Correct and Non-Defensive Glue Design using Abstract Models*

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

Hugo Andrade, Arkadeb Ghosal
Rhishikesh Limaye
Kaushik Ravindran
Guoqiang Wang, Guang Yang
National Instruments Corp.
Berkeley, CA, USA
{first.lastname}@ni.com

Jacob Kornerup
Ian Wong
National Instruments Corp.
Austin, TX, USA
{first.lastname}@ni.com

ABSTRACT

Current hardware design practice often relies on integration of components, some of which may be IP or legacy blocks. While integration eases design by allowing modularization and component reuse, it is still done in a mostly ad hoc manner. Designers work with descriptions of components that are either informal or incomplete (e.g., documents in English, structural but non-behavioral specifications in IP-XACT) or too low-level (e.g., HDL code), and have little to no automatic support for stitching the components together. Providing such support is the *glue design problem*.

This paper addresses this problem using a model-based approach. The key idea is to use high-level models, such as dataflow graphs, that enable efficient automated analysis. The analysis can be used to derive performance properties of the system (e.g., component compatibility, throughput, etc.), optimize resource usage (e.g., buffer sizes), and even synthesize low-level code (e.g., control logic). However, these models are only abstractions of the real system, and often omit critical information. As a result, the analysis outcomes may be defensive (e.g., buffers that are too big) or even incorrect (e.g., buffers that are too small). The paper examines these situations and proposes a correct and non-defensive design methodology that employs the right models to explore accurate performance and resource trade-offs.

*This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota. This work was also supported by direct contribution and funding from the National Instruments Corporation.

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: Design Aids

General Terms

Design, Theory, Verification

Keywords

Glue design, Dataflow, Abstraction, Non-defensiveness

1. INTRODUCTION

Both hardware and software design have historically evolved toward higher-level models and languages. In software, programming languages have evolved from assembly to structured to object-oriented programming. Hardware design has evolved from transistor and gate layout to logic synthesis to high-level synthesis. This evolution process, sometimes called "raising the level of abstraction", allows the designer to focus on design properties that matter most, while hiding lower-level details. Abstraction is essential in order to manage the ever-increasing size and complexity of designs.

Another aspect of modern hardware design flows, equally important in coping with large and complex systems, is the support for *component-based design*. This is manifested by methods that rely on integration of components such as Intellectual Property (IP) blocks from native and third-party sources, for instance, Xilinx CoreGen [36], National Instruments LabVIEW FPGA [25], or the OpenCores library [26]. Complex designs are created by stitching together multiple components.

While component-based design allows for modularization and component reuse, integration is still an ad hoc process, lacking the support of rigorous methodology, theory, and tools. In particular, the design of the requisite communication and control logic to connect the blocks is a manual and error-prone process. The interfaces of these blocks expose low-level control and timing artifacts that the designer must manually reconcile to create systems that are not only valid (i.e., functionally correct) but also meet performance requirements (e.g., throughput and area constraints). We call this the *glue design problem*.

In this paper, we approach this problem as follows. Abstract, high-level models, called *actors*, are first constructed for individual components. The actors are then composed

into a *system-level model*. The system-level model enables automated analysis of key correctness and performance properties of the system (compatibility of components, throughput, buffer sizes, etc.). The results of this analysis can in turn be used to synthesize (manually, or even automatically) the glue to connect the components together.

The use of abstract models and efficient algorithms is key. Even in cases where the same information could in theory be derived using the detailed, low-level descriptions of individual blocks (e.g., HDL code), this is often infeasible in practice because of state-explosion problems.

However, these high-level models, by the fact that they are abstractions of the real system, often omit critical information. As a result, what we find in our study is that the outcomes of the analysis of the high-level models are often *defensive* (e.g., too conservative) and can even be *incorrect*. The main contribution of our paper is a careful examination of the abstraction properties of these high-level models.

For instance, Static Dataflow (SDF)[1] models have been applied to abstract hardware components [35, 10, 6]. Efficient algorithms are available for SDF models to compute performance metrics such as throughput and buffer sizes, as well as execution schedule [3, 31, 11, 33, 24, 6]. All this information can be used to design the glue. However, an SDF model typically loses information about the precise timing of consumption and production of *tokens* (i.e., data values) by an actor during a firing cycle. This loss of information results in defensive glue designs that conservatively estimate the resources needed, e.g., designs with buffers that are larger than necessary.

