A machine model for dataflow actors and its applications

Janneck, Jörn

Published in:
Proceedings of the 45th Annual Asilomar Conference on Signals, Systems, and Computers

DOI:
10.1109/ACSSC.2011.6190107

2011

A Machine Model for Dataflow Actors and its Applications

Jorn W. Janneck
Department of Computer Science
Lund University, Sweden
jwj@cs.lth.se

***Abstract*—In application areas that process stream-like data such as multimedia, networking and DSP, the pipelined concurrent processing is frequently represented as a dataflow network of communicating computational kernels connected by FIFO queues. However, while dataflow is a natural medium for conceptualizing and modeling stream-processing systems, its adoption as a programming methodology has been hindered by an unappealing choice between expressiveness and efficient implementability—efficient implementation techniques being primarily limited to restricted subclasses of dataflow programs.**

The paper presents a simple machine model for a very general class of dataflow programs and shows how it can be used as a foundation for their efficient implementation.

I. Introduction

Describing systems as collections of computational kernels (*actors*) that communicate through FIFO-buffered sequences or *streams* of data packets (*tokens*) has a long tradition, especially in application areas characterized by the processing of stream-like data, e.g. [1], [2], [3], [4]. Recent years have seen a resurgence of interest in this kind of *dataflow* model of computation—for instance, MPEG and ISO adopted CAL dataflow programming language as part of their video coding standards. [5]

Implementing dataflow programs on sequential machines involves *scheduling* the computation performed within the various actors onto a single computing resource.[1] It is here that dataflow has so far offered an unattractive trade-off: For dataflow programs whose activities are amenable to compile-time scheduling, very efficient software synthesis techniques are available [3], [6] that gain much of their performance by eliminating all scheduling decision making from the runtime, and also by exploiting the regularities of accessing the buffers used to communicate between actors. However, the expressiveness of statically schedulable dataflow programs is limited to behavior that is independent of the values of the input data and of its timing. More general dataflow programs, such as Kahn process networks [2], or even those supported by the CAL language, are outside of the scope of these techniques. Accordingly, software synthesis technology for these kinds of dataflow programs has fallen short of being competitive with low-level hand-written code, in spite of some significant improvements in recent years. [7], [8], [9]

One technique for improving software synthesis for very general dataflow programs attempts to find regions of a dataflow program that are amenable to compile-time scheduling by analyzing the dataflow actors comprising the program. [10] Once such regions are found, existing algorithms for static scheduling can be applied to each of them in order to eliminate unnecessary runtime overhead from the synthesized code. This approach thus leverages techniques developed for specialized dataflow programs to improve software synthesis for the more general case, but is predicated on the existence of statically schedulable regions in a given dataflow program, as well as on the ability of tools to recognize them.

This paper presents a different take on this problem that is based on a simple machine model for actors. It turns out that one application for this model is an approach to generating efficient sequential implementations of arbitrary dataflow networks that naturally generalizes the techniques for statically schedulable dataflow. The machine model is also a promising starting point for actor analysis and optimization. Building on this model, we expect to create software synthesis for general dataflow programs that equals the current state of the art for statically schedulable programs, and which gracefully degrades for programs whose behavior depends on the value of input data and its timing.

II. Actors and action selection

The behaviors of the actors we are concerned with in this paper are defined by finite collections of *actions*, as for instance in the CAL actor language [11]. The execution of an actor is a sequence of action executions (*firings*). At each step in the sequence, an actor selects one of its *enabled* actions (if any—otherwise it needs to wait until at least one action becomes enabled), and then executes it. During its execution, an action can (a) read and consume input tokens, (b) produce output tokens, (c) modify the internal state of its actor if it has any. In addition to the code describing these activities, the

[1] In the more general case where a dataflow program is implemented on a parallel computing substrate, scheduling still remains essential as long as the number of computing elements is exceeded by the number of actors in the program.

description of an action also contains a definition of its *enabling conditions*. There are two kinds of enabling conditions—an action may require that one of the input queues to its actor, q, have at least a n tokens available, which we write as (q, n), and it may require that some *guard* predicate g (on the state of the actor and the input tokens) is true.

```
actor Split () A ⇒ P, N:

A1: action A: [v] ⇒ P: [v]
    guard v >= 0
    end

A2: action A: [v] ⇒ N: [v]
    guard v < 0
    end
end
```

Listing 1. A simple actor.

