J Sign Process Syst
DOI 10.1007/s11265-015-0971-y

Tokens vs. Signals: On Conformance between Formal Models of Dataflow and Hardware

Stavros Tripakis · Rhishikesh Limaye · Kaushik Ravindran · Guoqiang Wang · Hugo Andrade · Arkadeb Ghosal

Received: 10 October 2014 / Revised: 31 December 2014 / Accepted: 12 January 2015
© Springer Science+Business Media New York 2015

Abstract Designing hardware often involves several types of modeling and analysis, e.g., in order to check system correctness, to derive performance properties such as throughput, to optimize resource usages (e.g., buffer sizes), and to synthesize parts of a circuit (e.g., control logic). Working directly with low-level hardware models such as finite-state machines (FSMs) to answer such questions is often infeasible, e.g., due to state explosion. Instead, designers often use dataflow models such as SDF and CSDF, which are more abstract than FSMs, and less expensive to use since they come with more efficient analysis algorithms. However, dataflow models are only abstractions of the real hardware, and often omit critical information. This raises the question, when can one say that a certain dataflow model faithfully captures a given piece of hardware? The question is of more than simply academic interest. Indeed, as illustrated in this paper, dataflow-based analysis outcomes may sometimes be defensive (e.g., buffers that are too big) or even incorrect (e.g., buffers that are too small). To answer the question of faithfully capturing hardware using dataflow models, we develop a formal conformance relation between the heterogeneous formalisms of (1) finite-state machines with synchronous semantics, typically used to model synchronous hardware, and (2) asynchronous processes communicating via queues, used as a formal model for dataflow. The conformance relation preserves performance properties such as worst-case throughput and latency.

Keywords Finite state machines · Dataflow · Conformance · Hardware design · Hardware synthesis · Verification · Formal methods

S. Tripakis (✉)
University of California, Berkeley, CA, USA
e-mail: stavros.tripakis@gmail.com

S. Tripakis
Aalto University, Espoo, Finland

R. Limaye · K. Ravindran · G. Wang · H. Andrade · A. Ghosal
National Instruments Corporation, Berkeley, CA, USA

R. Limaye
e-mail: rhishikesh.limaye@ni.com

K. Ravindran
e-mail: kaushik.ravindran@ni.com

G. Wang
e-mail: guoqiang.wang@ni.com

H. Andrade
e-mail: hugo.andrade@ni.com

A. Ghosal
e-mail: arkadeb.ghosal@ni.com

1 Introduction

The hardware design process today is largely *model-based* in the sense that designers work with high-level models which capture the essential properties of the system under design, while hiding irrelevant details. Automatic synthesis tools are then used which take as input a high-level model and generate a lower-level model, filling in the implementation details. This process often involves multiple layers of abstraction.

A standard model for hardware (HW), and in particular synchronous HW, which is the main focus of this paper, is the model of synchronous finite-state machines (FSMs). FSMs are a natural model, as they are semantically very close to synchronous HW. Ignoring concerns such as critical path delay, technology mapping, placement, routing, noise cancellation, and other problems which are taken care

Published online: 30 January 2015

of during lower levels of abstraction, a piece of HW logically behaves as an FSM, where a transition of the FSM corresponds to a tick of the HW clock.

Although FSMs model HW at a higher level of abstraction than, say, models based on differential equations (e.g., SPICE), FSMs are still cycle-accurate and detailed models. This can often be problematic when FSMs are used for analysis, because of the well-known *state explosion problem*, which means that the FSMs become too large to handle.

For this reason, HW designers often use higher-level models coming from the family of *dataflow* models, such as SDF [3], CSDF [4], and SADF [5]. These models admit efficient analysis methods for computing key performance metrics such as throughput, latency, or buffer sizes. Dataflow models suffer much less from state explosion because they abstract much of the information contained in the FSM descriptions (e.g., Verilog or VHDL). For example, models such as SDF typically omit data values and use only abstract notions of *tokens* (as done in, say, Petri nets [6]).

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* to the original HW. (2) is not simply an academic concern. As we illustrate in this paper, using the example of the *glue design problem*, dataflow models are often used incorrectly (meaning that the dataflow model does not conservatively approximate the HW), or too defensively (meaning that the dataflow model is too conservative). A prerequisite for answering questions (1) and (2) is to make the notion of faithfulness precise, and this is one of the main questions that this paper tries to answer. The paper combines and extends previous work reported in [1, 2].

When attempting to define faithfulness, we are faced with a major difficulty. The dataflow model is semantically very different from FSMs. FSMs communicate synchronously by means of input/output (boolean) *signals*. In dataflow, a set of concurrent processes communicate *asynchronously* by producing and consuming tokens from/to a set of (usually FIFO) queues. It appears that the two models "live in different worlds" and that comparing them is like comparing apples and oranges.

In this paper, we study this comparison problem. Our goal is to bridge the semantic gap between dataflow and HW implementations. We do this by defining a formal *conformance* relation between FSMs and a formal operational model of dataflow. The latter has a notion of time that we map to HW clock ticks.[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.

The rest of the paper is organized as follows. We briefly review FSMs and their composition in Section 2. In Section 3 we describe the glue design problem, which is a main motivation for this work. In Section 4 we illustrate the problems that can arise when using abstract dataflow models to solve the glue design problem. An operational process model for dataflow is proposed in Section 5. We present a conformance relation between FSMs and dataflow networks in Section 6. Related work is discussed in Section 7. Conclusions and plans for future work are presented in Section 8.

2 A Model for Hardware

We model hardware as *finite-state machines* (FSMs) [9] of type *Mealy* (and *Moore* as a special case).[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.
- $\delta : S \times 2^X \to S$ 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

[1] Our formal dataflow model is similar to standard timed dataflow models such as timed SDF [7]. Original works on dataflow models such as SDF consider their untimed versions, e.g., [3, 8]. Timed properties such as throughput cannot be evaluated on untimed models. For this reason, we work with timed dataflow models.

[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.

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 Fig. 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 $\{\mathsf{v}_1, \mathsf{r}_2\}$. 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" to the output: v_2 is set to 1 and the buffer continues to be empty. In the latter case, v_2 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.

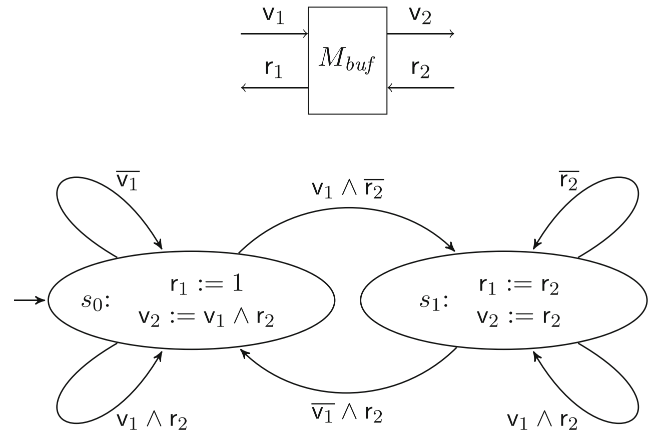


Figure 1 Example FSM: structure (*top*) and behavior (*bottom*). This FSM is of type Mealy, since, say, output v_2 is a function of input v_1 in state s_0. Note that we use a slightly different notation for Mealy machines than traditionally used, and represent the output function on the states rather than on the transitions of the machine. The transitions are annotated only with guards on the inputs, and represent only the transition function. We find this notation more appropriate, since the inputs may change multiple times during a clock cycle, while the state remains fixed, and only changes at clock ticks. Then, the outputs may also change multiple times at a given state, without the machine making a transition.