Cyclo-Static Dataflow (CSDF) [4] is a generalization of SDF that attempts to remedy this problem. CSDF allows to "break" a firing into phases, and describe individual consumptions and productions of tokens in each of these phases. The problem is, however, that the standard CSDF token consumption semantics relies on the hypothesis that in every phase an actor will wait ("stall") until a sufficient number of tokens has accumulated at its input queues. Unfortunately, many hardware components are built in a way so that they cannot stall when started. For example, a component may impose the following requirement: once it begins receiving data samples, it must continue to receive samples for 8 consecutive clock cycles. CSDF cannot express this, and as a result, buffer sizing based on CSDF can be overly aggressive in this case. In fact, if glue is synthesized automatically from the CSDF model this glue will be incorrect (buffers will overflow).

Instead of dataflow models like SDF or CSDF, one could use Finite State Machines (FSMs). Indeed, FSMs can express interaction protocols at the cycle-accurate level, thus preserving the information lost by SDF or CSDF, while still hiding unnecessary information, e.g., data values. Unfortunately, automatic glue synthesis based on general FSM models is a non-trivial problem, both theoretically as well as practically.

For situations where using SDF/CSDF is not sufficient and using FSMs is infeasible, a possible remedy is a new model, called SDF-AP, introduced in this paper. SDF-AP attempts to strike a balance between the analyzability of SDF-like models while accurately capturing the interface timing behavior by including *access patterns*.

[1] SDF has been called *synchronous* dataflow by the authors that introduced the model [20], but the model is fundamentally asynchronous, since actors can fire independently and asynchronously. For this reason, and in order not to confuse SDF with truly synchronous models such as synchronous FSMs, we prefer the term "static".

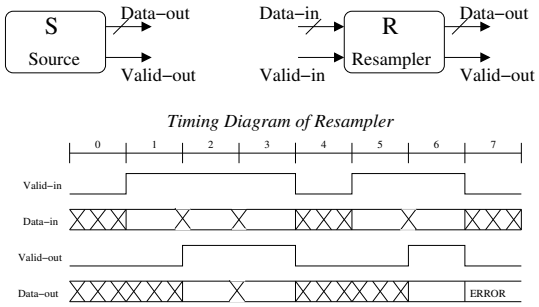


Figure 1: Interfaces of Source and Resampler IP blocks and timing diagram of Resampler.

In the rest of the paper we illustrate the issues of defensiveness and correctness using concrete examples, and discuss the conditions under which a correct and non-defensive design methodology can be obtained. Section 2 describes the glue design problem. Section 3 discusses the SDF, CSDF, and SDF-AP models and respective analysis methods, as well as the issues of defensiveness and correctness that arise by using such models. Section 4 presents a case study. Section 5 discusses related work. Section 6 concludes the paper.

2. THE GLUE DESIGN PROBLEM

We motivate the need for formal compositional abstractions in hardware design with a simple yet realistic use case. Fig. 1 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. 2.

Alternatively, Resampler could be implemented using the FIR IP block in the Xilinx CoreGen library [36]. 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. 1 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 2: 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. 1 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.

2.1 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, as mentioned in the introduction. As a first remedy to this problem, we model actors using Finite State Machines (FSMs) such as Mealy or Moore machines [17]. In these FSMs we often abstract away the data signals and preserve only the control signals, which allows us to obtain simpler and smaller FSMs. Another key characteristic is that the error conditions are modeled explicitly.

Fig. 3 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.

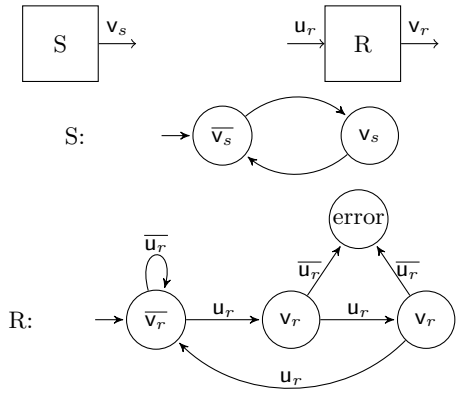


Figure 3: FSM models for the Source (S) and Resampler (R) actors.

2.2 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. 4. In the sequel, we elaborate on the salient components of glues.