The above actor contains two actions, and their selection is controlled by three conditions: the condition that there be one input token available in the queue connected to port A, which we shall write as $(A, 1)$, and the two guards $g_1 : v \geq 0$ and $g_2 : v < 0$, where v in each case is the first input token.

Conditions are occasionally related to each other. For instance, for any input queue q, it is always true that $(q, n) \Rightarrow (q, m)$ if $n \geq m$ (in other words, the presence of n tokens in q implies the presence of less than n tokens in q). Also, if a guard depends on the value of the nth token of some input queue q (as is the case above), then it cannot be tested unless (q, n) is true. Furthermore, if $(A, 1)$ is found to be true, it will remain so until the actor fires an action, whereas if it is false, it may become true at some later point 'by itself' (due to the arrival of a token). We call the former situation *stable* and the latter *volatile*.

In this model, selecting the next action to fire amounts to testing sufficiently many conditions to know either (a) that no action can be fired or (b) identify at least one action that can be. For instance, in the above actor, we would start by testing the condition $(A, 1)$. If it is found to be false, we know no action can be fired, and that we need to wait. If it is true, we can then proceed to test with g_1 or g_2. Note that we cannot start with the guards, since both depend on $(A, 1)$.

III. Actor machines

An *actor machine* is an automaton that implements an actor description. Let that actor have the enabling conditions $C = \{c_i : i \in I\}$, and the actions $A = \{a_j : j \in J\}$. The actor also has internal state, but for the sake of brevity we will omit its formal treatment here (cf. [12] for a complete formal description) and focus on the action selection process only. An actor machine has a finite number of *controller states* Σ, and an initial controller state $\sigma_0 \in \Sigma$. It has an *interpretation function* $\mathcal{K} : \Sigma \longrightarrow C \longrightarrow \{X, 0, 1\}$ which maps each controller state to a function that assigns each condition a value from the set $\{X, 0, 1\}$. This function represents the knowledge the controller maintains about the conditions c_i of the actor, 1 meaning that the condition is true, 0 that it is false, and X that its value is unknown to the controller.

Each state $\sigma \in \Sigma$ is associated with a set $\mathcal{I}(\sigma)$ of *instructions* the machine can perform in that state. Possible instructions are $\mathbf{test}(c, \sigma_1, \sigma_2)$, $\mathbf{exec}(a, \sigma)$ and $\mathbf{wait}(\sigma)$ with $c \in C$, $a \in A$ and $\sigma, \sigma_1, \sigma_2 \in \Sigma$. The **test** instruction tests the specified condition, and causes the machine to proceed to state σ_1 if it is true, and to σ_2 otherwise. For the machine to be *consistent*, therefore, it must be the case that $\mathcal{K}(\sigma_1)(c) = 1$, that $\mathcal{K}(\sigma_2)(c) = 0$, and that $\mathcal{K}(\sigma)(c) = X$ for any σ such that $\mathbf{test}(c, \sigma_1, \sigma_2) \in \mathcal{I}(\sigma)$. Finally, all conditions c' that c depends on must be true, i.e. $Ctrl(\sigma)(c') = 1$. (In this case, c is called *testable* in σ.)

The **exec** instruction executes the specified action and the causes the actor machine to proceed to the specified controller state. During action execution, input tokens may be consumed, output tokens may be created, and the actor state (which is distinct from the controller state) may be changed.