2.3 FSM Composition

FSMs can be composed with other FSMs, to form larger FSMs. Since we use FSMs to model synchronous hardware, we consider synchronous composition, where all composed FSMs take a transition simultaneously. Different types of composition 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, we need to define *combinational (i.e., instantaneous) dependencies* between the output and input signals of a machine. Recall that, for an output signal $y \in Y$, the function $\lambda_y : S \times 2^X \to \{0, 1\}$ returns the value of y for given state and input. We say that y *combinationally depends* on a given input signal $x \in X$ if the Boolean function λ_y changes value when x is toggled between 0, 1, while all other input signals

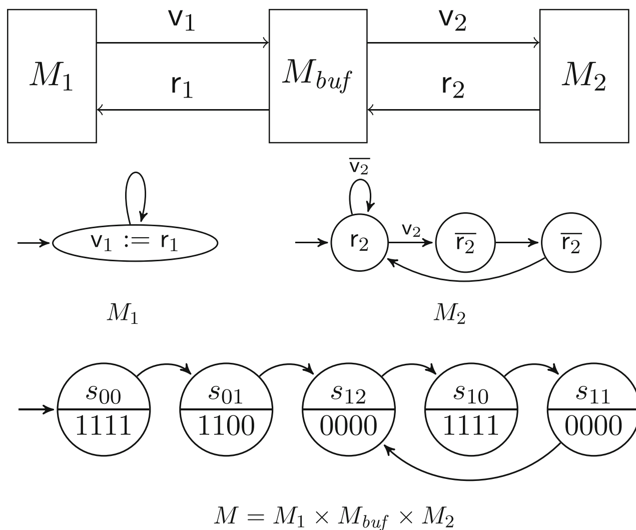


Figure 2 Closed FSM M obtained by composing FSMs M_1, M_2 above, with M_{buf} from Fig. 1. The vectors in the lower half of each state denote the values of the four output signals $\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2$ in that state.

remain constant. For example, in the FSM M_{buf} from Fig. 1, output v_2 combinationally depends on both inputs v_1 and r_2, because of the assignment $\mathsf{v}_2 := \mathsf{v}_1 \wedge \mathsf{r}_2$ at state s_0. On the other hand, output r_1 combinationally depends on input r_2, but not on input v_1.

A composition of two or more FSMs is valid if the combinational dependencies of the FSMs do not form a cycle after we connect the signals. This is checked easily by constructing a dependency graph from the network of FSMs, and checking whether this directed graph has cycles. The nodes of the graph are signals, and the edges are combinational dependencies. For example, consider the composition of the three FSMs shown in Fig. 2. M_{buf} is the FSM from Fig. 1, while the behaviors of M_1 and M_2 are shown in Fig. 2. This is a valid composition, because the combinational dependency graph, shown in Fig. 3, has no cycles. This is because, first, the output r_2 of M_2 does not depend on its input v_2 (M_2 is a Moore machine), and second, the output r_1 of M_{buf} does not depend on its input v_1. Note that none of the outputs of M_{buf} is a Moore output.

Provided the combinational dependencies are acyclic, the composition of a network of FSMs forms a new, *composite* FSM. We will not define this composite FSM formally, as it is standard. Instead, we give an example. Consider again

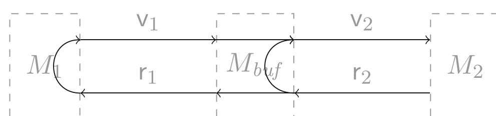


Figure 3 Combinational dependency graph of the composite FSM $M = M_1 \times M_{buf} \times M_2$ from Fig. 2. The absence of cycles in the graph makes the composition valid.

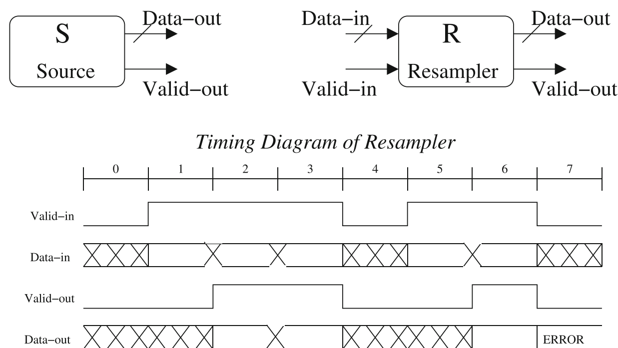
the composition of the three FSMs shown in Fig. 2. The composite FSM M is shown at the bottom of the figure. M is the synchronous composition of M_1, M_{buf} and M_2, and is denoted also as a product $M_1 \times M_{buf} \times M_2$. M has no input signals: all its four signals $\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2$ 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 $\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2$ 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).

3 The Glue Design Problem

We motivate the need for a formal definition of conformance between hardware and dataflow models with a simple yet realistic use case. Figure 4 (left) shows the interfaces for two hardware IP blocks: a Signal Source (S) and a Rational Resampler (R). The Source block generates one data sample every two clock cycles. The sample value is produced on the *Data-out* output, and the *Valid-out* signal is asserted to indicate the presence of a sample on *Data-out*. The Resampler block does 2/3 resampling, that is, for every three input samples, it produces two output samples. A simple implementation of Resampler in VHDL is shown in Fig. 4 (right).

Alternatively, Resampler could be implemented using the FIR IP block in the Xilinx CoreGen library [10]. Both implementations have the following semantics on their interfaces: *Data-in* carries input data samples into the Resampler. The corresponding *Valid-in* input indicates when the sample on *Data-in* is valid. Three data samples must be provided in three consecutive cycles, i.e. *Valid-in*, once asserted, must stay high for three cycles. Output sample values are produced on *Data-out* in second and third cycles, with *Valid-out* asserted to indicate their validity.

The above interface semantics can be derived by analyzing the VHDL implementation, and/or documentation and timing diagrams in data sheets. The timing diagram of Fig. 4 shows examples of both correct and incorrect usages of the Resampler block. A correct usage is illustrated in the first 4 clock cycles; an incorrect one in cycles 5-7. In cycle 5, the *Valid-in* input for the Resampler turns *true*, but it becomes *false* in cycle 7. This voids the input requirement that three data samples must be provided in three consecutive cycles. This results in the timing diagram indicating "ERROR" in cycle 7.




```
entity Resampler is
  port (clk: in std_logic;
    di: in signed(7 downto 0);    -- Data-in
    vi: in std_logic;     -- Valid-in
    do: out signed(16 downto 0); -- Data-out
    vo: out std_logic     -- Valid-out
  );
end Resampler;

architecture arch of Resampler is
  type state_type is (s0,s1,s2);
  signal state: state_type := s0;
  signal next_state: state_type;
  signal dp : signed(7 downto 0);
begin
  next_state <=
    s0 when state = s0 and vi /= '1' else
    s1 when state = s0 and vi = '1' else
    s2 when state = s1 and vi = '1' else
    s0;

  do <= c0*di + c2*dp when state = s1 else
        c1*di + c3*dp when state = s2 else
        to_signed(0, 17);

  vo <= vi when state = s1 else
        vi when state = s2 else '0';

  process (clk)
  begin
    if rising_edge(clk) then
      state <= next_state;
      dp <= di;
    end if;
  end process;
end arch;
```

Figure 4 *Left*: interfaces of Source and Resampler IP blocks and timing diagram of Resampler. *Right*: an implementation of Resampler in VHDL; `c0`, `c1`, `c2`, `c3` are 8-bit constants that are declared elsewhere.

The challenge to the hardware designer is to create a valid composition of these blocks (i.e., a *system*) so that behavioral and timing constraints are respected. These constraints include correctness requirements such as correct usage of blocks like Resampler. But there may also be additional requirements related to performance. For instance, requirements imposed on the output sample rate, analogous to a throughput constraint.

An obvious composition of the blocks of Fig. 4 is to connect *Data-out* and *Valid-out* from the Source directly to the *Data-in* and *Valid-in* on the Resampler, respectively. However, this results in an invalid composition, since the Source produces an output sample every other cycle, whereas the Resampler requires 3 samples in consecutive cycles. Hence, some *glue* consisting of buffer and control logic is necessary to connect these blocks. Alternatively, a different Source block that produces a sample every cycle satisfies the timing requirements on the Resampler inputs. In this case, a direct connection between these blocks results in a valid configuration. The overarching challenge is to reason about these compositions and design the appropriate glue to coordinate the interaction between components.

In the rest of this section, we discuss in more detail the components of a system, namely, actors and glue, and define the glue design problem in a more precise manner.

3.1 Actors

We use the term *actors* to refer to system components such as IP blocks, legacy blocks, or other blocks that perform computations. These are typically available in hardware description language (HDL) such as VHDL, Verilog, or as low-level netlists.

Designing glue directly at the HDL or netlist level is often infeasible in practice, due to state explosion and other combinatorial explosion problems. As a first

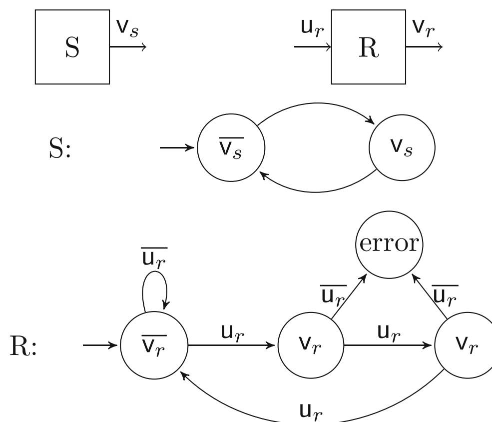


Figure 5 FSM models for the Source (S) and Resampler (R) actors.

