On Tokens and Signals: Bridging the Semantic Gap between Dataflow Models and Hardware Implementations

Stavros Tripakis
Rhishikesh Limaye
Kaushik Ravindran
Guoqiang Wang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-164
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-164.html

June 8, 2012

On Tokens and Signals: Bridging the Semantic Gap between Dataflow Models and Hardware Implementations*

Stavros Tripakis
University of California
Berkeley, CA, USA
stavros@eecs.berkeley.edu

Rhishikesh Limaye Kaushik Ravindran
Guoqiang Wang
National Instruments Corp.
Berkeley, CA, USA
{first.lastname}@ni.com

June 8, 2012

Abstract

Dataflow models serve as useful abstractions of digital hardware in signal processing and other application domains. But when can one say that a certain dataflow model faithfully captures a given piece of hardware? The question is investigated in this paper. Finite state machines with synchronous semantics are used to model hardware. Asynchronous processes communicating via queues are employed for dataflow. A conformance relation between these two heterogeneous models of computation is developed that preserves performance properties such as worst-case throughput and latency.

1 Introduction

Dataflow is a model of computation where a set of concurrent processes communicate asynchronously by exchanging data through a set of (usually FIFO) queues. Dataflow is popularly used to specify, analyze, and implement multi-rate computations that operate on infinite streams of data. The expressiveness of the dataflow model in naturally capturing streaming applications, coupled with its strong formal analysis properties, has made it prominent in modern embedded systems design, particularly in the domains of multimedia, digital signal processing, and communication.

One of the typical uses of dataflow in embedded system design is to model systems implemented in hardware (HW). The primary motivation for this is the fact that many dataflow models, such as SDF [20], CSDF [2], SADF [27] admit efficient static compile time methods for computing key performance metrics of an embedded system, such as throughput, latency, or buffer sizes. In principle these metrics could be also computed at the (cycle-accurate) HW level (e.g., VHDL or Verilog programs). In fact, doing so presents several advantages over doing the same analysis at the dataflow level: (1) it does not require building dataflow models; (2) it is more accurate, as it operates directly on the HW description. But it also suffers from a

*This work was supported in part by NSF Award #1138996 *Expeditions in Computer Augmented Program Engineering (ExCAPE)*, and by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from NSF (awards #1035672 (CPS: Medium: Timing-Centric Software), #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs)), the U. S. Army Research Office (ARO #W911NF-11-2-0038), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

major disadvantage, namely, *state explosion* due to huge state spaces. This makes it infeasible in practice. Analysis of dataflow models, on the other hand, is much more efficient because these models abstract much of the information contained in the HW descriptions. For example, they typically omit data values and use only abstract notions of *tokens*. The timing properties of the system are often independent of data values. In cases where they are dependent, abstraction techniques can be used to build dataflow models that have no data values, yet conservatively approximate the behavior of the HW (e.g., using non-determinism).

Still, two questions remain, namely: (1) how to build a dataflow model for a given piece of HW, and (2) how to ensure that the model is "faithful", or at least conservative, with respect to the original HW. In fact, it is unclear what faithful even means. A prerequisite for answering (1) and (2) is to make the notion of faithfulness precise, and this is the question that concerns us in this paper.

When attempting to define faithfulness, we are faced with the following difficulty. The dataflow model of computation is very different from the model of computation of synchronous digital hardware. To model the latter, one typically uses finite state machines (FSMs) with *synchronous* semantics. An FSM communicates with its environment by means of input/output *signals* that take Boolean values in a succession of synchronous rounds. In the dataflow model of computation, on the other hand, a set of concurrent processes execute *asynchronously* and communicate by means of consuming and producing *tokens* from/to a set of queues. It appears that the two models "live in different worlds" and that comparing them is a bit like comparing apples and oranges.

In this paper, we study this comparison problem. Our goal is to bridge the semantic gap between dataflow models and HW implementations. We do this by defining a formal *conformance* relation between FSMs (modeling HW) and *dataflow processes*, which we propose as a formal operational model for dataflow. Dataflow processes have a notion of time that we use to map to HW clock cycles.[1] In addition, we require explicit signals at the HW level that allow us to observe token production and consumption events that are primitive events at the dataflow level. Conformance is then defined with respect to a mapping of HW signals to the above events, which allows to translate HW behaviors to dataflow behaviors.

In the rest of the paper, and after discussing related work, we briefly review FSMs and their composition in Section 2 and propose an operational process model for dataflow in Section 3. We present a conformance relation between FSMs and dataflow networks in Section 4, discussing the rationale behind the definition and illustrating the concepts through a series of examples. Conclusions and plans for future work are presented in Section 5.

1.1 Related Work

Prior research has extensively studied methods to generate (HW or SW) implementations from dataflow models. Algorithmic solutions have been developed for joint code and buffer size optimization, throughput computation, buffer sizing under throughput constraints, and schedule computation [1] [20] [22] [26] [32] [18]. The focus of these solutions is predominantly to deliver software implementations for processor targets. Hardware generation from dataflow models has also been extensively studied, e.g., in [19, 33, 12, 15, 13, 28, 5, 23]. The goals of that line of work are akin to those of high-level synthesis, namely, obtaining efficient HW implementations automatically from high-level descriptions. Even if we admit that these methods are *correct-by-construction*, in which case the resulting implementation is guaranteed to conform to the high-level description, there is still a need to explicitly define conformance, something missing from

[1] There are also untimed dataflow models (e.g., untimed SDF [20]). Timed properties such as throughput cannot be evaluated on untimed models. It is therefore hard to see how one could bridge the gap between a timed and an untimed model, while at the same time preserving such properties. For this reason, we work with a timed dataflow model.

the above works. An explicit notion of conformance is useful in the context of high-level synthesis, for instance, in order to catch compiler bugs. But conformance is also useful in other contexts, for instance, when abstract models are used to estimate performance of an existing HW system (e.g. [10]), or in the context of IP integration (e.g. [29]).

The problem of bridging the semantic gap between hardware and higher-level models arises in many abstraction-based design and verification methodologies, such as *transaction-level modeling* (TLM), e.g. [9] or *equivalence checking between system-level and RTL models*, e.g. [24]. A rigorous formalization of the relation between the concrete (RTL) and the abstract (transaction- or system-level) models is often missing in such methodologies, and it is unclear how such a relation could be defined, since the models "live in different semantical worlds" (e.g., clock cycles vs. transactions). Indeed, the abstract models are often untimed C programs and the focus is to check functional equivalence within a cycle [4].

The works [10, 29] pursue goals similar in spirit to this paper, however, they do not define a formal conformance relation. [10] presents a method for building conservative dataflow models of a specific class of network-on-chip channels. Our work aims to be more general, and applicable to general hardware modeled as FSMs. The main focus of [29] is the synthesis of glue, and the notions of correctness and non-defensiveness between models and systems are defined with respect to the glue (e.g., whether buffer sizes estimated by the model are overly pessimistic or optimistic).

Formal conformance relations abound in the field of formal verification, such as trace inclusion, simulation, bisimulation, and so on (see, for instance, [31, 30]). However, these works typically relate processes that "live in the same world", in other words, follow the same model of computation. In contrast, we develop a conformance relation between two heterogeneous models that preserves key execution properties.

