]), which we call the *actor automaton*. The automaton for the example from the previous section is depicted in figure 3a. Note that it has exactly an initial and a final state, and that the transitions are labeled with action names.

In order to construct the Petri net elements from a regular actor, we replace each state in its actor automaton with a place, and each state transition with a Petri net transition which represents the action that is indicated by the label. It is therefore connected accordingly, i.e. with incoming arcs from the input buffers that the action consumes from, and corresponding outgoing arcs. It also has an incoming arc from (place that represents) the state it comes from, and an outgoing arcs to the state it goes to. This mapping is shown for the constrained actor from the previous section in figure 3b. The places Q_1 and Q_2 correspond to the two states of the actor automaton.

While in the free case, the initial as well as the final marking were the empty marking (i.e. no tokens on any place), in the regular case the initial marking contains exactly one token on each place that represents an initial state of an actor automaton. Similarly, the final marking is empty except for one token in the final state of each actor automaton.

This does not affect the firing vector in the example (though it might very well do so in more complex cases). However, it does constrain the legal firing sequences: the transition representing S_1^{Up} is no activated until S_1^{Down} has fired, as the constraint dictates. This means that now

$$I—I—I—S_1^{Up}—S_1^{Up}—S_1^{Down}—S_2^{Down}—S_2^{Up}—A_1—A_2—O—O$$

is no longer a legal firing sequence, which is what we wanted to achieve.

9 Discussion and conclusion

In this paper we have presented a notion of dataflow actor that views actors as collections of *actions* which essentially describe different possible behaviors of the actor. When composing actors into actor networks, the (static) scheduling algorithm can choose from these behaviors whenever an actor is fired. This results in an additional degree of freedom because scheduling can now happen at the level of the individual actions, rather than the actors. We have shown how well-known techniques based on Petri nets can be applied to solve this problem.

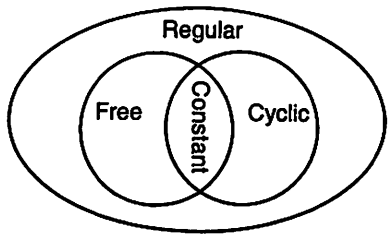

Fig. 4. Classes of actors.

Unfortunately, the additional degree of freedom is also a source of possibly harmful ambiguities. We have shortly discussed a criterion for detecting such ambiguities and then proposed a solution that allowed a user to gradually inject more constraints into the actor network. Even though in the limit this technique corresponds to specifying a cyclo-static dataflow network, it is usually possible to infer much of the scheduling information from a much looser description. This, too, could be solved by mapping the problem to an equivalent Petri net problem.

We introduced two new classes of actors, which we called *free* and *regular*. They relate to SDF actors (called *constant* here) and CSDF actors (called *cyclic* in this work) as shown in figure 4. Of course, both free and cyclic actors have constant actors as a special case; in fact, constant actors are precisely their intersection. As discussed in section 7, the set of regular actors properly contains both free and cyclic actors.

Even though the scheduling technique presented in this paper works for all regular actors, and hence also for the more specialized actor classes, it seems more appropriate to view it as a complement to more traditional scheduling techniques such as SDF or CSDF, rather than as a competing approach. While it allows a somewhat more general description of actors, it does not take into account optimizations such as buffer size or code size minimization. But since a schedule found by this technique can be abstracted into a CSDF or even an SDF specification, all the mechanisms developed in these contexts can be brought to bear on it.

This work can be built upon in a number of ways. As we have briefly discussed in section 5, the efficiency of the scheduling techniques depends on the existence of reasonably fast algorithms for finding legal firing sequences—this could be either good heuristics or solutions for special cases that frequently occur in practice. This has been a well-studied problem in the context of Petri nets, so one obvious next step is to apply the existing results from that field to our scenario. In addition, the structures typically occurring in networks of regular dataflow actors may very well give rise to more specialized heuristics that lead to even better solutions for those typical cases.

Another direction of work would use an action schedule computed for a network for generating code for that network—e.g. by employing a source-level transformation on the `Cal` descriptions that generates an `Cal` actor representing the network.

Further work is also needed in the context of handling ambiguities. The notion of equivalent schedules developed in section 6 could be extended to make use of knowledge

about the computation inside the actors, such as the commutativity of the Add actors in our example.

Acknowledgements

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent Technologies, Cadence Design Systems, Hitachi, National Semiconductor, and Philips.

References

1. The Ptolemy Project. Department EECS, University of California at Berkeley (*http : //ptolemy.eecs.berkeley.edu*).
2. Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, 1986.
3. John Davis, 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, EECS, University of California, Berkeley, March 2001.
4. Johan Eker and Jörn Janneck. Caltrop—language report. Technical Memorandum UCB/ERL ???/??, Electronics Research Lab, Department of Electrical Enginee and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002.
5. Marc Engels, Greet Bilsen, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow: Model and implementation. In 1994, editor, *28th Annual Asilomar Conference on Signals, Systems, and Computers*, pages 503–507, October-November.
6. Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, June 1977.
7. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
8. Jörn W. Janneck. *Syntax and Semantics of Graphs—An approach to the specification of visual notations for discrete event systems*. PhD thesis, ETH Zurich, Computer Engineering and Networks Laboratory, July 2000.
9. Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, Paris, France, 1974. International Federation for Information Processing, North-Holland Publishing Company.
10. R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete. Grape-2: A tool for the rapid prototyping of mutli-rate asynchronous DSP applications on heterogeneous multiprocessors. *IEEE Computer*, 28(2):35–43, February 1995.
11. Edward A. Lee. A denotational semantics for dataflow with firing. Technical Report UCB/ERL M97/3, EECS, University of California at Berkeley, January 1997.
12. Edward A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, January 1987.
13. Edward A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, September 1987.
14. Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

15. Toshimasa Watanabe, Yutaka Misubata, and Kenji Onaga. Legal firing sequences and minimal initial markings for petri nets. In *Proceedings ISCAS '89*, pages 323–326, 1989.
16. Toshimasa Watanabe, Yutaka Misubata, and Kenji Onaga. Legal firing sequence and related problems of petri nets. In *Proceedings of the Third International Workshop On Petri Nets and Performance Models*, pages 277–286. IEEE Computer Society Press, 1990.
17. Masahiro Yamauchi and Toshimasa Watanabe. A heuristic algorithm fsd for the legal firing sequence problem of petri nets. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'98)*, pages 78–83, October.