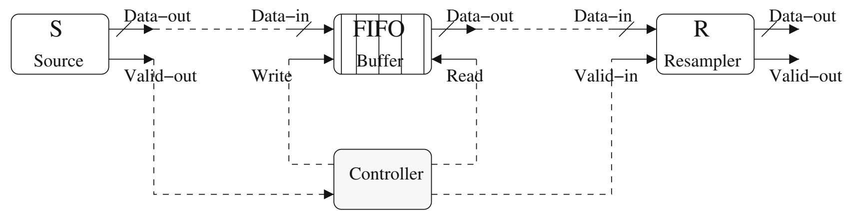


Figure 6 System built by connecting the Source and Resampler blocks of Fig. 4 with glue that includes a FIFO buffer and a controller.

remedy to this problem, we can model actors using simplified FSMs where we abstract away the data signals and preserve only the control signals. This allows us to obtain simpler and smaller FSMs. Another key characteristic is that the error conditions are modeled explicitly.

Figure 5 shows the FSMs for the Source and Resampler actors. We refer to the FSMs by the same name as the actors they represent but shorten and rename the control signals *Valid-out* of *S*, *Valid-in* of *R*, and *Valid-out* of *R* to v_s, u_r,and v_r, respectively. The FSM *S* makes a sample (also called *token*) available every second clock cycle, by setting its output v_s to true. The FSM *R* waits for its input signal u_r to become true. Once u_r becomes true, *R* requires that it stays true for 3 consecutive clock cycles (these include the first cycle where u_r became true). If this requirement is violated, *R* moves into an "error" state. Otherwise, *R* produces two samples in the last two of the three consumption clock cycles, by setting its output signal v_r to true.

Note that the machine for the Resampler actor given in Fig. 5 does not strictly conform to the definition of FSM given in Section 2. This is because of the error state. Error states can be seen as states where the machine enters a mode where its behavior is *undefined*, or *chaotic* ("anything can happen"). This models a non-deterministic FSM, which can produce any output once it enters the error state. In Section 6.4, we shall see how this non-determinism can be used to catch non-conformance and thus indicate cases of erroneous dataflow models.

3.2 Glue

The problem a designer faces is to connect actors such as *S* and *R* to form a *system*. In this example, *S* and *R* cannot be connected directly, i.e., by connecting output v_s to input u_r, because this would violate the requirements of *R*, as mentioned above. The system must therefore include some glue which in the context of this work consists of a set of intermediate *buffers* and corresponding *control logic*. An example of a system formed by composing *S* and *R* via some glue is shown in Fig. 6. In the sequel, we elaborate on the salient components of glues.

3.2.1 Buffers

The glue often includes buffers that store data tokens produced by components until these tokens can be consumed by other components. In case such a buffer is finite, its behavior can be modeled using FSMs, as with actors.

Figure 7 shows the FSM for a buffer *Buf2* that can hold at most two tokens. *Buf2* has two input control signals w and r, representing a write request and a read request respectively. As before, data signals are abstracted. Transitions labeled "else" denote the default behavior. Note that this buffer requires that it not be read from when it is empty, or written to when it is full. This buffer allows simultaneous reads and writes, except when it is empty.

3.2.2 Control Logic

The glue may also include some type of control logic to control the execution of actors. Figure 8 shows an example. Controller Ctrl has the interface of Fig. 6: it has a single input signal v_s and three output signals, $\mathsf{w}, \mathsf{r}, \mathsf{u}_r$. Output w is set to v_s at all states, meaning that whenever the Source produces a token, the controller writes this into the buffer. At the 5th cycle, when two tokens have already been stored into the buffer, the controller starts issuing read requests, at the same time enabling the input signal of the Resampler. Provided a buffer of size at least 2 is used, such as Buf2 of Fig. 7, this controller guarantees that the requirements of the Resampler are satisfied.

3.3 System-Level Properties

Once we have a set of actors and corresponding glue, we can compose them together to form a system. In our setting,

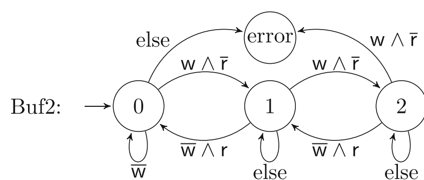


Figure 7 FSM for a buffer of size 2.

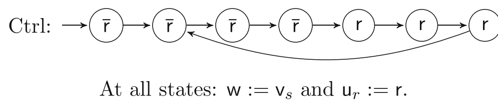


Figure 8 A mealy machine modeling the control logic.

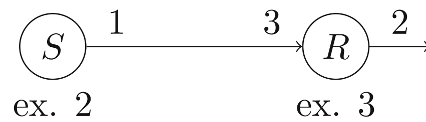


Figure 9 SDF model of the source-resampler example.

a system is a *closed* FSM, that is, an FSM without input signals. For example, Fig 6 represents a system.

The objective of the designer is to build a system that has certain properties. These include *correctness* properties such as *compatibility* of actors, absence of *deadlocks*, absence of *buffer overflow*, etc. These correctness properties can often be described as some type of *safety* properties on FSMs such as "an error state is never reached". For example, the system built by directly composing actors S and R of Fig. 5 is incorrect, because the error state of R is reachable; the system built by composing S, R, Buf2, and Ctrl, of Figs. 5, 7 and 8, is correct because no error state is reachable. Note that if in this system Buf2 is replaced by a buffer of size 1, then the system would no longer be correct, as the buffer would overflow.

In addition to correctness, the system must satisfy some *performance* properties. These often include lower bounds on throughput (the average number of tokens produced per cycle) as well as optimality requirements with respect to metrics such as sizes of buffers or control logic. For instance, in our running example, the throughput achieved by the system (S,R,Buf2,Ctrl), as measured at the output v_r of the Resampler, is $\frac{1}{3}$, as on average 2 tokens are produced by R every 6 cycles. It can be checked that increasing the buffer size further will not improve the throughput, in other words, a buffer of size 2 is *optimal* to achieve this throughput.

3.4 Glue Design Problem

Definition 1 (**Glue design problem**) Given a set of actors, synthesize a glue, consisting of buffers and control logic, such that the closed system resulting from composing the actors with the glue satisfies a set of given correctness, performance, and optimality properties.

The glue design problem is challenging for a number of reasons. At the theoretical level, one could formalize the problem as a *controller synthesis* problem, along the lines of works [11, 12] and their successors. However, there are challenges in doing so. First, a glue generally includes multiple buffers and control logic, which may itself be distributed. Thus, if we look at the glue as the controller to be synthesized, this controller, being a collection of components, is *decentralized*. Furthermore, some glue components (e.g., buffers) may be parameterized (e.g., buffer size), or they may be chosen from a library of available components. Also, the glue in general only has partial information about the actors. For example, in Fig. 6, the glue can only observe the data and control outputs of the Source. These characteristics lead to controller synthesis problems which are hard, and generally undecidable [13–16]. Finally, the requirements on the closed system are complex, and expressing these requirements formally is not an easy task. For example, part of the correctness requirements is that every token produced by an actor is eventually delivered to another actor, which strictly speaking is not a regular (finite-memory) property.

In addition to the theoretical challenges, there are practical challenges, as automatic controller synthesis methods are often plagued by state explosion, implementability, and other problems.

Because of the above challenges, conventional practice often follows a trial-and-error process, where some glue is chosen, the system is simulated for a finite number of inputs and cycles, and depending on the results, the glue is modified and the process repeated. This process is not guaranteed to converge to satisfactory results. An alternative is to employ higher-level, dataflow models, such as SDF and CSDF. However, care must be taken when using such models, as the following section illustrates.

4 Using Abstract Models to Solve the Glue Design Problem

The glue design problem is difficult to solve directly at the HDL level or even at the FSM level due to the theoretical and practical challenges described in the previous section. This is where abstract models such as SDF come into play. These models admit efficient algorithms for buffer sizing, throughput, and other tasks, which can be applied for automatic glue synthesis. However, care must be taken so that employing such abstractions yields valid (i.e., correct) and non-defensive (i.e., not too conservative) results, as we show in this section.