The **wait** simply proceeds to the specified controller state. The purpose of this instruction is to erase knowledge about volatile conditions. Say we are in state σ, with $c = (A, 1)$, $\mathcal{K}(\sigma)(c) = 0$, i.e. the condition is false which means there is no input token available on A. If nothing else can be done to select an action, we might want to erase the information about the absence of input on A by transitioning to a state σ' such that $\mathcal{K}(\sigma')(c) = X$, so that we may retest the condition. Accordingly, if $\mathbf{wait}(\sigma') \in \mathcal{I}(\sigma)$, it must be the case that (a) $\mathcal{K}(\sigma)(c) = \mathcal{K}(\sigma')(c) \vee \mathcal{K}(\sigma')(c) = X$ for all conditions c and that (b) $\mathcal{K}(\sigma) \neq \mathcal{K}(\sigma')$.

Any state σ such that $\mathcal{I}(\sigma) = \emptyset$ is called a *terminal* state, and if the actor machine reaches a state like this its execution terminates. If an actor machine has (no) terminal states it is called (non-) terminating. An actor machine that has at most one instruction in each $\mathcal{I}(\sigma)$ is called a *single-instruction actor machine (SIAM)*.

The process of executing an actor machine, then, is very simple: in any state σ, pick one of the instructions in $\mathcal{I}(\sigma)$ (terminate if it is empty), execute the instruction, proceed to the appropriate subsequent state and repeat the process.

Fig. 1 depicts a simple actor machine for the actor in listing 1. The oval nodes represent controller states (the labels indicate the value of the interpretation function $\mathcal{K}$, by listing the values for the three conditions, $(A, 1)$ and the two guards g_1 and g_2 in that order), while the diamonds, boxes, and rings represent **test**, **exec** and **wait** instructions, respectively. The **test** instructions are labeled with the condition they are testing (C1, C2, C3 for the three conditions in the above order), and the continuous and dashed edges are the next states in case

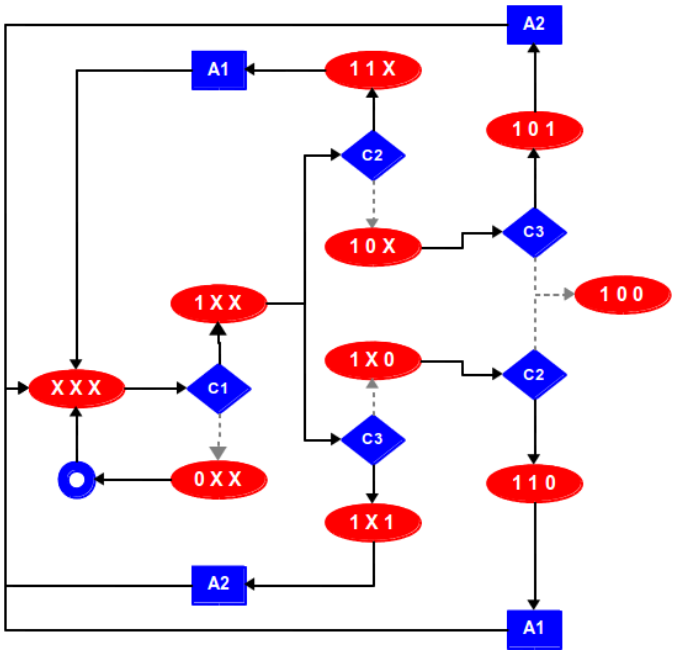

Fig. 1. Basic actor machine for the `Split` actor in listing 1.

the condition is found to be true and false, respectively. The **exec** instructions are labeled with the action they are executing.

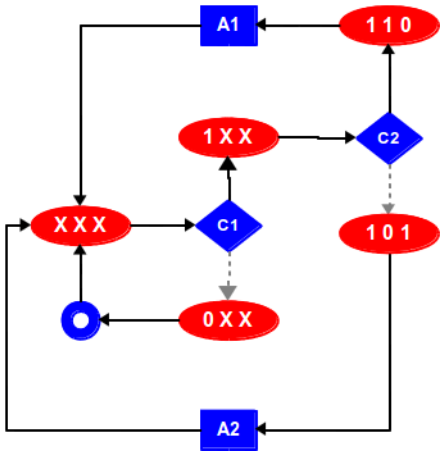

Fig. 2. Reduced actor machine for the one in Fig. 1.