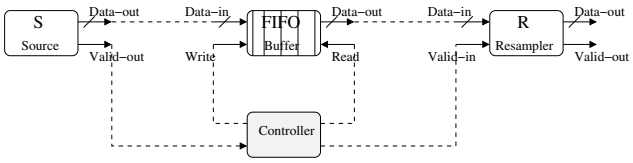


Figure 4: System built by connecting the Source and Resampler blocks of Fig. 1 with glue that includes a FIFO buffer and a controller.

2.2.1 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.

Fig. 5 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.

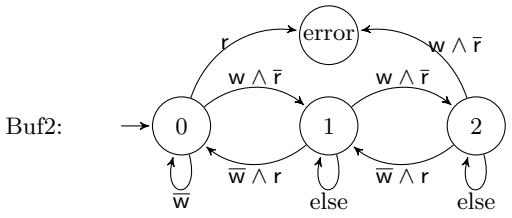
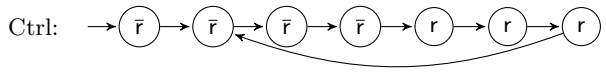


Figure 5: FSM for a buffer of size 2.

2.2.2 Control Logic

The glue may also include some type of control logic to control the execution of actors. Fig. 6 shows an example.[2] Controller Ctrl has the interface of Fig. 4: 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. 5, this controller guarantees that the requirements of the Resampler are satisfied.



At all states: $(\mathsf{w} \Leftrightarrow \mathsf{v}_s) \wedge (\mathsf{u}_r \Leftrightarrow \mathsf{r})$.

Figure 6: Control logic.

2.3 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, a system is a *closed* FSM, that is, an FSM without input signals. For example, Fig. 4 represents a system. Note that even though the system is closed, it may still contain multiple behaviors, because we allow FSMs to be non-deterministic.

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. 3 is incorrect, because the error state of R is reachable; the system built by composing S, R, Buf2, and Ctrl, of Figs. 3,5,6, 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.

2.4 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 [28, 30] 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 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. 4, 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 [29, 18, 32, 21]. 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. In the following sections, we advance an abstraction-oriented methodology to effectively tackle the problem using high-level models.

[2] The FSMs presented so far are Moore machines, where the output at a given state depends only on the state and not on the input. The FSM of Fig. 6 is a Mealy machine, since the output w depends "combinationally" on the input v_s. 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. We find this notation more appropriate, since the inputs may change during a clock cycle, and the outputs will change accordingly, while the state remains fixed.

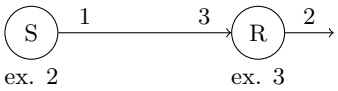


Figure 7: SDF model of the Source-Resampler example.

3. ABSTRACT MODELS AND DESIGN

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. In what follows, we review the abstract models that we use, and introduce the notions of correctness and non-defensiveness along the way.

3.1 SDF Model

Many streaming applications can be specified as SDF models [20]. An SDF model consists of a finite set of 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.

Fig. 7 shows a possible SDF model for the Source-Resampler example from Fig. 3. 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 *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. 1, or from FSM models like the ones in Fig. 3. 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 [20, 3]. 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. 7 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 [31, 34]. 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 [3, 31, 34, 33]. Fig. 8 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.

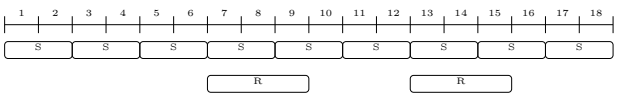


Figure 8: Firing schedule to achieve optimal throughput for the SDF model of Fig. 7 assuming buffer size 5.

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. 8 can be naturally incorporated as the buffer and controller components of Fig. 4, 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.

3.2 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.

Fig. 9 shows a possible CSDF model for the Source-Resampler example of Fig. 3. 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

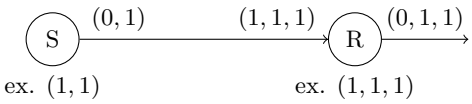


Figure 9: CSDF model of the Source-Resampler example.

CSDF [4, 31]. Fig. 10 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. 10 achieves the optimal throughput of 2 samples every 6 cycles at the output of R.

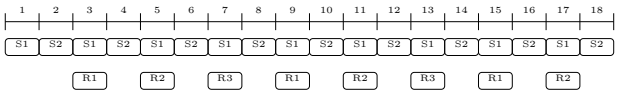


Figure 10: Firing schedule to achieve optimal throughput for the CSDF model of Fig. 10 assuming buffer size 1.

3.3 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. 7. 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 2.2.2 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. 7 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.