4.1 SDF Model

Many streaming applications can be specified as SDF models [3]. An SDF model consists of a finite set of

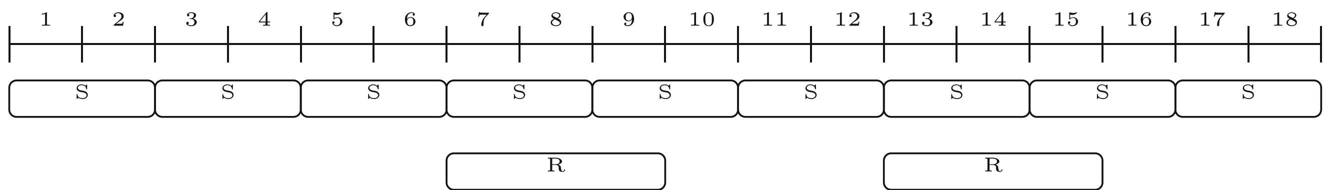


Figure 10 Firing schedule to achieve optimal throughput for the SDF model of Fig. 9 assuming buffer size 5.

computational actors inter-connected by directed links representing unbounded First In First Out (FIFO) channels that carry streams of data tokens. The SDF semantics requires the number of tokens consumed and produced by an actor per firing to be fixed and pre-specified. We restrict ourselves to SDF models without *auto-concurrency*, so that at most one firing of each actor can be active at a given time. Forbidding auto-concurrency can be done by explicitly adding self-loops to the model, but we avoid this for reasons of simplicity, and instead assume no auto-concurrency implicitly.

Figure 9 shows a possible SDF model for the Source-Resampler example from Fig. 5. This SDF model contains two actors, *S* and *R*: *S* produces 1 token every time it fires and *R* consumes 3 tokens and produces 2 tokens every time it fires. These static *token rates* annotate the links of the model.

The dataflow models that we consider are *timed*. The *execution time* (ET) of each actor (marked as "ex.") is the time it takes to complete one firing (measured in HW clock cycles). The ET includes the total time required to consume tokens from all input channels, perform computation, and produce tokens on all output channels. It could be exact or an upper bound on the worst case behavior. Note that the SDF abstraction hides the cycle-level details of exactly when consumption, computation and production happens within the span of an ET interval.

The ET and the token rates are typically derived from low-level behavior and timing diagrams, such as the ones specified in Fig. 4, or from FSM models like the ones in Fig. 5. How the abstract models (SDF or others) are built or extracted automatically from more concrete models is an interesting problem, but beyond the scope of this paper. We leave this discussion for future work.

The value of the SDF abstraction is that it enables static analysis of key execution properties. Absence of deadlocks (i.e., proper channel initialization) and consistency of execution rates (i.e., ability to execute the model with bounded channels) can be checked using efficient polynomial time algorithms [3, 17]. The result of the analysis also determines the relative execution rates of the actors in one iteration of the model. For example, the SDF model in Fig. 9 requires 1 firing of *R* for every 3 firings of *S* to guarantee that the channel remains bounded during execution.

Furthermore, the SDF model can be used to compute a static schedule of actor firings and the corresponding throughput of the model. One common scheduling strategy, guaranteed to achieve optimal throughput for a given selection of buffer sizes, is a *self-timed schedule*, where finite buffers are modeled using the standard technique of backward edges [18, 19]. As the SDF abstraction does not reveal the exact timing of consumption or production of tokens, the following conservative assumptions are made when deriving the self-timed schedule: 1) an actor starts firing exactly when enough tokens are available at all input channels (this, together with the backward edges, ensures also that the requisite number of vacancies are at that time available at all outputs); 2) output tokens are produced only at the last clock cycle in a firing (therefore delaying the firing of downstream actors as much as possible); 3) input tokens are consumed only at the last clock cycle in a firing (therefore delaying the firing of upstream actors as much as possible).

An optimization problem for SDF is to compute buffer sizes for the channels in order to achieve a specified throughput. Several exact and heuristic algorithms have been studied for this problem [17–20]. Figure 10 shows the self-timed schedule when the channel between *S* and *R* is implemented by a FIFO buffer of size 5. In this case, *R* fires every time 3 tokens are available on the buffer. The additional space in the buffer ensures that there are sufficient vacancies to start the subsequent firings of *S* while *R* works on 3 tokens present in the buffer. The throughput at the output of *R* is 2 samples every 6 cycles, which corresponds to the optimal throughput of the system.

The benefits of the SDF abstraction are not limited to static analysis of key properties. The abstraction also enables automatic synthesis of the glue to connect the underlying hardware IP blocks and generate a fully functional implementation. For example, the buffer and schedule shown in Fig. 10 can be naturally incorporated as the buffer and controller components of Fig. 6, respectively. Similarly, automatic synthesis of the glue (buffers and schedule) is possible not only from SDF, but also from other abstract models such as CSDF, discussed next.

4.2 CSDF Model

The CSDF model generalizes SDF by allowing the number of tokens consumed or produced by an actor to vary according to a fixed cyclic pattern [4]. Each firing of a CSDF actor corresponds to a *phase* of the cyclic pattern.

Figure 11 shows a possible CSDF model for the Source-Resampler example of Fig. 5. This CSDF model contains two actors, S and R. S cycles between two phases, each taking 1 cycle to execute: in phase 1, S produces nothing; in phase 2, S produces 1 token. R cycles between three phases, also taking 1 cycle each: in phase 1, R consumes 1 token and produces nothing; in phases 2 and 3, R consumes 1 token and produces 1 token.

Since the cyclic pattern is fixed and known a priori, all static analysis properties of SDF are also applicable to CSDF [4, 18]. Figure 12 shows a schedule of actor firings in steady state when the channel between S and R is implemented by a FIFO buffer of size 1. The schedule in Fig. 12 achieves the optimal throughput of 2 samples every 6 cycles at the output of R.

4.3 Correctness and Non-Defensiveness

We say that an abstract model M is *correct* if any analysis result that can be obtained on M is *sound*, i.e., it can be achieved (and potentially improved) by some implementation. We say that M is *defensive* if the analysis results obtained on M are too conservative (with respect to a certain metric). We proceed to discuss and illustrate these notions by example.

Consider the SDF model of Fig. 9. As mentioned above, according to the SDF semantics and corresponding analysis, a buffer of size 5 is required to achieve optimal throughput. However, we have seen in Section 3.2.2 that a buffer of size 2 is sufficient to achieve the optimal throughput. The difference is due to the fact that the SDF buffer analysis conservatively allocates space for tokens from the firings of S that occur while R executes. We conclude that the SDF model of Fig. 9 is defensive.

We should note that defensiveness is intentionally defined above to be a qualitative rather than quantitative notion. It is up to the designer to decide exactly what "too conservative" means. It could mean, for instance, "at least X % more buffer space than the optimal implementation". The exact value of X depends on the application domain, and is therefore beyond our immediate scope.

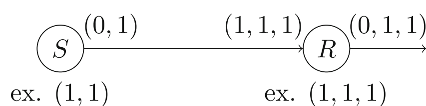


Figure 11 CSDF model of the Source-Resampler example.

Let us next turn to the CSDF model in Fig. 11. As mentioned above, CSDF analysis yields a required buffer size of 1 for this model to achieve optimal throughput. This result is unfortunately misleading. It leads the designer to believe that an implementation with buffer size 1 exists, whereas this is not the case. The reason is that R expects to receive 3 tokens consecutively on 3 cycles, but S only produces a token every 2 cycles. With a buffer similar to Buf2 shown in Fig. 7 but with capacity 1, the requirement of R cannot be satisfied. For instance, if the controller of Fig. 8 is used, then the buffer will overflow. We conclude that the CSDF model is incorrect.

5 An Operational Model for Dataflow

In order to be able to define a formal conformance relation between dataflow and hardware, we need first to define an operational model for dataflow. A variety of formal models for dataflow systems exist in the literature, e.g., see [18, 21–25], 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, 18, 24, 26]. Time is typically introduced in dataflow models by means of a special action denoted tick, modeling the lapse of one unit of time. 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:

- $n \geq 0$ is an integer representing the number of *input ports* of A. Each input port will be connected to an input queue.