A formal refinement relation for a model of actors has been proposed in [8]. Actors are viewed as relations between input and output timed traces and the refinement relation preserves worst-case throughput and latency properties. Our work pursues goals similar to those pursued in that paper, however, there are differences. The primary difference is that [8] uses an abstract, denotational model of actors, which does not answer the question how to map the semantic gap between tokens and signals. Here we use operational models for both dataflow and hardware, and directly consider how to map signals to tokens. A secondary difference is that the refinement relation used in [8] is based on the "earlier the better" principle, whereas here we employ the more traditional principle of subset of behaviors. More discussion on the relation to [8] is provided in Section 5.

2 A Model for Hardware

We model hardware as *finite-state machines* (FSMs) and in particular *Mealy machines* [17].[2] An FSM is a tuple $M = (X, Y, S, s_0, \delta, \lambda)$, where:

- X is a finite set of Boolean variables, called the *input signals* of M.
- Y is a finite set of Boolean variables, called the *output signals* of M.
- S is a finite set of states.
- $s_0 \in S$ is the initial state of M.

[2] For simplicity, we use deterministic FSMs. However, the results, and in particular the definition of conformance, directly extend to non-deterministic FSMs as well.

- $\delta : S \times 2^X \to S$ is the *transition function* of M: it takes a state $s \in S$ and an input assignment $a \in 2^X$ and produces a next state $s' = \delta(s, a) \in S$. An assignment is a function that assigns a value to each of a set of variables. An input assignment is a function $a : X \to \{0, 1\}$ that assigns a Boolean value to each input signal. δ is a *total* function meaning it is defined for any $s \in S$ and $a \in 2^X$.
- $\lambda : S \times 2^X \to 2^Y$ is the *output function* of M: it takes a state $s \in S$ and an input assignment $a \in 2^X$ and produces an output assignment $b = \lambda(s, a) \in 2^Y$. An output assignment is a function $b : Y \to \{0, 1\}$ that assigns a Boolean value to each output signal. λ is a total function. For $y \in Y$, we define $\lambda_y : S \times 2^X \to \{0, 1\}$ to be the function that returns a Boolean value for output signal y, given the current state and inputs. That is, $\lambda_y(s, a) = (\lambda(s, a))(y)$.

An FSM M is *closed* if its set of input signals is empty, i.e., $X = \emptyset$. In that case, the transition and output functions become simply functions of S: $\delta : S \to S$ and $\lambda : S \to 2^Y$. If $X \neq \emptyset$ then M is called *open*.

An FSM M is a *Moore machine* if the value of each one of its output signals only depends on the current state and not on the inputs, that is, λ is only a function of S: $\lambda : S \to 2^Y$. Clearly, every closed FSM is a Moore machine. More generally, we will say that a certain output signal $y \in Y$ is a *Moore output of* M if the value of that output only depends on the current state (whereas the value of other outputs may also depend on the inputs), that is, λ_y is only a function of S: $\lambda_y : S \to \{0, 1\}$. Clearly, M is a Moore machine iff every output of M is a Moore output.

2.1 FSM semantics

An FSM M defines a set of behaviors of the form

$$s_0 \xrightarrow{a_0/b_0} s_1 \xrightarrow{a_1/b_1} s_2 \xrightarrow{a_2/b_2} \cdots$$

where $s_i \in S$, $a_i \in 2^X$, $b_i \in 2^Y$, $s_{i+1} = \delta(s_i, a_i)$ and $b_i = \lambda(s_i, a_i)$, for all i. Intuitively, at synchronous clock cycle i, if the current state is s_i and the current inputs are a_i, then the current outputs are b_i and the next state (at clock cycle $i + 1$) will be s_{i+1}. We say that the sequence $(a_0, b_0)(a_1, b_1) \cdots$ is an *observable behavior* of M.

2.2 FSM example

An example of an FSM is shown in Figure 1. The top part of the figure shows the structure (or "black-box" view) of the FSM, namely, its name M_{buf}, its set of input signals $\{\mathsf{v}_1, \mathsf{r}_2\}$ and its set of output signals $\{\mathsf{r}_1, \mathsf{v}_2\}$. The bottom part of the figure shows the behavior of the FSM, namely, its set of states, initial state, and transition and output functions. M_{buf} models a simple buffer of size one. It has two states, denoted s_0 and s_1, representing the fact that the buffer is empty and full, respectively. s_0 is the initial state. The assignment expressions inside the state represent the output function. For example, $\mathsf{r}_1 := 1$ at state s_0 specifies that r_1 is set to true when M_{buf} is in that state (in this case, r_1 does not depend on the inputs), and $\mathsf{v}_2 := \mathsf{v}_1 \wedge \mathsf{r}_2$ specifies that v_2 is set to the logical conjunction of the two inputs.

Intuitively, the operation of M_{buf} is as follows. Initially, the buffer is empty and declares it is ready to receive input by setting r_1 to 1. A writer may request to write something to the buffer (provided $\mathsf{r}_1 = 1$) by asserting v_1. If this is done, there are two cases: either a read is also requested simultaneously, by having $\mathsf{r}_2 = 1$; or no read is requested at this time, i.e., $\mathsf{r}_2 = 0$. In the former case, the buffer acts as a "wire", letting the input "flow through" the output: v_2 is set to 1 and the buffer continues to be empty. In the latter case, v_2

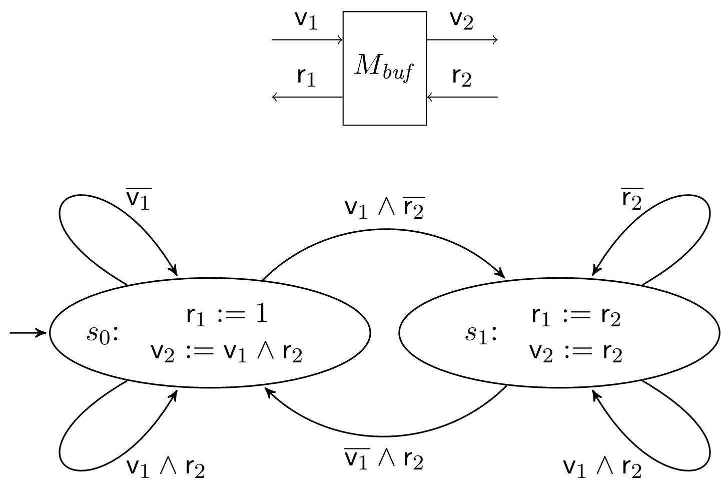

Figure 1: Example FSM: structure (top) and behavior (bottom).

is set to 0 and the buffer moves to s_1. The behavior at s_1 is analogous. Notice that data values are abstracted away in this FSM, and only control signals are captured.

2.3 FSM composition

FSMs can be composed with other FSMs. Different composition operators can be considered: parallel composition (putting two FSMs "side by side"), serial composition (connecting an output signal of one FSM to an input signal of another FSM), feedback composition (connecting an output signal of an FSM to one of its input signals), and so on. The FSM model is *compositional* in the sense that, under quite mild conditions, the composition of a set of FSMs (with respect to any of the above composition operators) defines an FSM.

The conditions are imposed to avoid problems of *cyclic dependencies* during feedback composition: the fact that the value of a signal may depend on itself. To avoid this, a typical condition is to require that in order to form a feedback loop connecting an output signal y to an input signal x, y must be a Moore output.