However, this actor machine is a very poor implementation of the original actor. Some simplifications are very straightforward, such as fusing the states labeled [11X] and [110], as well as [1X1] and [101], since both pairs are assigned the same instruction and are thus equivalent. A more sophisticated simplification requires the insight that the two guards are logical inverses of each other. If this can be determined, we can reduce the actor machine to the one in Fig. 2. Note that the actor machine in Fig. 1 (falsely) appeared to be terminating, while its reduction in Fig. 2 is not.

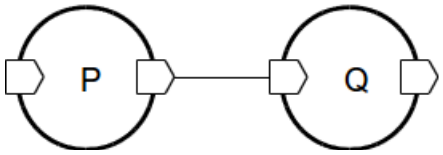

Fig. 3. A very simple actor network.

IV. Composition

One application of the actor machine model is the generation of sequential implementations of networks of actors. Consider the simple actor network in Fig. 3. One way of conceptualizing the task of mapping this network to a sequential processing element is as a *composition* of the actor machines representing the component actors in such a way that the resulting composite machine behaves in a way that is consistent with the behavior of the network. For the sake of brevity we will eschew a more detailed discussion of the notion of consistency here, other than pointing out that it does not, in general, imply equivalence—just as sequential implementations of dataflow networks typically involve some form of *scheduling*, which ends up restricting the observable behaviors, so may composition result in a composite actor machine that has a smaller set of behaviors. See [13] for a more thorough treatment of the composition of actors, and [12] for a more complete discussion of the composition of actor machines.

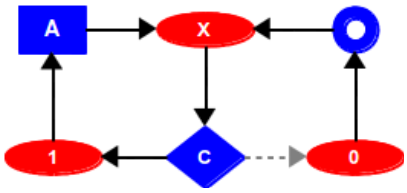

Fig. 4. The actor machine for `A` and `B` in listing 2.

```
actor P () In ⇒ Out:
    A: action In: [v] ⇒ Out: [f(v)] end
end

actor Q () In ⇒ Out:
    A: action In: [v, w] ⇒ Out: [g(v,w)] end
end
```

Listing 2. Two actors.

Let us assume that the two actors in Fig. 3 are defined as in listing 2.[2] They can both be implemented by the actor machine in Fig. 4, the only difference is that the condition C is $(In, 1)$ in the case of the actor machine for `P`, and $(In, 2)$ in the case of the machine for `Q`.

[2]This network is, of course, an SDF program. We have chosen this for the sake of simplicity—nothing in the following assumes a particular class of dataflow actor. Even a network that cannot be statically bounded can, in general, be subject to the techniques presented here.

When composing the actors of a network, all buffers associated with internal connections become state variables of the composite actor, and consequently, all input token conditions associated with these internal connections are now conditions on those internal state variables, i.e. they become guards. In the example, `Q.In`, the input queue of the actor `Q`, becomes a state variable, and accordingly the single condition associated with `Q`, $(Q.In, 2)$, becomes a guard.

One way of achieving the composition of the actor machines in a network is by building the product machine. Apart from the fact that this scales very poorly with the number of component actors, it also leads to a composite actor that is very nondeterministic, and that does not take advantage of the available knowledge about how the executions on the component actors relate to each other. For instance, in the example in Fig. 3, there is no need to test the input condition for `Q` until `P` has fired twice, and then it is automatically known to be true, so tests for that condition are, in this case, always redundant, even though the product machine would include the test instruction, and would in fact require its execution before the action of `Q` could be executed.