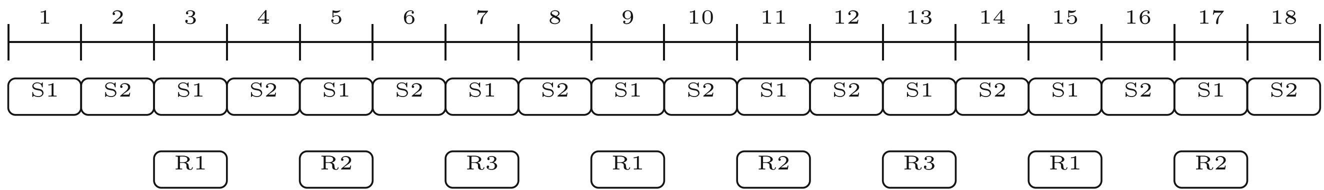


Figure 12 Firing schedule to achieve optimal throughput for the CSDF model of Fig. 11 assuming buffer size 1. The number following the actor name indicates the phase of the firing.

- $m \geq 0$ 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 $s \xrightarrow{\ell} s'$.

An example dataflow process is shown in Fig. 13. 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 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 Fig. 13. 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]

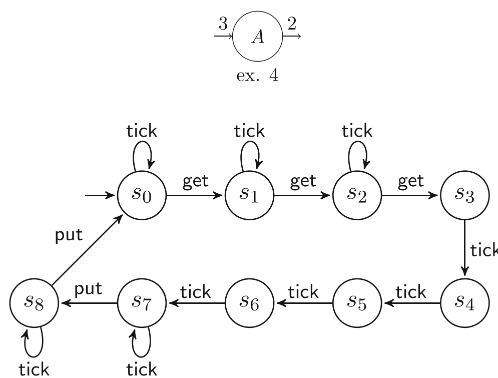


Figure 13 Example SDF process: structure (*top*) and behavior (*bottom*).

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 Fig. 14. Note that Figs. 13 and 14 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 capture dataflow models operationally, and this is part of the challenge in coming up with faithful models.

Remark 1 Although most of the examples in this paper are simple dataflow processes that fall in the SDF, CSDF, or Kahn Process Network (KPN) [21] classes,[4] the modeling framework as well as the conformance relation defined in Section 6 are more broadly applicable. In particular, contrary to what is customary [24], 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 $s \xrightarrow{\mathsf{get}_1} s_2$

[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.

[4] Examples of CSDF processes can be found in Fig. 21.

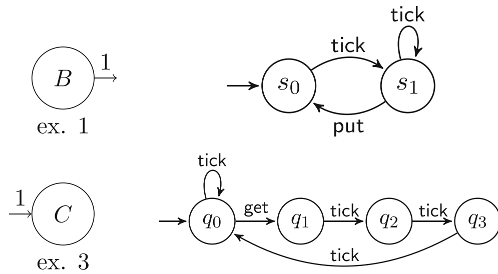


Figure 14 Source SDF process (*top*) and sink SDF process (*bottom*).

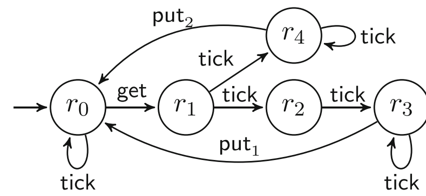


Figure 15 A non-deterministic dataflow process.

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 neither SDF, CSDF, or KPN. 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 of a non-deterministic dataflow process is shown in Fig. 15. 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 Fig. 15, where the test is replaced by a non-deterministic choice.

5.1 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 somewhat 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

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

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

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

5.2 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 Fig. 16. 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.

5.3 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, in the sense that it 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 Fig. 17 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 Fig. 13, however, would be open.

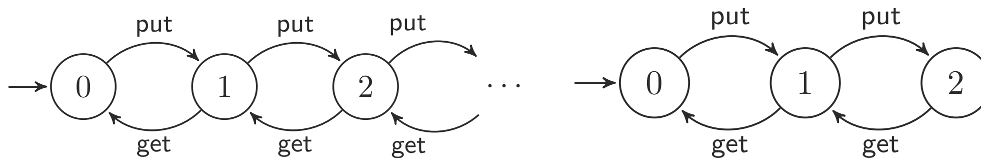


Figure 16 Queue processes: infinite queue (*left*) and queue of size 2 (*right*).

5.4 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 [27] or CSP [28]. 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. 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.

5.4.1 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.

5.4.2 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.

5.4.3 Example

As an example, a dataflow network is shown at the top of Fig. 17. It consists of the two SDF processes B and C of Fig. 14 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 bottom of Fig. 17, left and right, 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.

6 Conformance

We are now ready to attack our main problem, which is to define a formal conformance relation between the formal model for hardware defined in Section 2 and the formal model for dataflow defined in Section 5. 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.

6.1 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);

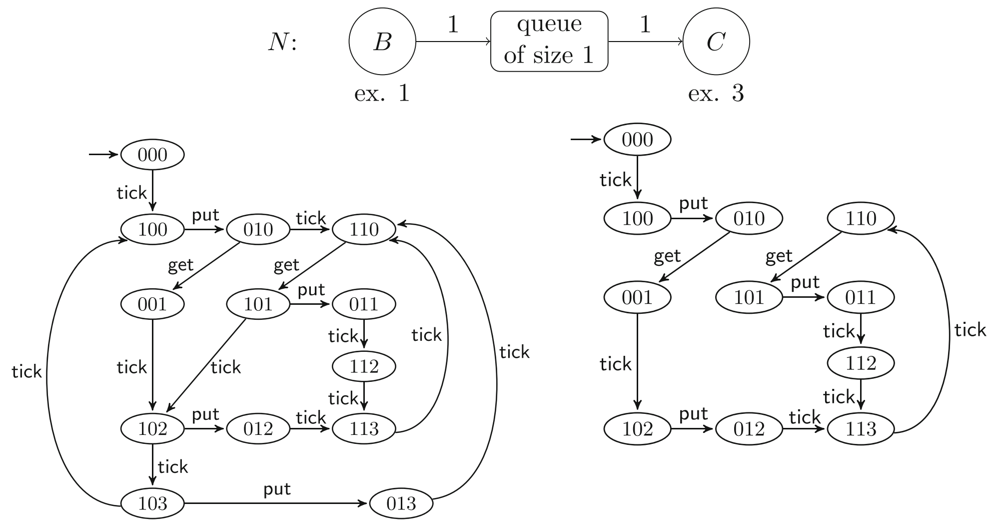


Figure 17 A closed dataflow network N (*top*) and the corresponding composite dataflow processes: non-idling (*bottom-left*) and eager (*bottom-right*).

- 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 Fig. 17. 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 Fig. 17, 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 Fig. 17, the worst-case latency between a put and a get is equal to 3 ticks, and the best-case latency is 0 ticks.

6.2 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 8).

Suppose we want to compare a closed dataflow network such as the one of Fig. 17 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 token consumptions or productions ccur.

To overcome this, we propose to make such events explicitly observable at the FSM level.[5] 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.

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.

[5] 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.

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 2 (**Conformance**) 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.

6.3 Examples of Conformance and Non-conformance

Consider the dataflow network N shown in Fig. 17 and the FSM M shown in Fig. 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 Fig. 2, M has a single infinite behavior yielding the infinite observable behavior

$$\sigma = (\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2) \cdot (\mathsf{r}_1, \mathsf{v}_1, \overline{\mathsf{r}_2}, \overline{\mathsf{v}_2}) \cdot ((\overline{\mathsf{r}_1}, \overline{\mathsf{v}_1}, \overline{\mathsf{r}_2}, \overline{\mathsf{v}_2}) \cdot (\mathsf{r}_1, \mathsf{v}_1, \mathsf{r}_2, \mathsf{v}_2) \cdot (\overline{\mathsf{r}_1}, \overline{\mathsf{v}_1}, \overline{\mathsf{r}_2}, \overline{\mathsf{v}_2}))^\omega$$

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 – Fig. 17, bottom right. Therefore, M conforms to both the eager and non-idling semantics of N.