Let us next turn to the CSDF model in Fig. 9. 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. 5 but with capacity 1, the requirement of R cannot be satisfied. For instance, if the controller of Fig. 6 is used, then the buffer will overflow. We conclude that the CSDF model is incorrect.

The challenge to the designer is to choose the right level of abstraction that guarantees correct implementation while enabling accurate analysis. In the following section, we propose an abstraction of the interface timing behavior that retains important analyzability properties and results in non-defensive implementations that achieve better performance and improved resource usage.

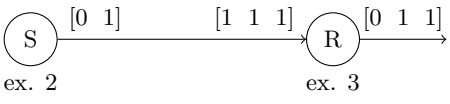


Figure 11: SDF-AP model of the Source-Resampler example.

3.4 SDF-AP Model

We have seen above that SDF models can be defensive due to loss of critical information regarding the precise timing of token consumptions and productions. We propose an extension to the SDF model that allows a more fine-grained specification of token access patterns and associated timing properties at its inputs and outputs. We call this model Static Dataflow with Access Patterns (SDF-AP).

An SDF-AP model is similar to an SDF models, except that each input and output terminal of an actor is annotated by a vector called *consumption pattern* (CP) for an input terminal, and *production pattern* (PP) for an output terminal. Each such vector has length equal to the execution time of the actor. Each element of the vector describes the number of tokens to be processed in the corresponding cycle.

It is most common for hardware implementations to process at most one data token per cycle, so CPs and PPs are assumed to consist of only 0's and 1's in the following, where 1 denotes consumption/production of one token in a cycle, while 0 denotes no consumption/production. Nevertheless, the concept can be easily adapted to capture actors that process more than 1 token per cycle by standardizing the hardware protocol of terminals.

SDF-AP should not be confused with CSDF, despite the fact that the two models share great similarity in syntactic notation. The key difference is that SDF-AP defines strict timing for all the tokens in one firing, while CSDF defines the timing for each phase but is not at all strict between the firing of phases (in particular, CSDF allows stalling between phases). Formally, the semantics of SDF-AP can be derived by starting from the same class of possible schedules as in the corresponding CSDF model and then restricting this class by removing those schedules that do not satisfy the strictness requirements (for instance, schedules where an actor stalls during a firing).

Fig. 11 shows a possible SDF-AP model for the Source-Resampler example. This SDF-AP model contains two actors, S and R. S produces 1 token every time it fires, and its execution takes 2 cycles. The access pattern additionally specifies that no token is produced on the first cycle and one token is produced on the second cycle. R consumes 3 tokens and produces 2 tokens every time it fires, and its execution takes 3 cycles. Similarly, the access pattern at the input to R specifies that it consumes one token every cycle for the duration of its firing. The access pattern at the output pf R specifies that it does not produce any token in the first firing, and produces one token each at the second and the third firing.

The SDF-AP model retains key static analysis properties of SDF. Fig. 12 shows a schedule of actor firings in steady state when the channel between S and R is implemented

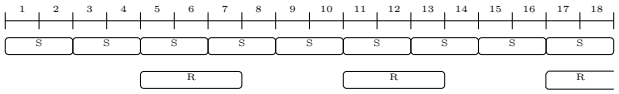


Figure 12: Firing schedule to achieve optimal throughput for the SDF-AP model of Fig. 11 assuming buffer size 2.

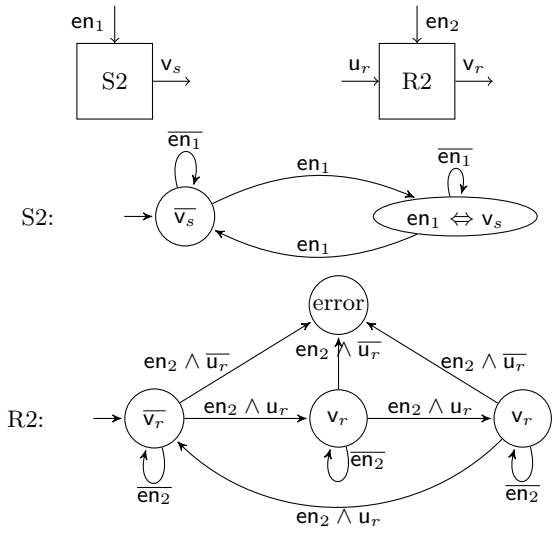


Figure 13: Actors with enable signals.