We will not define FSM composition formally, as it is standard. Instead, we give an example. Consider the composition of the three FSMs shown in Figure 2. M_{buf} is the FSM from Figure 1, while the behaviors of M_1 and M_2 are shown in Figure 2. The composite FSM M is shown at the bottom of the figure. M is the synchronous composition of M_1, M_{buf} and M_2, denoted $M_1 \times M_{buf} \times M_2$. M has no input signals: all its four signals r_1, v_1, r_2, v_2 are outputs. Therefore, by definition, M is a Moore machine. The vectors in the lower half of each state denote the values of the four output signals r_1, v_1, r_2, v_2 in that state. Each state of M is a composite state, that is, a vector describing the local states of the components of M. Since M_1 is *stateless* (it has a single state that never changes) we omit its state from the composite vector and include only the states of M_{buf} and M_2. Thus, state s_{12} of M represents the fact that M_{buf} is at state s_1 and M_2 is at state 2 (we suppose that the states of M_2 are numbered $0, 1, 2$).

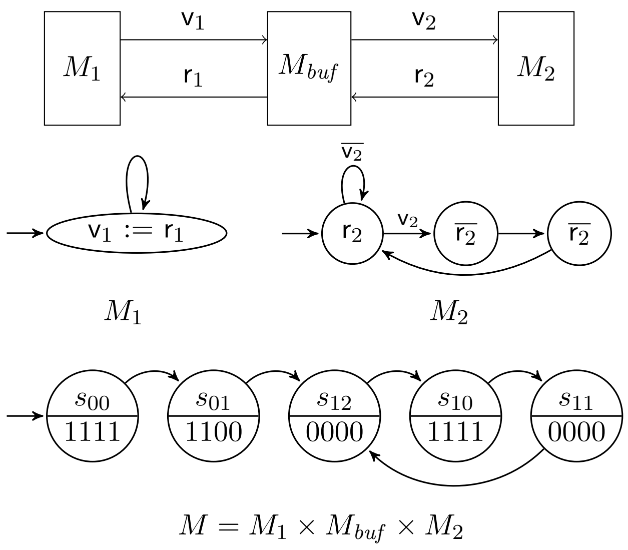

Figure 2: Closed FSM M obtained by composing FSMs M_1, M_2 and M_{buf} of Figure 1. The vectors in the lower half of each state denote the values of the four output signals $\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2$ in that state.

3 A Model for Dataflow

A variety of formal models for dataflow systems exist in the literature, e.g., see [16, 6, 14, 7, 26, 8], although they are not as standard as FSMs are for hardware. The operational model we present here is in the spirit of those proposed in [7, 26]. The model in [7] is untimed, whereas our model includes time. On the other hand, our model abstracts away from data which are included in [7]. The model in [26] is timed and data-abstract, but it is limited to CSDF graphs, whereas our model captures a broader class of dataflow process (in fact, broader than Kahn Process Networks, or KPNs [16]).

Time is typically introduced in dataflow models by means of a special action denoted tick, modeling the lapse of one unit of time [25, 26]. We follow the same approach. Specifically, we model a dataflow system using two types of components:

- *Processes:* These are finite-state automata whose transitions are labeled with actions of the following three types: get_i (get token from the i-th input queue), put_i (put token into the i-th output queue), or tick (one time unit elapses).
- *Queues:* These are essentially counters counting the number of tokens in the queue at a given point in time. put actions increment the queue's counter by one. get actions decrement the queue's counter by one when the counter is greater than zero, otherwise get is not possible. A queue may be unbounded which means the counter can grow arbitrarily large, yielding an infinite-state automaton; or the queue may be bounded meaning the counter can only grow up to a given constant K, at which point put is no longer possible.

The above models abstract away from data and the functional aspects of dataflow. They only maintain information on production/consumption of tokens and timing, which is our focus in this paper.

Formally, a dataflow process is modeled as an automaton $A = (n, m, S, s_0, \rightarrow)$ where:

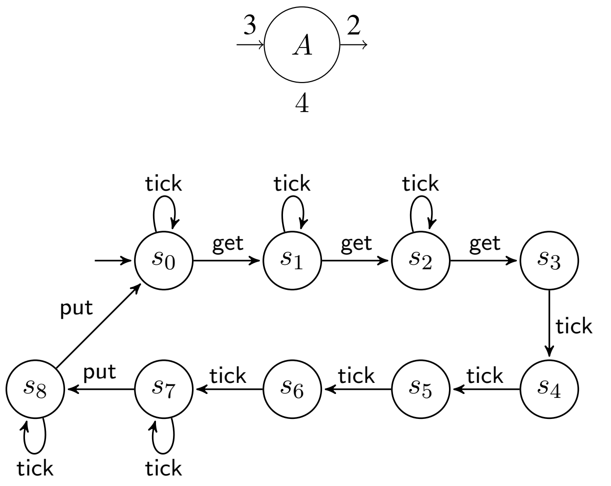

Figure 3: Example SDF process: structure (top) and behavior (bottom).

- $n \geq 0$ is an integer representing the number of *input ports* of A. Each input port will be connected to an input queue.
- $m \geq 0$ is an integer representing the number of *output ports* of A. Each output port will be connected to an output queue.
- S is a set of states (not necessarily finite).
- $s_0 \in S$ is the initial state of A.
- $\rightarrow \subseteq S \times L \times S$ is the *transition relation* of A, where the set of labels L is defined as follows:

$$L = \{\mathsf{get}_1, \mathsf{get}_2, ..., \mathsf{get}_n, \mathsf{put}_1, \mathsf{put}_2, ..., \mathsf{put}_m, \mathsf{tick}\}$$

  A transition $(s, \ell, s') \in \rightarrow$ is also denoted $s \xrightarrow{\ell} s'$.

Remark 1 *Although our examples are simple dataflow processes that fall in the SDF or KPN classes, the modeling framework as well as the conformance relation defined in Section 4 are more broadly applicable. In particular, contrary to what is customary [7], we make no assumptions on determinism or confluence of the transition relation $\rightarrow$ of a dataflow process. For instance, it is allowed to have a process with multiple transitions $s \xrightarrow{\mathsf{get}_1} s_1$ and $s \xrightarrow{\mathsf{get}_2} s_2$ emanating from the same state s. This would typically be interpreted as the process choosing non-deterministically to read from channel 1 or from channel 2, something which is not allowed in Kahn Process Networks. It is also possible to have non-determinism in the successor states, e.g., $s \xrightarrow{\mathsf{get}_1} s_1$ and $s \xrightarrow{\mathsf{get}_1} s_1'$, with $s_1 \neq s_1'$. These types of non-determinism are useful, for instance, when abstracting data-dependent behavior.*

An example dataflow process is shown in Figure 3. A is an SDF process with a single input queue and a single output queue, represented by the incoming and outgoing arrows of A, respectively. A repeatedly consumes 3 tokens and then produces 2 tokens, as indicated by the numbers annotating the arrows. Each such repetition is called a firing of A. The firing lasts for 4 time units, as indicated by the number below

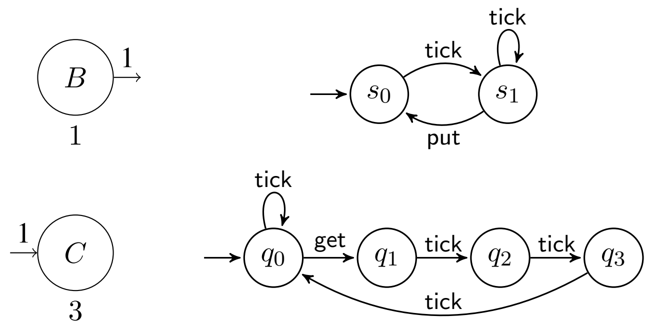

Figure 4: Source SDF process (top) and sink SDF process (bottom).