Consider next Fig. 18. The figure shows three variants of FSM M_1 of Fig. 2 and the synchronous FSM composition of each of these variants with FSM M_2 of Fig. 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 Fig. 17, 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 Fig. 14 may incorrectly capture

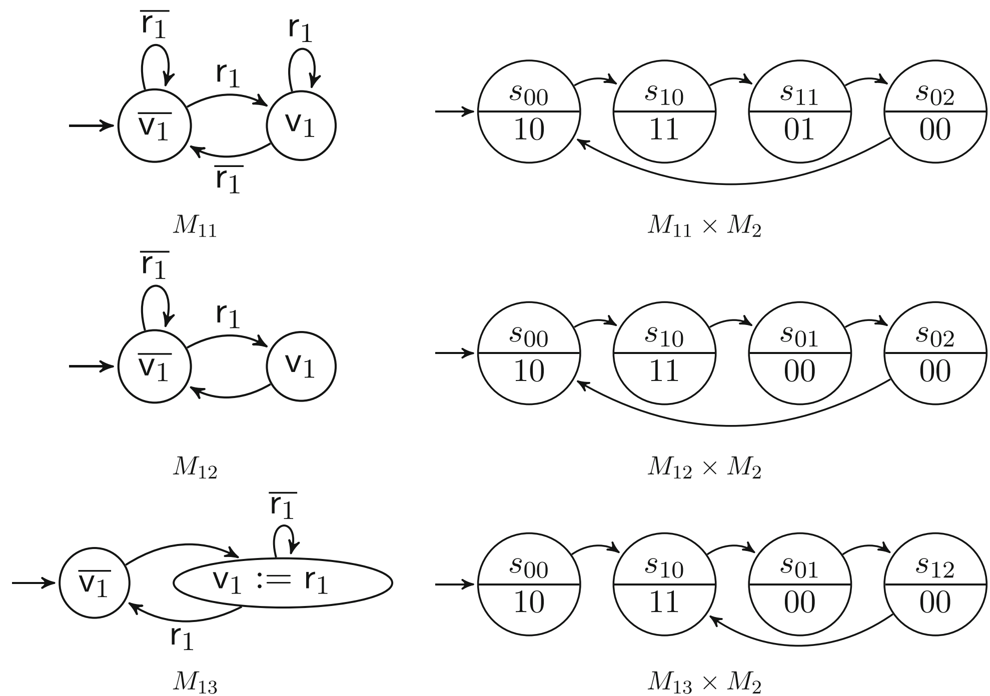


Figure 18 *Left*: three variants, M_{11}, M_{12}, M_{13}, of FSM M_1 of Fig. 2. We compose each of these with M_2 (without use of M_{buf} in the *middle*). Let states of M_2 be labeled q_0, q_1, q_2. Let states of M_{1i} be labeled s_0, s_1. Resulting three composite FSMs are shown to the right column of the figure. 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 $r = r_1 = r_2$ and $v = v_1 = v_2$ respectively in that state.

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 Fig. 19 and dataflow network N_{DC} shown at the top of Fig. 20. N_{DC} is similar to the network of Fig. 17 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 Fig. 20, bottom. 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.

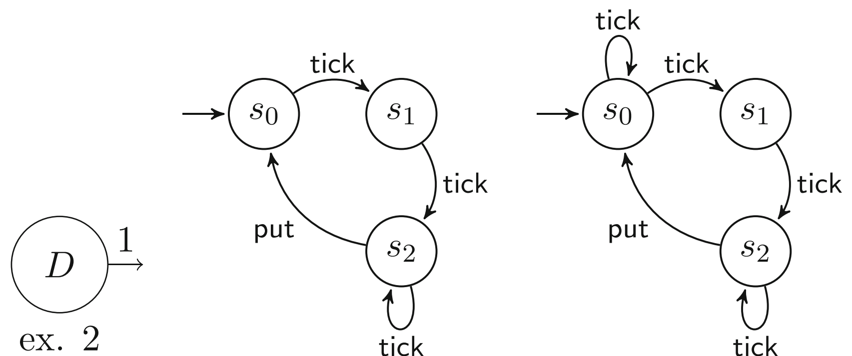


Figure 19 Two variants of SDF process D.

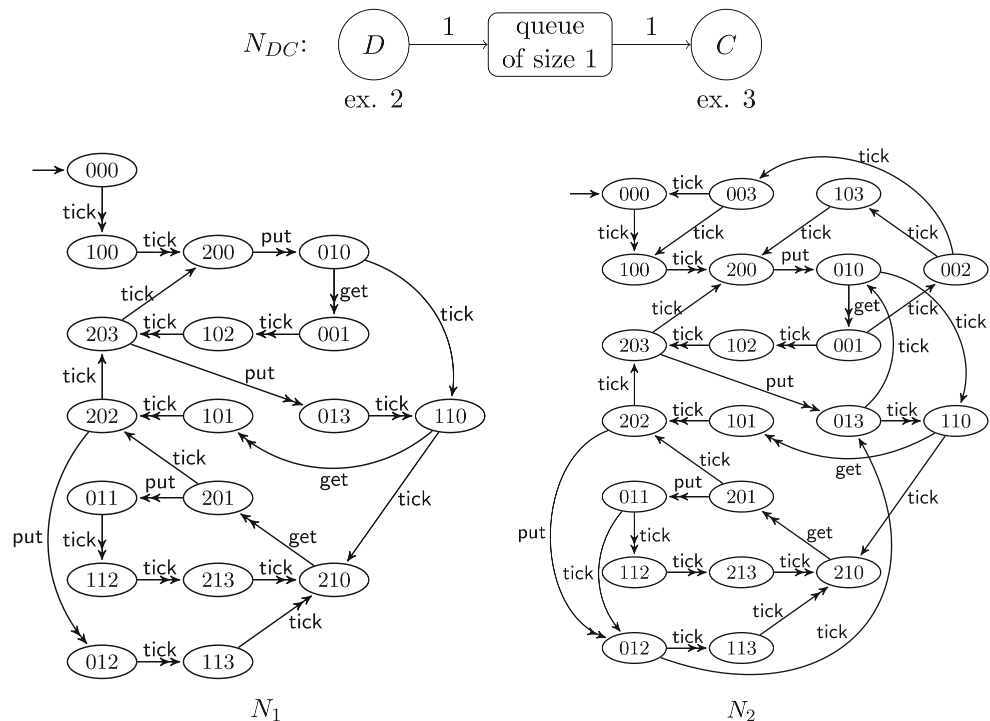


Figure 20 *Top*: closed dataflow network of actors D, C connected using queue of size 1. *Bottom-left*: composite non-idling dataflow process, N_1, using *left*-most variant of process D from Fig. 19. *Bottom-right*: composite non-idling dataflow process, N_2, using right-most variant of D. In each of the composites, the corresponding eager composition is embedded, as shown by edges with double arrowheads.

6.4 The Sampler-Resampler Example

Let us return to the Sampler-Resampler example from Sections 3 and 4, to see how the conformance relation can be used to catch the incorrectness of the CSDF model.

As a first approach, we consider the comparison of two closed systems:

- The closed FSM shown in Fig. 4, that is, the FSM formed by composing four FSMs, S, R, Buf2, and Ctrl, from Fig. 5, 7, and 8, respectively. Call this system M_{SR}.
- The closed dataflow network consisting of the two CSDF actors of Fig. 11, connected by a queue of size 1. The dataflow processes for these actors are shown in Fig. 21. Call this system N_{SR}.

Let us identify the signals of interest to be:

- For M_{SR}, the signals w and r, which are both inputs of Buf2, and outputs of Ctrl.
- For N_{SR}, the output put_s of Sampler process S, and the input get_r of Resampler process R. Note that, for now, we ignore the output of R.

As might be expected, we use the mapping $\theta = \{\mathsf{put}_s \mapsto \mathsf{w}, \mathsf{get}_r \mapsto \mathsf{r}\}$.

Then, as it can be verified, by looking in particular at the behavior of S and Ctrl, M_{SR} generates the observable behavior

$$\sigma \quad = \quad (\overline{\mathsf{w}}, \overline{\mathsf{r}})\,(\mathsf{w}, \overline{\mathsf{r}})\,(\overline{\mathsf{w}}, \overline{\mathsf{r}})\,(\mathsf{w}, \overline{\mathsf{r}})\,\cdots$$

which is mapped to the dataflow observable behavior

$$\Theta(\sigma) = \mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}_s\} \cdot \mathsf{tick} \cdot \mathsf{tick} \cdot \{\mathsf{put}_s\} \cdots$$

But $\Theta(\sigma)$ is not admitted by N_{SR}. This is because $\Theta(\sigma)$ contains two consecutive put_s actions without a get_r in between. This behavior is impossible in N_{SR}, which has a queue of size one, forcing every put_s to be followed by a get_r.

Non-conformance of M_{SR} to N_{SR} indicates the incorrectness of the latter. Note that, comparing a hardware implementation which uses a buffer of size 2, to a dataflow model which uses a queue of size 1, is not unreasonable. Recall that the queue of size 1 was obtained as the optimal one by the CSDF analysis. And one would expect that increasing the available buffer space in the hardware implementation should not result in breaching conformance.