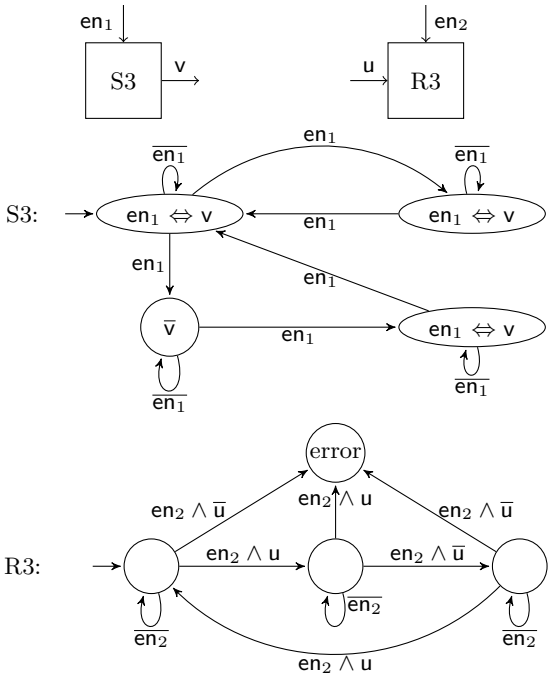


Figure 14: A system with two actors. Actor S3 has a non-deterministic execution time.

by a FIFO buffer of size 2. This is the minimum buffer size required for a valid deadlock-free execution of the SDF-AP model. The resulting schedule also achieves the optimal throughput of 2 samples every 6 cycles at the output of *R*. We conclude that the SDF-AP model of Fig. 11 is both correct and non-defensive.

3.5 Trade-offs in Glue Design

It is important to understand that a model is correct or non-defensive only with respect to a concrete hardware implementation. To see this, consider a slight variation of the Source-Resampler example, with the actors having additional *enable* input signals, as shown in Fig. 13. Enable signals control when a firing occurs, and allow actors to "stall" waiting for input tokens, or for output buffer space to become available. In a hardware implementation enable signals may correspond to the clock input.

Enable signals accord greater flexibility in the choice of abstractions for the composed system. Indeed, the SDF and SDF-AP abstractions from Figs. 7 and 11 remain correct for the system of Fig. 13. But additionally, the CSDF abstraction of Fig. 9 now also becomes correct, since actor R2 can be stalled between firings and does not need to be provided with 3 tokens in 3 successive cycles.

The schedules in Figs. 8, 10, and 12 corresponding to the SDF, CSDF, and SDF-AP models are all valid references for the design of the glue that achieves optimal throughput. The choice of the model has implications on the nature of the resulting glue. The SDF model is most defensive with respect to buffer size, while the CSDF model achieves the smallest buffer size. However, the glue for the CSDF model requires a more elaborate control logic compared to the SDF model to regulate the *enable* signal according to the schedule in Fig. 10. The advantage of abstract models is that they allow these trade-offs to be rigorously evaluated at design time. The challenge to the designer is to select the right abstraction that captures essential properties of the system, and reason about system requirements to select the appropriate glue for implementation.

3.6 Non-Determinism

We have so far considered examples where actors can be modeled as deterministic FSMs. This is not always the case. In particular, actors often have data-dependent behavior. When data is abstracted from the HDL to the simpler FSM model, such behavior is typically captured as a non-deterministic FSM. A frequent case is when different data values result in different processing times. In that case, data abstraction will yield an actor model with a non-deterministic execution time.

Non-determinism can be handled in our approach. We illustrate this with the example of Fig. 14 which is a variation of Fig. 13. Both actors *S3* and *R3* can be stalled in any clock cycle by making the corresponding enable signal *false*.

S3 produces two output data samples in one of two ways: (a) two samples in two consecutive enabled cycles, or (b) one sample in one enabled cycle, another in the third enabled cycle. The FSM for *S3* is non-deterministic: if en_1 is true at the initial state, then there are two possible next states and corresponding behaviors – the top-right state capturing behavior (a) and the bottom-left state capturing behavior (b). Note that *S3* is a Mealy (not Moore) machine: at the initial state, for example, the output v is equal to the input en_1. Thus, assuming en_1 is true at the initial clock cycle, v is also true then. Assuming en_1 is true during the first three clock cycles, one possible output behavior (behavior (b)) is $\mathsf{v}\ \overline{\mathsf{v}}\ \mathsf{v}$. Another (behavior (a)) is $\mathsf{v}\ \mathsf{v}$.