In order to improve on this, the controller states need to represent more information than just the interpretation function $\mathcal{K}$, particularly the size of the internal queues. Let $\mathcal{Q}$ be the set of the internal queues of the network, then we call $\mathcal{L} : \Sigma \longrightarrow \mathcal{Q} \longrightarrow \mathbb{N}$ is the *extended interpretation function* that assigns each state (in the composite actor) a function that maps each internal queue to a natural number indicating the number of tokens currently in that queue. Again we will skip the details, but in order for an extended interpretation to be consistent additional conditions apply, for instance that states connected by **wait** and **test** have the same extended interpretation function, and that the difference of extended interpretation of states connected by an **exec** instruction be consistent with the action executed in that instruction.

With this, the initial state of our example network might be written as [XX-0], i.e. the two conditions are unknown and the internal buffer is empty. Note, however, that the condition of `Q` is actually a condition on the number of tokens in the internal buffer, and if we know that it is empty, we know that this condition is false, in other words we can refine [XX-0] into [X0-0]. As long as we maintain accurate information about the size of the internal queues, we will always be able to determine the value of the input conditions related to them and thus will never have to test for the size of internal buffers.

Armed with this insight, we can now start to create the composite actor machine. We call the process used to do this *abstract simulation* in analogy to the common program analysis technique of abstract interpretation, because it essentially amounts to executing (or simulating) the dataflow network without knowing the values of the tokens being computed on, exploring multiple execution paths in the process.

```
todo := networkInitialState;
Σ := ∅;
while todo ≠ ∅ do
  s := choose(todo);
  todo := todo − s;
  Σ := Σ ∪ {s};
  if instructions(s) ≠ ∅ then
    i := choose(instructions(s));
    I(s) := i;
    todo := todo ∪ successors(i);
  end
end
```

Listing 3. Abstract simulation.

The algorithm above outlines the basic idea of abstract simulation. Starting with the initial state of the network (in the example, the state [X0-0]) in the todo list of states to be processed, we proceed until that todo list is empty, picking one of the states to be processed, adding it to the state space of the composite, and picking one of the instructions that can be executed in it, if it has any. Note that in many cases, several actors within the network have at least one instruction that could execute, so this choice amounts to some kind of scheduling process, and it will significantly affect how the state space is being explored. Once that instruction is chosen, we add all successor states to the todo list, after computing the proper internal queue lengths and resolving all internal conditions accordingly, as we did above for the initial state of the example.

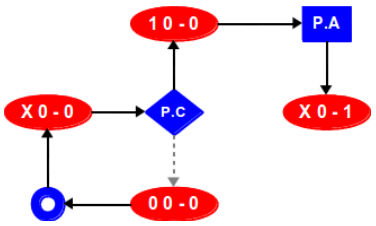

Fig. 5. The first few steps in the abstract simulation of the composition of Fig. 3.

Fig. 5 shows an intermediate result after a few steps of abstract simulation for our example. Starting from the initial state [X0-0], only one instruction can be performed, **test**$(P.C, [10-0], [00-0])$, i.e. testing the condition of `P`. Its successor state [00-0] again only admits a single instruction, **wait**$([X0-0])$, and its other successor state [10-0] only allows for executing the action of `P`, **exec**$(P.A, [X0-1])$. Note that the internal queue now has length 1, since the firing of the action produced one token that was sent into that queue. However, the input condition for `Q` remains false, since that condition requires two tokens.

Fig. 6 shows the result from a complete abstract simulation of the example. Note that after two firings

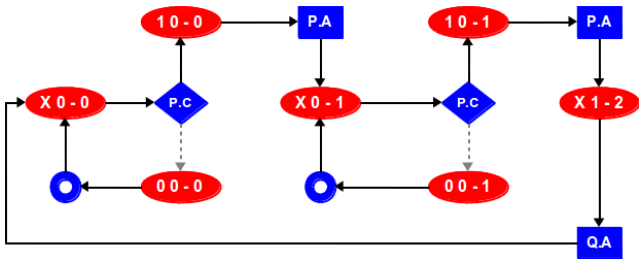

Fig. 6. The composite actor machine for the network in Fig. 3.

of P.A it reaches the state [X1-2], where the internal queue contains two tokens and the input condition of Q thus becomes true. At this point, there is actually a choice of instructions: the abstract simulation algorithm can execute the action of Q, as shown in the figure, or, alternatively, it could again test the input condition of P. In general, exact analysis or approximate heuristics will need to be employed to make this choice, and different choices might be preferable depending on the objective. In this case, choosing to fire Q will lead back to a previous state, which is usually desirable as it keeps the controller state space small.