A in the figure. That is, from the moment the last of the 3 input tokens is consumed, until the moment the first of the 2 output tokens is produced, in a given firing, 4 time units elapse. This behavior is specified at the bottom of Figure 3. A has nine states, labeled $s_0, ..., s_8$. A waits at state s_0 until there is a token to consume, in which case the get transition occurs representing consumption of one token, and moving A to state s_1. For simplicity, we write get instead of get_1, since there is only one input queue. Similarly we write put instead of put_1. After all three tokens have been consumed, A is at state s_3. The next four transitions are labeled with tick actions, representing the passage of time. Once four time units have elapsed, A is at state s_7 and is ready to output tokens, which is represented by transitions labeled with put actions. After producing two tokens, A returns to its initial state for a new firing.[3]

Note that states s_7 and s_8 have self-loop tick transitions, as do states s_0, s_1, s_2. Such transitions are perhaps to be expected in states s_0, s_1, s_2, since A receives its input tokens from an input queue, which might be empty. As long as the input queue is empty, A must wait, therefore, it must allow time to elapse at these states. The situation is similar in states s_7 and s_8: even though queues in dataflow semantics are typically considered to be of unbounded size, in which case put actions can never be blocked, it is often useful, as we shall see below, to consider an alternative semantics where queues are bounded. In that case, put may block when a queue is full, and in that case time must be allowed to elapse.

A dataflow process may have no input queues, in which case it is called a *source*, or no output queues, in which case it is called a *sink*. Examples of SDF source and sink processes are shown in Figure 4.

Note that Figures 3 and 4 are simply examples, and do not prescribe a way to capture SDF as dataflow processes. In fact, as we shall see, there are different ways to model SDF operationally, and this is part of the challenge in coming up with faithful models.

An example of a non-deterministic dataflow process is shown in Figure 5. This process has one input and two output ports. After reading from its input, the process can non-deterministically choose two courses of action: either to write to port 1 after two time units, or to write to port 2 after one time unit. Such non-determinism is often the result of *data abstraction*. For example, consider a Kahn process which reads a concrete value, tests this value, and based on the result of the test chooses to perform different types of computation (requiring longer or shorter execution times) and write to different output ports. Such a process can be captured as in Figure 5, where the test is replaced by a non-deterministic choice.

[3] For simplicity, in our examples we assume no *auto-concurrency*, that is, no overlapping of firings of the same process. Auto-concurrency can be captured in our model using more elaborate and potentially infinite-state processes.

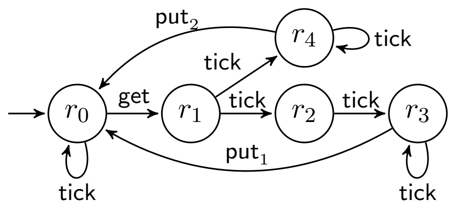

Figure 5: A non-deterministic dataflow process.

3.1 Dataflow process semantics

A dataflow process A defines a set of behaviors of the form

$$s_0 \xrightarrow{\ell_0} s_1 \xrightarrow{\ell_1} s_2 \xrightarrow{\ell_2} \cdots$$

where $s_i \in S$, $\ell_i \in L$, and $s_i \xrightarrow{\ell_i} s_{i+1}$, for all i. Intuitively, from state s_i, the process can perform action ℓ_i and move to state s_{i+1}. If $\ell_i = \mathsf{tick}$ then this action represents the passage of one time unit. Otherwise, the action is instantaneous. Action get_i means that A removes a token from its i-th input queue. Action put_i means that A adds a token to its i-th output queue.

As we did for FSMs, we will define a concept of observable behaviors for dataflow. This is a little more involved to do for dataflow than for FSMs because in the case of dataflow, consecutive put and get actions that are not "interrupted" by ticks are considered to be instantaneous. Therefore, it is reasonable to group all such actions together in a set. We will do this, and define an *observable behavior of* A to be a sequence $\alpha_0\alpha_1\cdots$ obtained by a behavior ρ of A, such that α_i is either tick or a set of consecutive put and get actions in ρ. For example, if

$$s_0 \xrightarrow{\mathsf{tick}} s_1 \xrightarrow{\mathsf{put}} s_2 \xrightarrow{\mathsf{get}} s_0 \xrightarrow{\mathsf{tick}} s_1 \xrightarrow{\mathsf{get}} s_2 \xrightarrow{\mathsf{put}} \cdots$$

is a dataflow behavior, then the corresponding observable dataflow behavior is

$$\mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdots .$$

3.2 Queues

Dataflow processes communicate via FIFO queues. In our model, data is abstracted away, therefore, the FIFO property of such queues is irrelevant, and does not have to be modeled. Therefore, we can easily model queues as counters that count the number of tokens currently in the queue. We can capture such counters using the same formalism as for processes. For example, the processes for an infinite queue and for a finite queue are shown in Figure 6. Queues are assumed to have an implicit self-loop transition labeled tick at every state: we omit these self-loops from the figures for the sake of simplicity.

3.3 Closed and open dataflow networks

A *dataflow network* is a collection of dataflow processes connected via queues. A dataflow network is *closed* if every input port of every process in the network is connected to some output port. This includes the ports get and put of queue processes, which are both inputs, since a queue is essentially a "passive" object: it

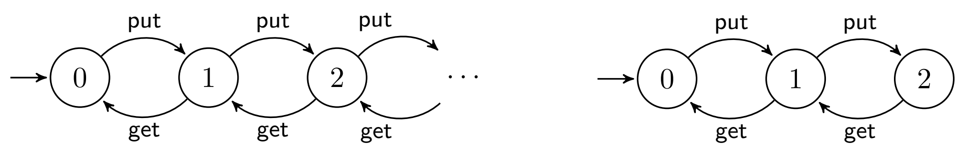

Figure 6: Queue processes: infinite queue (left) and queue of size 2 (right).

waits for a writer process to perform a put or for a reader process to perform a get, and it may sometimes disallow these actions (when full or empty), but it cannot initiate them.

For example, the network shown in Figure 7 is closed. If we removed C, however, it would be open. A network containing only process B would be closed. A network containing only process A of Figure 3, however, would be open.

3.4 Dataflow composition

Having obtained formal behavioral models for dataflow processes and for queues, the semantics of a dataflow network can be captured as the composition of the individual processes and queues. This composition can be defined as a standard composition of processes with *rendez-vous* communication in the style of CCS [21] or CSP [11]. In particular:

- a get action of a dataflow process A synchronizes with the get action of the process of the corresponding input queue of A;
- a put action of a dataflow process A synchronizes with the put action of the process of the corresponding output queue of A;
- tick actions synchronize across all processes in the network.

A composite process obtained by following the above rules is *maximal* in the sense that it contains all possible behaviors of a network. Maximality is important to have in an open network, that is, one that could be further composed (see paragraph below for a formal definition of open and closed networks). On the other hand, in a closed network, maximality may sometimes result in including behaviors that are not interesting or not optimal from a performance perspective. We may therefore need to exclude such behaviors. In order to do this, we define two composition semantics, obtained by restricting the maximal set of behaviors by adding extra rules.