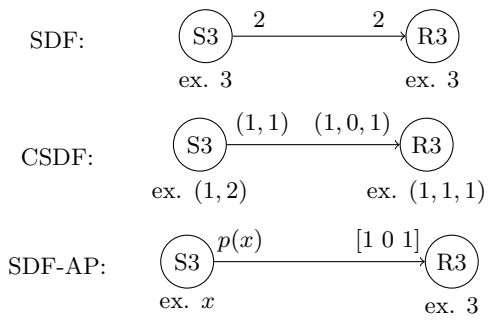


Figure 15: SDF, CSDF, and SDF-AP models for the example of Fig. 14.

R3 requires 2 input data samples in 3 enabled cycles (note that this is different from *R2* which requires 3 samples in 3 cycles). Availability of input samples is signaled by u being *true*. The first input sample must be made available in the first enabled cycle, and the second input sample must be made available in the third enabled cycle. If this requirement is violated, *R3* moves to an "error" state.

Possible SDF, CSDF, and SDF-AP models of this system are shown in Fig. 15. We explain below how these models are derived, how they are used for reasoning about properties such as throughput, and how they lead to various system implementations.

The SDF model is constructed by choosing the worst-case execution time 3 for *S3*. This is a reasonable choice, as SDF is known to be monotonic with respect to execution times, that is, smaller execution times cannot worsen the throughput [7]. Given this choice, one can obtain an implementation using buffer of size 4 and a suitable controller to achieve the worst-case throughput of 2 samples every 3 cycles. The SDF model is correct as such an implementation does not violate the requirements of *R3* even when the execution time of *S3* varies between 2 or 3 at run time.

The CSDF model splits execution of *S3* into two phases of worst-case execution times of 1 and 2 respectively. As in the SDF analysis, CSDF analysis gives a correct implementation with a buffer of size 1, and predicts worst-case throughput of 2 samples every 3 cycles.

The SDF-AP model captures both possibilities for the execution time of *S3*. In Fig. 15, the execution time x of *S3* can take values 2 or 3, and the corresponding production patterns are $p(2) = [11]$ and $p(3) = [101]$. With this model, it is possible to derive an implementation that uses a buffer of size 1. The throughput is 2 samples every 3 cycles.

In summary, for this example, all three abstract models correctly encode non-deterministic execution times. The SDF model is most defensive in terms of buffer size, while the CSDF and SDF-AP models result in non-defensive implementations with different glue designs.

4. CASE STUDY

In this section, we summarize our experience in using our design approach to deploy an Orthogonal Frequency Division Multiple Access (OFDMA) application on a Xilinx Virtex-5 FPGA target. The OFDMA application is a computation intensive component of the downlink physical layer of the Long Term Evolution (LTE) mobile technology standard. The study illustrates how an SDF model abstraction coupled with access pattern extensions enables reasoning about key application properties (such as buffer size and scheduling strategy) and facilitates glue design to generate efficient implementations.

4.1 SDF Model

The application building blocks are composed of user defined actors and hardware IP from the Xilinx Coregen library [36]. Following the approach described in the previous sections, the SDF abstraction is a viable starting point to reason about the composition of these blocks. Fig. 16 shows the top level SDF model of the OFDMA application, along with the execution times of the actors. The ZeroPad actor appends zeros at the DC and edge subcarriers in the input stream, and generates 2048 frequency domain complex values. The following block performs a 2048-point IFFT and appends a cyclic prefix to the result. The 25/24 and 25/32 FIR blocks perform sample rate conversion, and produce an output stream suitable for a 25 MHz D/A converter on the hardware platform. The performance target is a throughput of 25M samples/sec at the output D/A. The SDF model for the OFDMA application is reviewed in greater detail in [16].

4.2 SDF-AP Model

The IFFT and FIR actors are configurable IPs from the Xilinx Coregen library, while the rest are user defined actors. We apply the access pattern abstraction to characterize the actors in the application, following the timing diagrams presented in related data sheets. One key direction for future work is the automated characterization of these patterns from IP interface formalisms or standards such as IP-XACT (IEEE 1685) [12]. Nevertheless, even a manual specification of access patterns is helpful for subsequent analysis and non-defensive glue design.

The ZeroPad actor consumes 600 tokens, produces 2048 tokens, and takes 2048 cycles to execute. Its token consumption pattern is $[1^{300}0^{600}1^{300}0^{848}]$, where a^n denotes a string of a's of length n. The production pattern is $[1^{2048}]$, that is, one token is produced in every cycle.