Still, one might wonder whether we can compare N_{SR} to a variant of M_{SR} which uses a buffer of size 1 instead of 2. Call this variant M'_{SR}. Unfortunately, M'_{SR} is an erroneous FSM model, because either it reaches the error state of R, or it over-writes data in the buffer (two consecutive w's without an r in-between). The latter case will result in non-conformance with similar traces as σ and $\Theta(\sigma)$ above. In the former case, when R reaches the error state, it behaves non-deterministically. This results in arbitrary behavior also in M'_{SR}, which again results in non-conformance, as such arbitrary behavior is not part of N_{SR}.

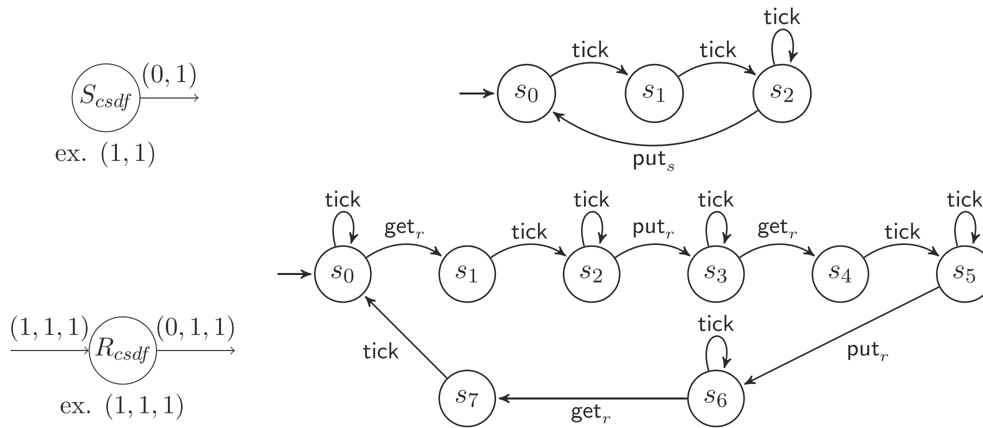


Figure 21 Dataflow processes for the CSDF actors of Fig. 11.

6.5 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.

7 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, e.g., [3, 17, 18, 20, 29–32]. Hardware generation from dataflow models has also been extensively studied, e.g., in [33–40]. 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 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. [41]), or in the context of IP integration (e.g. [2]).

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. [42] or *equivalence checking between system-level and RTL models*, e.g. [43]. 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 [44].

The works [2, 41] pursue goals similar in spirit to this paper, however, they do not define a formal conformance relation. [41] 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 [2] 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, [45, 46]). 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 [25]. 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 [25] uses an abstract, denotational model of actors, which does not answer the question how to bridge 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 [25] 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 [25] is provided in Section 8.

The glue design problem is related to controller synthesis problem [11–16, 47], for which theoretical and practical challenges remain. The same challenges are present in methods for *converter synthesis* [48, 49]. Moreover, these methods typically do not deal with buffer sizes and throughput constraints which are central in our context. Abstract dataflow models proposed in this paper can provide effective solutions, if used carefully.

Compositional verification techniques such as those in [50, 51] bear some similarity to our work in the sense that they also deal with abstractions. However, these methods focus at verification of an existing closed system and not at how to close an open system by adding glue.

Our work is also inspired by ideas that have existed for a long time in the software and programming language communities, such as abstraction by pre/post-conditions, non-defensive programming and design-by-contract [52]. Limited but practically useful versions of these ideas (e.g., types and interfaces) have found their way in widespread programming languages such as C++ and Java. More advanced concepts such as pre- and post-conditions are included in newer languages such as Spec# [53]. This paper advocates the adoption of these ideas as an integral part of the hardware design methodology.

8 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 study has been motivated by the extensive use of dataflow models in hardware design, and in particular by the *glue design problem*. We formulated this problem, defined the notions of correctness and non-defensiveness between abstract models and concrete hardware systems, and examined the scenarios under which these notions may be compromised.

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 techniques for automatically generating dataflow processes such as the ones used in the examples above from 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 [25]. This has already partly been done in [25] for SDF but not for general dataflow. It has also been done in [25] 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 [25], or another relation such as the one based on subsets of behaviors that we employ here.

Acknowledgments This work was partially supported by the Academy of Finland and by the NSF via projects *COSMOI: Compositional System Modeling with Interfaces* and *ExCAPE: Expeditions in Computer Augmented Program Engineering*. This work was also partially supported by IBM and United Technologies Corporation (UTC) via the iCyPhy consortium, and by Denso, National Instruments, and Toyota, via the UC Berkeley Center for Hybrid and Embedded Software Systems (CHESS).

References

1. Tripakis, S., Limaye, R., Ravindran, K., Wang, G. (2014). On tokens and signals: Bridging the semantic gap between dataflow models and hardware implementations. In *International conference on embedded computer systems: architectures, modeling and simulation – SAMOS XIV*.
2. Tripakis, S., Andrade, H., Ghosal, A., Limaye, R., Ravindran, K., Wang, G., Yang, G., Kormerup, J., Wong, I. (2011). Correct and non-defensive glue design using abstract models. In *7th IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis*, ser. CODES+ISSS '11 (pp. 59–68). ACM.
3. Lee, E., & Messerschmitt, D. (1987). Synchronous data flow. *Proceedings of the IEEE*, *75*(9), 1235–1245.
4. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J. (1995). Cyclo-static data flow. In *IEEE international conference acoustics, speech, and signal processing*.
5. Stuijk, S., Geilen, M., Theelen, B., Basten, T. (2011). Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications. In *International conference on embedded computer systems (SAMOS), 2011* (pp. 404–411).
6. Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541–580.
7. Ghamarian, A.H., Geilen, M., Stuijk, S., Basten, T., Theelen, B.D., Mousavi, M.R., Moonen, A.J.M., Bekooij, M. (2006). Throughput analysis of synchronous data flow graphs. In *6th international conference on application of concurrency to system design, 2006. ACSD 2006.* (pp. 25–36).
8. Janneck, J. (2011). A machine model for dataflow actors and its applications. In *Conference record of the 45th asilomar conference on signals, systems and computers (ASILOMAR), 2011*.
9. Kohavi, Z. (1978). *Switching and finite automata theory,* 2nd edn. Cambridge University Press.
10. Xilinx Inc. (2010). *Xilinx core generator*, ISE Design Suite 12.1 ed., Xilinx Inc.
11. Pnueli, A., & Rosner, R. (1989). On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on principles of programming languages*, ser. POPL '89 (pp. 179–190).
12. Ramadge, P., & Wonham, W. (1989). The control of discrete event systems. *Proceedings of the IEEE.*
13. Pnueli, A., & Rosner, R. (1990). Distributed reactive systems are hard to synthesize. In *Proceedings of the 31th IEEE symposium on foundations of computer science* (pp. 746–757).
14. Lamouchi, H., & Thistle, J. (2000). Effective control synthesis for DES under partial observations. In *39th IEEE conference on decision and control* (pp. 22–28).
15. Tripakis, S. (2004). Undecidable problems of decentralized observation and control on regular languages. *Information Processing Letters*, *90*(1), 21–28.
16. Lustig, Y., & Vardi, M. (2009). Synthesis from component libraries. In *12th international conference on foundations of software science and computational structures*, ser. FOSSACS '09 (pp. 395–409). Springer.
17. Bhattacharyya, S., Murthy, P., Lee, E. (1996). *Software synthesis from dataflow graphs.* Kluwer.
18. Stuijk, S., Geilen, M., Basten, T. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, *57*(10), 2008.
19. Wiggers, M.H., Bekooij, M.J., Smit, G.J. (2008). Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *RTAS'08* (pp. 183–194). IEEE Computer Society.
20. Wiggers, M.H., Bekooij, M.J.G., Smit, G.J.M. (2007). Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of the 44th annual design automation conference*, ser. DAC '07 (pp. 658–663).
21. Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Information processing 74, proceedings of IFIP congress 74*. North-Holland.
22. Faustini, A. (1982). An operational semantics for pure dataflow. In M. Nielsen & E. Schmidt (Eds.), *Automata, languages and programming*, ser. LNCS (vol. 140, pp. 212–224). Springer.
23. Jonsson, B. (1994). A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, *7*(4), 197–212.
24. Geilen, M., & Basten, T. (2010). Kahn process networks and a reactive extension. In *Handbook of signal processing systems*.
25. Geilen, M., Tripakis, S., Wiggers, M. (2011). The earlier the better: A theory of timed actor interfaces. In *14th international conference hybrid systems: computation and control (HSCC'11)*. ACM.
26. Sriram, S., & Bhattacharyya, S.S. (2009). *Embedded multiprocessors: scheduling and synchronization*, 2nd ed. CRC Press.
27. Milner, R. (1982). *A calculus of communicating systems.* Springer.
28. Hoare, C. (1985). *Communicating sequential processes.* Prentice Hall.
29. Moreira, O.M., & Bekooij, M.J.G. (2007). Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, *2007*(83710), 1–15.
30. Wiggers, M.H., Bekooij, M.J.G., Smit, G.J.M. (2007). Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proceedingsof the 10th international workshop on software & compilers for embedded systems*, ser. SCOPES '07 (pp. 11–22).
31. Kwok, Y.-K., & Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, *31*(4), 406–471.
32. Ghosal, A., Limaye, R., Ravindran, K., Tripakis, S., Prasad, A., Wang, G., Tran, T., Andrade, H. (2012). Static dataflow with access patterns: semantics and analysis. In *Design automation conference (DAC)*.
33. Lauwereins, R., Engels, M., Adé, M., Peperstraete, J.A. (1995). Grape-II: a system-level prototyping environment for DSP applications. *Computer*, *28*(2), 35–43.
34. Williamson, M., & Lee, E. (1996). Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *ASILOMAR*.
35. Horstmannshoff, J., & Meyr, H. (1999). Optimized system synthesis of complex RT level building blocks from multirate dataflow graphs. In *12th international symposium on system synthesis*. IEEE.
36. Jung, H., Yang, H., Ha, S. (2008). Optimized RTL code generation from coarse-grain dataflow specification for fast HW/SW cosynthesis. *Journal of Signal Processing System*, *52*(1), 13–34.