Non-idling semantics

This semantics is obtained by computing the composition according to the above rules, and then removing all self-loop transitions labeled with tick, except if such a transition is the only one left at a given state. Indeed, such transitions represent *idling* where time passes without any process doing something useful.

Eager semantics

Non-idling semantics guarantees absence of idling but often we require something more, namely, that processes consume and produce tokens *as soon as possible*. In order to obtain this *eager* semantics, we additionally impose the following rule: a tick action is allowed at a given state only when no other action is possible.

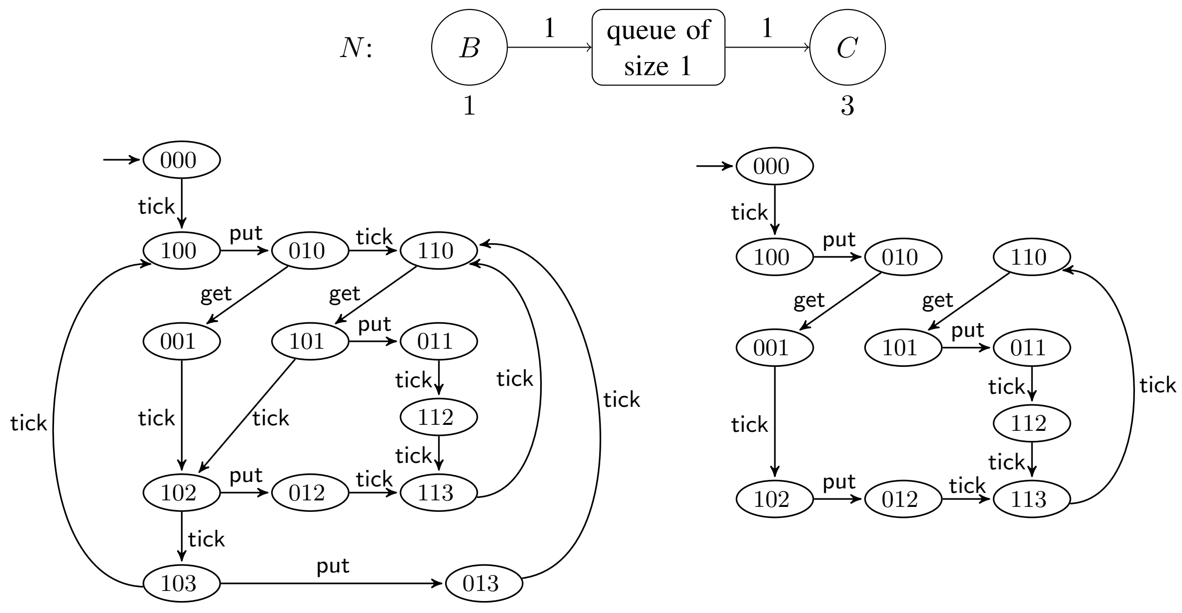

Figure 7: A closed dataflow network N (top) and the corresponding composite dataflow processes: non-idling (bottom-left) and eager (bottom-right).

Example

As an example, a dataflow network is shown at the top of Figure 7. It consists of the two SDF processes B and C of Figure 4 connected via a queue of size 1. The non-idling and eager composite processes obtained for N by following the rules described above are shown at the middle and bottom of Figure 7, respectively. The states of the composite processes are product states, that is, vectors consisting of one element state for each process in the network. To save space, we write ijk for a composite state instead of (s_i, j, s_k). Thus, 010 represents product state $(s_0, 1, q_0)$ where B is at state s_0, the queue is at state 1 (i.e., contains one token) and C is at state q_0. Notice that the eager semantics has no tick transition from that state, whereas the non-idling semantics has one.

4 Conformance

Having defined formal models and semantics for hardware and dataflow, we proceed in attacking our main problem, which is to define a formal conformance relation between the two. We are immediately faced with a difficulty. FSMs and dataflow processes are different mathematical objects, with heterogeneous semantics. How to compare them?

To overcome this difficulty, we take a pragmatic approach. Before defining conformance, let us recall that dataflow models are usually employed for estimation of timing and performance properties of the HW system. We examine such properties first, and then define conformance.

4.1 Timing properties

At the dataflow level, timing properties can be defined by referring to basic events: token consumptions, token productions, and the passage of time. More specifically:

- throughput can be defined by measuring how many tokens are produced within a given window of time (or the limit of such);
- latency can be defined by measuring the amount of time that elapses between the consumption and production of certain tokens;
- timing properties refer to which points in time certain consumptions or productions may or may not occur.

For example, consider the SDF network N shown in Figure 7. We can define throughput as the asymptotic average of the number of tokens consumed by C per unit of time. In the behaviors of N, consumptions are represented by get actions and time units by tick actions. Therefore, for a given behavior, we can compute the throughput by counting the average number of gets per number of ticks. As we can see from the composite processes for N shown in Figure 7, different behaviors achieve different throughput. In the non-idling process, there are behaviors that achieve throughput $\frac{1}{3}$ but also others that achieve throughput $\frac{1}{4}$. In the eager process there is only one behavior that achieves the optimal throughput $\frac{1}{3}$.

As for latency, we can define it as the time delay between the production of a token by B and the next corresponding consumption by C. This delay is not constant: it depends not only on the behavior of N, but it can also vary at different points within a behavior, for different productions and consumptions. In the case of the example of Figure 7, the worst-case latency between a put and a get is equal to 3 ticks, and the best-case latency is 0 ticks.

4.2 Conformance for closed systems

Having seen examples of typical properties that we are interested in, let us return to the question of conformance. In this paper we tackle this question in the case of closed systems. The case of open system is the subject of future investigation (see Section 5).

Suppose we want to compare a closed dataflow network such as the one of Figure 7 with a closed FSM. When should one say that the FSM conforms to the dataflow network? A standard principle for defining conformance in behavioral models is that of *containment* of sets of behaviors: a certain model M_1 conforms to another model M_2 if the set of all possible behaviors of M_1 is a subset of the set of behaviors of M_2.

We would like to apply the above principle in our setting. However, we are still faced with the problem that the behaviors of dataflow and FSM models are not directly comparable. In particular, although time elapse is observable from the behaviors of FSMs (by simply counting the number of transitions), token productions and consumptions are not directly observable at the FSM level. Indeed, it is not clear, by looking at the input and output Boolean signals of an FSM as they take values across successive clock cycles, when do token consumptions or productions occur.

To overcome this, we propose to make such events explicitly observable at the FSM level.[4] More specifically, with each put or get action of the dataflow network that we are interested in observing, we associate a corresponding output signal of the FSM. The intended meaning is that whenever that signal becomes 1, the corresponding production or consumption occurs.

[4] An alternative could be to attempt to *discover* consumptions and productions automatically by observing the behavior of the FSM. This problem is much more difficult, and is the topic of future work.

Let us formalize this. Let N be a closed dataflow network and let L be the set of actions of N to be observed. Let $M = (X, Y, S, s_0, \delta, \lambda)$ be a closed FSM. Because M is closed, $X = \emptyset$. Let $\theta : L \to Y$ be a 1-1 mapping from L to Y, associating to each action $\ell \in L$ a distinguished output signal $\theta(\ell) \in Y$ serving to observe action ℓ at the FSM level.