The 25/24 FIR resampler is a configurable IP from the Xilinx CoreGen library. In this case, the user selects a specific configuration from a number of available choices for each IP, trading off area and performance. The actor instance in the OFDMA application consumes 24 tokens, produces 25 tokens, and takes 144 cycles to execute. Its consumption pattern is $[(100000)^{24}]$ (i.e., it consumes one token every sixth cycle starting with the first cycle). Its production pattern is $[(000001)^{23}001001]$.

Note that in each actor the lengths of the production and consumption patterns are equal to the execution time of the actor.

4.3 Glue Design

Once the SDF and SDF-AP models have been specified, the next step is to validate the models and evaluate tradeoffs to generate efficient hardware implementations. For the SDF model in Fig. 16, using the self-timed scheduling and sizing algorithm of [31], we get buffer sizes of [600 2048 2133 56 25] (from left to right) for the channels to meet the target throughput of 25M samples/sec.

If we employ the SDF-AP model, these sizes can be further reduced while maintaining correctness and throughput.

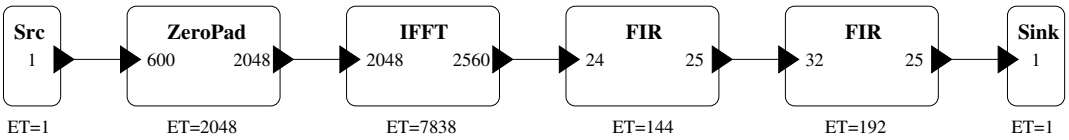


Figure 16: SDF model of the OFDMA application.

Firstly, the production pattern of ZeroPad, $[1^{2048}]$, matches exactly with the consumption pattern of IFFT. Thus, we need only a size-1 buffer between ZeroPad and IFFT, provided we schedule IFFT to start executing 1 cycle after start of firing of ZeroPad. The production pattern of 25/24 FIR, $[(000001)^{23}001001]$, does not match exactly with consumption pattern of 25/32 FIR, $[(100000)^{32}]$. Still we can schedule them such that buffer of size 2 suffices. After these two optimizations using access patterns, we get the buffer sizes of [600 1 2133 2 25], which are better than the sizes given by the SDF self-timed scheduling algorithms. These buffer sizes, along with the schedule obtained by SDF-AP analysis, allows us to generate the appropriate glue to synthesize a non-defensive implementation that is correct-by-construction.

The modeling, analysis, and glue design solutions have been prototyped as extensions to the National Instruments LabVIEW FPGA framework [25]. LabVIEW FPGA can then be used to synthesize the design, generate hardware implementations, and deploy them on Xilinx FPGAs.

5. RELATED WORK

Our work is 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 [23]. 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#. This paper advocates the adoption of these ideas as an integral part of the hardware design methodology.

Our work is closely related to other abstraction-based design methodologies such as *transaction-level modeling* (TLM) [8]. What often seems to be missing in methodologies such as TLM is a rigorous formalization of the relation between the concrete (e.g. cycle-accurate) models and the abstract (e.g. transaction-level) models. Moreover, it is unclear how such a relation could be defined since these models "live in different worlds" semantically (e.g. clock cycles vs. transactions). In our approach, we are careful to use models that can be given semantics in a common domain. For instance, both the concrete IP block descriptions (say VHDL or Verilog), as well as the abstract FSM and SDF models, can be given semantics in a synchronous-clock time domain.

A large body of research exists on synthesizing hardware from dataflow models [1, 5, 10, 13, 14, 15, 19, 35, 37]. These works study problems similar to the glue design problem, however, some of them assume that hardware components for actors are to be synthesized, or at least that they have certain characteristics that match the proposed synthesis techniques. Other works use existing IP blocks coming from a library, but assume that the IP blocks conform to the SDF semantics. As a result, the above approaches suffer from the problems of correctness and defensiveness that are raised in this paper: they may result in glue that is non-optimal or even incorrect.

As mentioned in Section 2.4, the glue design problem is related to controller synthesis [28, 30, 29, 18, 32, 21] but theoretical and practical challenges remain. The same challenges are present in methods for *converter synthesis* [2, 27]. 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 as we have discussed.

Compositional verification techniques such as those in [9, 22] 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. Moreover, issues regarding how exactly to express the requirements of the closed system, mentioned in Section 2.4, apply here as well.