37. Janneck, J., Miller, I., Parlour, D., Roquier, G., Wipliez, M., Raulet, M. (2008). Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In *Signal processing systems*.
38. Thavot, R., Mosqueron, R., Alisafaee, M., Lucarz, C., Mattavelli, M., Dubois, J., Noel, V. (2008). Dataflow design of a co-processor architecture for image processing. In *DASIP*.
39. Dubois, J., Thavot, R., Mosqueron, R., Miteran, J., Lucarz, C. (2009). Motion estimation accelerator with user search strategy in an RVC context. In *IEEE ICIP'09*.
40. Olsson, T., Carlsson, A., Wilhelmsson, L., Eker, J., von Platen, C., Diaz, I. (2010). A reconfigurable OFDM inner receiver implemented in the CAL dataflow language. In *Circuits and systems (ISCAS)*.
41. Hansson, A., Wiggers, M., Moonen, A., Goossens, K., Bekooij, M. (2009). Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *Computers Digital Techniques, IET*, *3*(5), 398–412.
42. Ghenassia, F. (Ed.) (2005). *Transaction-level modeling with systemC.* Springer.
43. Pixley, C. (2009). Practical considerations concerning HL-to -RT equivalence checking. In *Hardware and software: verification and testing*, ser. LNCS (vol. 5394). Springer.
44. Clarke, E., Kroening, D., Yorav, K. (2003). Behavioral consistency of C and verilog programs using bounded model checking. In *DAC*.
45. van Glabbeek, R.J. (1990). The linear time-branching time spectrum. In *CONCUR'90* (pp. 278–297). Springer.
46. van Glabbeek, R., & Goltz, U. (2000). Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, *37*(4–5), 229–327.
47. Ehlers, R., Lafortune, S., Tripakis, S., Vardi, M. (2014). Bridging the gap between supervisory control and reactive synthesis: case of full observation and centralized control. In *12th IFAC international workshop on discrete event systems (WODES)*.
48. Bhaduri, P., & Ramesh, S. (2006). Synthesis of synchronous interfaces. In *Proceedings of the 6th international conference on application of concurrency to system design*.
49. Passerone, R., de Alfaro, L., Henzinger, T.A., Sangiovanni-Vincentelli, A.L. (2002). Convertibility verification and converter synthesis: two faces of the same coin. In *Proceedings of the international conference on computer-aided design*.
50. Henzinger, T., Qadeer, S., Rajamani, S. (1998). You assume, we guarantee: Methodology and case studies. In *CAV'98*, ser. LNCS (vol. 1427). Springer-Verlag.
51. McMillan, K. (1997). A compositional rule for hardware design refinement. In *Computer aided verification (CAV'97)*, ser. LNCS (vol. 1254). Springer-Verlag.
52. Meyer, B. (1992). Applying design by contract. *Computer*, *25*(10), 40–51.
53. Barnett, M., Leino, K.R.M., Schulte, W. (2005). The spec# programming system: an overview. In *International conference on construction and analysis of safe, secure, and interoperable smart devices*, ser. CASSIS'04 (pp. 49–69). Springer.

Stavros Tripakis is an Adjunct Associate Professor at the University of California, Berkeley, and an Associate Professor at Aalto University, Finland. He received a Ph.D. degree in Computer Science in 1998 at the Verimag Laboratory, Joseph Fourier University, Grenoble, France. He was a Postdoc at UC Berkeley from 1999 to 2001, a CNRS Research Scientist at Verimag from 2001 to 2006, and a Research Scientist at Cadence Research Labs, Berkeley, from 2006 to 2008. His research interests include formal methods, computer-aided system design, and cyber-physical systems. Dr. Tripakis was co-Chair of the 10th ACM & IEEE Conference on Embedded Software (EMSOFT 2010), and Secretary/Treasurer (2009-2011) and Vice-Chair (2011-2013) of ACM SIGBED. His h-index, according to Google Scholar, is 40.

Rhishikesh Limaye Staff R&D Engineer, LabVIEW R&D, National Instruments, Berkeley, CA. Rhishikesh Limaye is a Staff R&D Engineer at National Instruments (NI). He received B.Tech. and M.Tech. in Electrical Engineering from the Indian Institute of Technology, Bombay in 2004, and received M.S. in Electrical Engineering and Computer Science from the University of California at Berkeley in 2010. Since then he has been with the NI LabVIEW R&D group. His research interests can be summed up as using and developing models, tools, and algorithms, for computer-aided verification and cyber-physical system development.

Kaushik Ravindran Senior R&D Engineer, LabVIEW R&D, National Instruments, Berkeley, CA. Kaushik Ravindran is a Senior R&D Engineer at National Instruments (NI). He received a BS degree in Computer Engineering from the Georgia Institute of Technology in 2001 and a PhD in Electrical Engineering and Computer Sciences from the University of California at Berkeley in 2007. He was a student intern at the IBM T. J. Watson Research Center in 2003. He has served with the NI LabVIEW R&D group since 2007. His research interests include system design, models of computation, and optimization algorithms. He holds 5 patents and has authored or co-authored over 30 publications.

Guoqiang Wang received the B.E. and M.E. degrees from Xi'an Jiaotong Univerisity, China, in 1995 and 1998, respectively. He obtained the M.S. degree in Computer Sciences and the Ph.D. degree in Electrical Engineering and Computer Sciences from the University of California, Berkeley, in 2007 and 2008, respectively. Dr. Wang is a research scientist at United Technologies Research Center. He mainly focuses on design, analysis and optimization of cyber-physical systems. Prior to UTRC, he was employed at National Instruments Corporation.

Hugo A. Andrade is a Principal Architect in Software Marketing at National Instruments, where he has been employed since 1989. He currently focuses on advanced applications of the LabVIEW RIO Architecture, around system-level design and cyber-physical system design. He has been visiting industrial fellow at the University of California, Berkeley (2007-), and was the founding manager and tech lead of the NI Berkeley R&D site, where he worked on advanced tools for system level design and synthesis to heterogeneous platform. He serves as liaison to academic and industrial research labs in the area. Hugo holds 60 patents and over 20 academic publications in the areas of instrumentation software, hardware/software interfacing, reconfigurable computing, graphical programming, models of computation and system-level design. He earned BS degrees in ECE and CS and an MS degree in ECE from the University of Texas at Austin.

Dr. Arkadeb Ghosal received a Bachelor's degree in Electrical Engineering from the Indian Institute of Technology at Kharagpur, India in 2001 and a Master's and Doctoral (Ph.D.) degree in Electrical Engineering and Computer Sciences from the University of California at Berkeley in 2004 and 2008, respectively. From 2008 to 2010, Dr. Ghosal worked at General Motors Research in the Electrical Control Integration Lab on different aspects of automotive electronic control system design. Dr. Ghosal is currently an R&D Engineer at National Instruments in Berkeley, California. His research interests include modeling and analysis of real-time systems, design methodologies for heterogeneous platforms, and graphical system design frameworks.