The mapping θ defines a mapping Θ from FSM observable behaviors to dataflow observable behaviors as follows. Let $\sigma = (a_0, b_0)(a_1, b_1)\cdots$ be an observable behavior of M. Because $X = \emptyset$, all a_k's are trivial (empty assignments). Then, each b_k is mapped to a subsequence $\rho_k = \mathsf{tick} \cdot \alpha_k$, where

$$\alpha_k := \{\ell \in L \mid b_k(\theta(\ell)) = 1\}.$$

That is, α_k is the set of all actions that are observed to occur at the FSM level, according to the distinguished outputs that are true in b_k. If α_k is empty then we let ρ_k be simply tick. Then, Θ maps the FSM observable behavior σ to the dataflow observable behavior $\Theta(\sigma) = \rho_0 \cdot \rho_1 \cdots$.

For example, let $L = \{\mathsf{put}, \mathsf{get}\}$ and $Y = \{y_{\mathsf{put}}, y_{\mathsf{get}}\}$. Let $\theta = \{\mathsf{put} \mapsto y_{\mathsf{put}}, \mathsf{get} \mapsto y_{\mathsf{get}}\}$. Then we have the following mappings from FSM observable behaviors to dataflow observable behaviors:

$$(y_{\mathsf{put}} = 0, y_{\mathsf{get}} = 0) \cdot (y_{\mathsf{put}} = 1, y_{\mathsf{get}} = 0) \cdot (y_{\mathsf{put}} = 0, y_{\mathsf{get}} = 1)$$

is mapped to

$$\mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}\} \cdot \mathsf{tick} \cdot \{\mathsf{get}\}$$

and

$$(y_{\mathsf{put}} = 0, y_{\mathsf{get}} = 0) \cdot (y_{\mathsf{put}} = 1, y_{\mathsf{get}} = 1) \cdot (y_{\mathsf{put}} = 0, y_{\mathsf{get}} = 0)$$

is mapped to

$$\mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdot \mathsf{tick}.$$

Having specified this mapping, we define two types of conformance as follows:

Definition 1 *M conforms to the non-idling (respectively, eager) semantics of N with respect to mapping θ iff for every observable behavior σ of M, the sequence $\Theta(\sigma)$ defined as above, is an observable behavior in the non-idling (respectively, eager) semantics of N.*

It is worth noting that if N is a dataflow model whose eager semantics is a subset of its non-idling semantics (e.g., as in a KPN), then, if M conforms to the eager semantics of N then it also conforms to the non-idling semantics of N.

Also note that since M is a closed FSM, it is by definition a Moore machine, and since we consider deterministic FSMs, M has a single behavior. We could therefore simplify the above definition to state "for the unique observable behavior σ of M" instead of "for every observable behavior σ of M". We prefer not to do so, however, in order to have a definition that generalizes to the case of non-deterministic FSMs.

We proceed to illustrate conformance by examples.

4.3 Examples of conformance and non-conformance

Consider the dataflow network N shown in Figure 7 and the FSM M shown in Figure 2. Let θ be the mapping

$$\theta = \{\mathsf{put} \mapsto \mathsf{v}_1, \mathsf{get} \mapsto \mathsf{v}_2\}.$$

That is, at the level of M, every time $\mathsf{v}_1 = 1$ this corresponds to a put in the buffer, and every time $\mathsf{v}_2 = 1$ this corresponds to a get.

We claim that M conforms to both the eager and non-idling semantics of N with respect to θ. As shown in Figure 2, M has a single infinite behavior yielding the infinite observable behavior

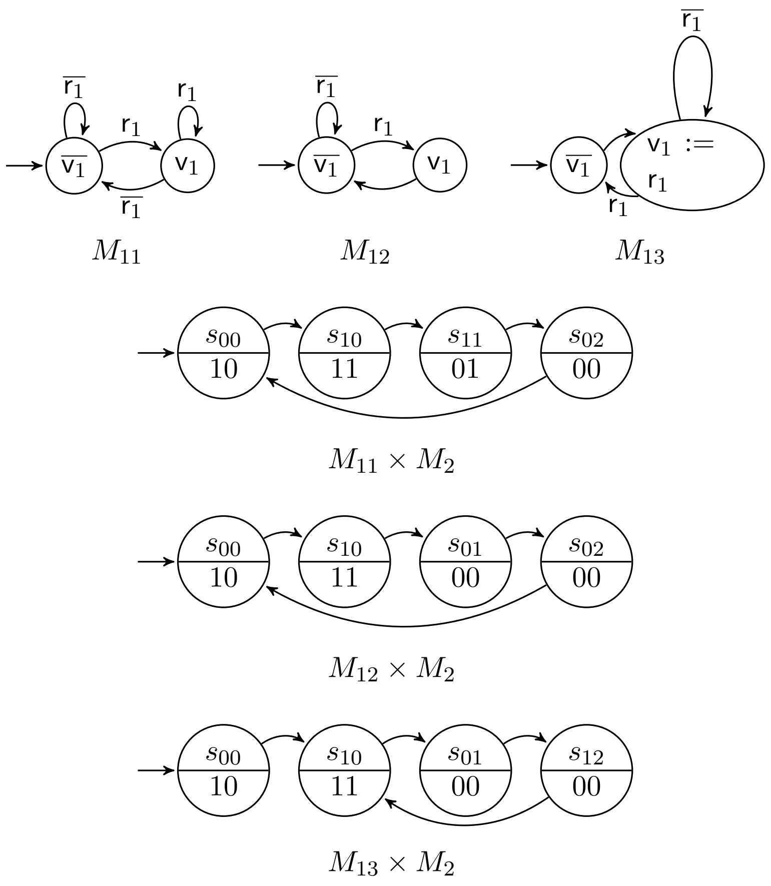

Figure 8: Top: three variants, M_{11}, M_{12}, M_{13}, of FSM M_1 of Figure 2. We compose each of these with M_2 (without use of M_{buf} in the middle). Let states of M_2 be labeled q_0, q_1, q_2. Let states of M_{1i} be labeled s_0, s_1. Resulting three composite FSMs are shown. In each of the composites, state s_{ij} is composed of s_i of M_{1k} and q_j of M_2, and vector in the lower half of each state denotes values of signals $\mathsf{r} = \mathsf{r}_1 = \mathsf{r}_2$ and $\mathsf{v} = \mathsf{v}_1 = \mathsf{v}_2$ respectively in that state.

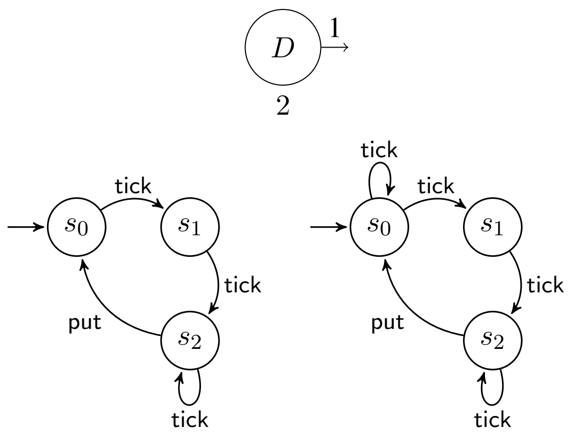

Figure 9: Two variants of SDF process D.

$$\begin{aligned}\sigma \quad = \quad & (\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2) \cdot (\mathsf{r}_1, \mathsf{v}_1, \overline{\mathsf{r}_2}, \overline{\mathsf{v}_2}) \cdot \\ & \left((\overline{\mathsf{r}_1}, \overline{\mathsf{v}_1}, \overline{\mathsf{r}_2}, \overline{\mathsf{v}_2}) \cdot (\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2) \cdot (\overline{\mathsf{r}_1}, \overline{\mathsf{v}_1}, \overline{\mathsf{r}_2}, \overline{\mathsf{v}_2})\right)^{\omega}\end{aligned}$$

where ρ^ω denotes the infinite repetition of a sequence ρ.