6. CONCLUSION

We have introduced the glue design problem and proposed a methodology for it based on abstract dataflow and FSM models. We have 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.

A key message of this paper is that there is no unique abstract model that serves as a "silver bullet". Instead, different models are good for different purposes. Understanding the limitations of each model is an important aspect of design, and our paper aims to contribute toward that goal.

Future work includes formalization of the relations between abstract and concrete models, development of effective methods to guarantee or check correctness and non-defensiveness, and automatic extraction of abstract models from HDL code. Another area of study is the development of efficient analysis and synthesis algorithms that work directly on the access pattern representations. We would also like to eventually propose that behavioral interface formalisms such as the ones presented in this paper be included in IP interface standards such as IP-XACT (IEEE 1685).

7. REFERENCES

[1] M. Adé, R. Lauwereins, and J. A. Peperstraete. Data Memory Minimisation for Synchronous Data Flow Graphs Emulated on DSP-FPGA Targets. In *In Proceedings of the Design Automation Conference*, pages 64–69. ACM Press, 1996.

[2] P. Bhaduri and S. Ramesh. Synthesis of synchronous interfaces. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, 2006.

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

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

[5] N. Chandrachoodan, S. Bhattacharyya, and K. Liu. The hierarchical timing pair model for multirate DSP applications. *Signal Processing, IEEE Transactions on*, 52(5):1209 – 1217, may 2004.

[6] M. Edwards and P. Green. The implementation of synchronous dataflow graphs using reconfigurable hardware. In *10th Intl. Workshop on Field-Programmable Logic and Applications*, FPL'00, pages 739–748. Springer, 2000.

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

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

[9] T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[10] J. Horstmannshoff and H. Meyr. Optimized System Synthesis of Complex RT Level Building Blocks from Multirate Dataflow Graphs. In *12th International Symposium on System Synthesis*. IEEE, 1999.

[11] C.-J. Hsu, M.-Y. Ko, and S. S. Bhattacharyya. Software Synthesis from the Dataflow Interchange Format. In *Internations Worksop on Software and Compilers for Embedded Processors*, Dallas, Texas, September 2005.

[12] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group. IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows. *IEEE Std 1685-2009*, pages C1 –360, 18 2010.

[13] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs. In *Journal of Signal Processing Systems*, 2009.

[14] H. Jung, K. Lee, and S. Ha. Efficient hardware controller synthesis for synchronous dataflow graph in system level design. In *Proceedings of the 13th international symposium on System synthesis*, ISSS '00, pages 79–84, Washington, DC, USA, 2000. IEEE Computer Society.

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

[16] H. Kee, S. Bhattacharyya, I. Wong, and Y. Rao. FPGA-Based Design and Implementation of the 3GPP-LTE Physical Layer Using Parameterized Synchronous Dataflow Techniques. In *Proc. of the 2010 Intl. Conf. Acoustics, Speech, and Signal Processing*, Dallas, TX, March 2010.

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

[18] H. Lamouchi and J. Thistle. Effective control synthesis for DES under partial observations. In *39th IEEE Conference on Decision and Control*, pages 22–28, 2000.

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

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

[21] Y. Lustig and M. Vardi. Synthesis from component libraries. In *12th Intl. Conf. on Foundations of Software Science and Computational Structures*, FOSSACS '09, pages 395–409. Springer, 2009.

[22] K. McMillan. A compositional rule for hardware design refinement. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*. Springer, 1997.

[23] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

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

[25] National Instruments Corp. LabVIEW FPGA 2010. www.ni.com/fpga.

[26] Open Cores for FPGA and ASIC Development. www.opencores.org.

[27] R. Passerone, L. de Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the International Conference on Computer-Aided Design*, November 2002.

[28] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM Symp. POPL*, 1989.

[29] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science*, pages 746–757, 1990.

[30] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, Jan. 1989.

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

[32] S. Tripakis. Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters*, 90(1):21–28, Apr. 2004.

[33] M. Wiggers, M. Bekooij, and G. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *DAC'07*, pages 658–663. ACM, 2007.

[34] M. Wiggers, M. Bekooij, and G. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *RTAS'08*, pages 183–194. IEEE, 2008.

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

[36] Xilinx Inc. *Xilinx Core Generator*. Xilinx Inc., ISE Design Suite 12.1 edition, 2010.

[37] P. Zepter, T. Grötker, and H. Meyr. Digital receiver design using VHDL generation from data flow graphs. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 228–233, New York, NY, USA, 1995. ACM.