Note that in this case abstract simulation in fact computed the SDF schedule for the composite, because that is really the only way in which to execute it (apart from the one choice mentioned above). In general, things need not always go this smoothly: since the state of the composite actor machine not only includes the states of the component machines, but also the length of the internal queues, the network state space can become very large very quickly, and is, in general, not even guaranteed to be finite. The latter problem would lead to nontermination of the abstract simulation algorithm, and can in general only be avoided by introducing some 'artificial' termination criterion, such as bounding the size of the internal queues and adding special error handling if an instruction is encountered that would violate those bounds. If keeping track of internal queue sizes proves to lead to an undue explosion of the state space, more abstract representations for queue sizes can be used that trade a smaller state space for more frequent testing of internal buffers, cf. [12] for more details.

Even though the example was a statically schedulable dataflow network, we never made any assumption that it was, and the technique seamlessly generalizes to arbitrary dataflow networks, effectively reducing to static scheduling techniques in limit-cases such as the one of the example.

V. Discussion and Conclusion

This paper presented a simple machine model for dataflow actors that captures the structure and the logic of selecting and executing the actions comprising an actor. It then illustrated how that model can be used to simplify and optimize the selection process based on analysis of the actor machine itself and of the actor description. Finally, it showed how actor machines can be the conceptual foundation of actor composition, and how that leads to the elimination of all testing of internal buffers. The expectation is that this will eventually facilitate the generation of very efficient code for arbitrary actor networks, alleviating the tension between expressiveness and efficient implementability of dataflow networks.

Many question still remain open, however. It is still unclear how this model behaves on at-size applications, and which techniques are required to best address state space explosion and unboundedness. Furthermore, the abstract simulation algorithm involves a number of choices that can have a large impact on the efficiency and size of the implementation derived from a composition, and much work needs to be done on building robust heuristics that make these choices in the general case, and to integrate exact analysis techniques that identify and handle more specialized situations, such as statically schedulable networks.

References

[1] J. B. Dennis, "First version data flow procedure language," MIT Lab. Comp. Sci., Technical Memo MAC TM 61, May 1975. 1

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*. North-Holland Publishing Co., 1974. 1

[3] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, 1987. 1

[4] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995. 1

[5] S. S. Bhattacharyya, J. Eker, J. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Journal of Signal Processing Systems*, 2009. 1

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Static scheduling of multi-rate and cyclo-static dsp applications," in *Workshop on VLSI Signal Processing*. IEEE Press, 1994. 1

[7] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software Code Generation for the RVC-CAL Language," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 203–213, 2009, 10.1007/s11265-009-0390-z. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0390-z 1

[8] A. Carlsson, J. Eker, T. Olsson, and C. von Platen, "Scalable parallelism using dataflow programming," in *Ericson Review, Online publishing www.ericsson.com*, 2011. 1

[9] "ACTORS Project." http://www.actors-project.eu/. 1

[10] R. Gu, J. W. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," in *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, ser. ICASSP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 565–568. 1

[11] J. Eker and J. W. Janneck, "CAL Language Report," Electronics Research Lab, University of California at Berkeley, Technical Memo UCB/ERL M03/48, December 2003. 1

[12] J. W. Janneck, "Actor machines — a machine model for dataflow actors and its applications," Department of Computer Science, Lund University, Tech. Rep. LTH 96-2011, LU-CS-TR 201-247, 2011. 2, 3, 5

[13] ——, "Actors and their composition," *Formal Aspects of Computing*, vol. 15, no. 4, pp. 349–369, December 2003. 3