σ is mapped to the dataflow observable behavior

$$\Theta(\sigma) = \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdot \mathsf{tick} \cdot \{\mathsf{put}\} \cdot (\mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdot \mathsf{tick})^{\omega}.$$

It can be seen that $\Theta(\sigma)$ is identical to the observable behavior of the eager semantics of N – Figure 7, bottom. Therefore, M conforms to both the eager and non-idling semantics of N.

Consider next Figure 8. The figure shows three variants of FSM M_1 of Figure 2 and the synchronous FSM composition of each of these variants with FSM M_2 of Figure 2. Note that the buffer FSM M_{buf} is not used in these compositions. Let $\mathsf{r} = \mathsf{r}_1 = \mathsf{r}_2$ and $\mathsf{v} = \mathsf{v}_1 = \mathsf{v}_2$ be the names of the signals of the composite FSMs.

Define $\theta = \{\mathsf{put} \mapsto \mathsf{v}, \mathsf{get} \mapsto q_0 \wedge \mathsf{v}\}$. The expression $\mathsf{get} \mapsto q_0 \wedge \mathsf{v}$ means that we interpret v to correspond to a get action only when M_2 is at its initial state q_0, otherwise, even if $\mathsf{v} = 1$, we will not consider this a get. We use such expressions merely for reasons of convenience, without departing from the framework we set up above. Indeed, we could easily consider an additional signal v' defined to be 1 iff M_2 is at q_0 and $\mathsf{v} = 1$. Then, we could define θ equivalently as $\theta = \{\mathsf{put} \mapsto \mathsf{v}, \mathsf{get} \mapsto \mathsf{v}'\}$. Therefore, using such expressions is not more expressive than our original framework.

With the above mapping θ, the observable behaviors of the three composite FSMs are mapped to the following observable dataflow behaviors:

1. $(\mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdot \mathsf{tick} \cdot \{\mathsf{put}\} \cdot \mathsf{tick})^{\omega}$,
2. $(\mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdot \mathsf{tick} \cdot \mathsf{tick})^{\omega}$,
3. $(\mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\} \cdot \mathsf{tick})^{\omega}$.

None of these composites conforms to dataflow network N of Figure 7, because N does not admit the starting sequence $\mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}, \mathsf{get}\}$. This non-conformance indicates that SDF process B of Figure 4 may incorrectly capture HW blocks M_{1k}. Indeed, B can produce a token every 1 time unit, whereas it appears that, M_{1k} require 2 time units.

Instead of B, consider SDF process D of Figure 9 and dataflow network N_{DC} shown at the top of Figure 10. N_{DC} is similar to the network of Figure 7 except that B is replaced by D. N_{DC} defines two composite dataflow processes, one for each of the two variants of D: the two composite processes are denoted N_1 and N_2 and are shown in Figure 10. Then:

1. $M_{11} \times M_2$ conforms to neither N_1 nor N_2. On inspecting the behavior of $M_{11} \times M_2$, it is evident that every other token generated by M_{11} is dropped, i.e., it is not read by M_2 because M_2 is busy processing the previous token. This is a case of wrong synchronization between the two FSMs, which is revealed by attempting to show conformance to an SDF model.

2. $M_{12} \times M_2$ does not conform to N_1, but conforms to the non-idling semantics of N_2. In this case, one may interpret $M_{12} \times M_2$ as a non-idling implementation of N_{DC} where the execution of D and C is pipelined in such a way as to overlap the last cycle of C with the first one of the next D, achieving a non-optimal throughput of $\frac{1}{4}$. Such a pipelining can be captured by N_2 but not by N_1. This indicates that N_1 is not a faithful model of this HW. Also, although $M_{12} \times M_2$ conforms to the non-idling semantics of N_2, it does not conform to its eager semantics, and indeed, does not achieve the optimal throughput of $\frac{1}{3}$.

3. $M_{13} \times M_2$ conforms to the non-idling semantics of N_1 and therefore also of N_2 since N_1 is a subset of N_2. $M_{13} \times M_2$ achieves optimal throughput $\frac{1}{3}$. Despite this, its behavior is not eager, and therefore it does not conform to the eager semantics of N_1 or N_2.

4.4 Discussion

As seen from the examples presented above, conformance can be used in a number of different scenarios. It can provide guarantees of throughput preservation between dataflow models and HW implementations. It can point to timing or synchronization errors in HW implementations, or to inadequacies of the dataflow model of the HW. Thus, our framework can be used in a *bottom-up* methodology where HW is given and the goal is to build faithful performance models of this HW, as well as in a *top-down* or *model-based design* methodology where the goal is to synthesize from a high-level model (e.g., SDF) a HW implementation that preserves the properties of the model.

The definition of conformance as containment of behaviors allows to derive such preservation for properties of type “for-all”. More precisely, if a property P is stated as “for all behaviors of N something holds” then if N satisfies P, any model whose behaviors are a subset of N also satisfies P.

Conformance can be used in particular to show preservation of performance bounds such as worst-case or best-case throughput and latency. For example, bounds on throughput can be expressed using “for-all” properties of the form “for any behavior ρ, the throughput of ρ is in $[T_{\min}, T_{\max}]$”.

Our conformance relation is essentially a *language inclusion* type of conformance, modulo the fact that a translation Θ from FSM behaviors to dataflow process behaviors needs to be performed first. Such a translation can be performed automatically by appropriately transforming an FSM into another type of finite automaton. If the process automaton is also finite-state, then conformance can be checked automatically, using standard model-checking type of techniques [3].

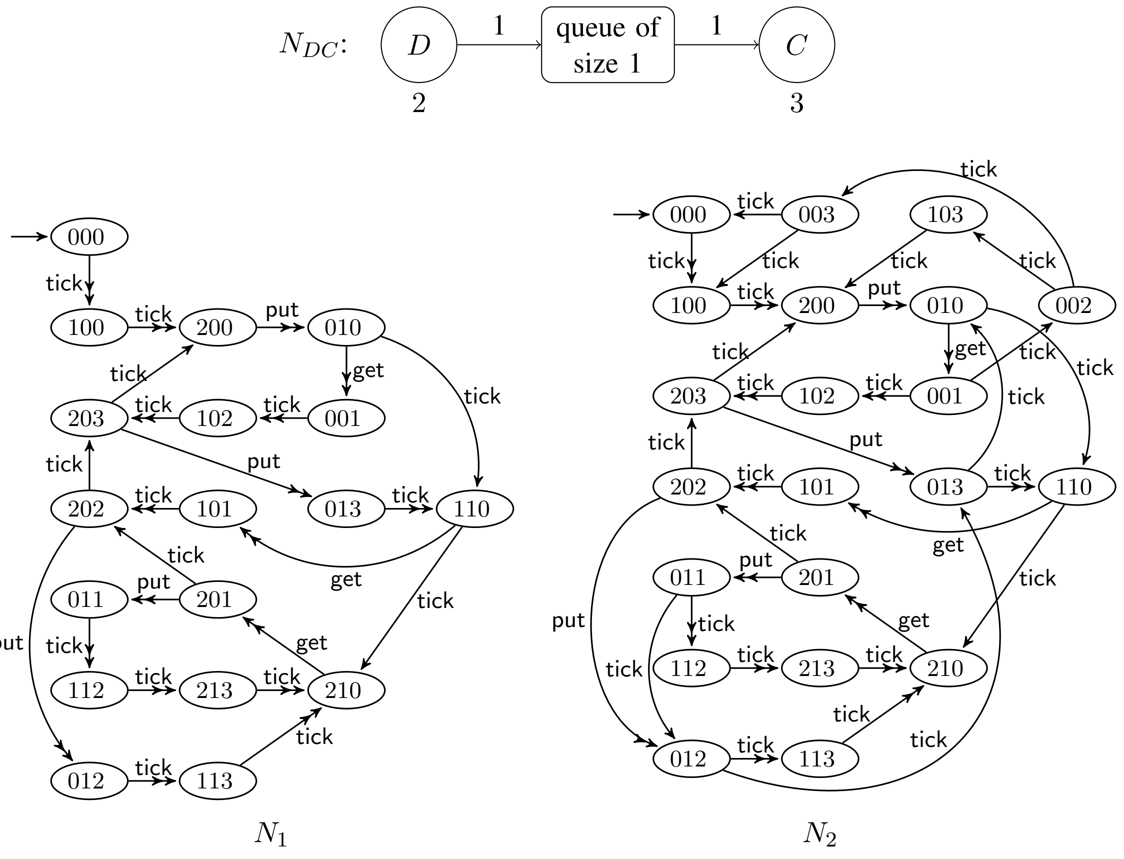

Figure 10: Top: closed dataflow network of actors D, C connected using queue of size 1. Bottom-left: composite non-idling dataflow process, N_1, using left-most variant of process D from Figure 9. Bottom-right: composite non-idling dataflow process, N_2, using right-most variant of D. In each of the composites, the corresponding eager composition is embedded, as shown by edges with double arrowheads.

5 Conclusions and Future Work

We have investigated the question of faithfulness of dataflow models to hardware implementations by proposing a formal conformance relation between the two. The examples of dataflow processes presented above are SDF, but our process model is general enough to capture other dataflow variants as well. Since conformance is defined with respect to the process model, this means that the framework is applicable to a wide class of dataflow models.

Our current study is limited to closed systems. One of our future goals is to study conformance between open systems, with the main challenge being to guarantee some notion of *compositionality*. For instance, we would like our framework to guarantee that if M_1 conforms to N_1 and M_2 conforms to N_2, then $M_1 \times M_2$ conforms to $N_1||N_2$ (where $||$ denotes dataflow composition). This is essential for scalable conformance checking, but also for incremental design, where a HW component can replace another one without compromising the properties of the overall system.

Another direction of future work is to develop "recipes" for generating dataflow processes such as the ones used in the examples above for a variety of dataflow models (SDF, CSDF, HDF, ...). Developing specialized algorithms for checking conformance with respect to these subclasses is an additional interesting objective.

An alternative way to bridge the gap between dataflow and hardware is to give them both semantics in terms of the denotational actor model of [8]. This has already partly been done in [8] for SDF but not for general dataflow. It has also been done in [8] for different models of discrete automata, but not for the Mealy and Moore machines which are the standard hardware models. Once both dataflow and hardware are given actor semantics, they "live in the same world" and can therefore be compared using the refinement relation defined in [8], or another relation such as the one based on subsets of behaviors that we employ here.

References

[1] S. Bhattacharyya, E. A. Lee, and P. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *IEEE Intl. Conf. Acoustics, Speech, and Signal Processing*, volume 5, pages 3255–3258, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *DAC '03*. ACM, 2003.

[5] J. Dubois, R. Thavot, R. Mosqueron, J. Miteran, and C. Lucarz. Motion estimation accelerator with user search strategy in an RVC context. In *Proceedings of the 16th IEEE international conference on Image processing*, ICIP'09, pages 761–764, Piscataway, NJ, USA, 2009. IEEE Press.

[6] A. Faustini. An operational semantics for pure dataflow. In M. Nielsen and E. Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *LNCS*, pages 212–224. Springer, 1982.

[7] M. Geilen and T. Basten. Kahn Process Networks and a Reactive Extension. In S.S. Bhattacharyya et al., editor, *Handbook of Signal Processing Systems*. Springer, 2010.

[8] M. Geilen, S. Tripakis, and M. Wiggers. The earlier the better: A theory of timed actor interfaces. In *14th Intl. Conf. Hybrid Systems: Computation and Control (HSCC'11)*. ACM, 2011.

[9] F. Ghenassia, editor. *Transaction-Level Modeling with SystemC*. Springer, 2005.

[10] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *Computers Digital Techniques, IET*, 3(5):398 –412, September 2009.

[11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[12] J. Horstmannshoff and H. Meyr. Optimized system synthesis of complex rt level building blocks from multirate dataflow graphs. In *12th International Symposium on System Synthesis*. IEEE, 1999.

[13] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 287 –292, oct. 2008.

[14] B. Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distrib. Comput.*, 7(4):197–212, 1994.

[15] H. Jung, H. Yang, and S. Ha. Optimized RTL Code Generation from Coarse-Grain Dataflow Specification for Fast HW/SW Cosynthesis. *J. Signal Process. Syst.*, 52(1):13–34, July 2008.

[16] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.

[17] Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.

[18] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[19] R. Lauwereins, M. Engels, M. Adé, and J. A. Peperstraete. Grape-II: A System-Level Prototyping Environment for DSP Applications. *Computer*, 28(2):35–43, 1995.

[20] E. A. Lee and D. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.

[21] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[22] O. M. Moreira and M. J. G. Bekooij. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing*, 2007(83710):1–15, April 2007.

[23] T. Olsson, A. Carlsson, L. Wilhelmsson, J. Eker, C. von Platen, and I. Diaz. A reconfigurable OFDM inner receiver implemented in the CAL dataflow language. In *Circuits and Systems (ISCAS)*, 2010.

[24] C. Pixley. Practical Considerations Concerning HL-to -RT Equivalence Checking. In *Hardware and Software: Verification and Testing*, volume 5394 of *LNCS*. Springer, 2009.

[25] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2 edition, 2009.

[26] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *Computers, IEEE Transactions on*, 57(10):1331 –1345, October 2008.

[27] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *SAMOS XI*, pages 404–411, July 2011.

[28] R. Thavot, R. Mosqueron, M. Alisafaee, C. Lucarz, M. Mattavelli, J. Dubois, and V. Noel. Dataflow design of a co-processor architecture for image processing. In *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2008.

[29] S. Tripakis, H. Andrade, A. Ghosal, R. Limaye, K. Ravindran, G. Wang, G. Yang, J. Kornerup, and I. Wong. Correct and non-defensive glue design using abstract models. In *CODES+ISSS11*. ACM, 2011.

[30] R. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4-5):229–327, 2000.

[31] R. J. van Glabbeek. The linear time-branching time spectrum. In *CONCUR'90*, pages 278–297. Springer, 1990.

[32] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 658–663, New York, NY, USA, 2007. ACM.

[33] M.C. Williamson and E.A. Lee. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *Signals, Systems and Computers, 13th Asilomar Conference on*. IEEE, 1996.