# Codesign of Reactor-Oriented Hardware and Software for Cyber-Physical Systems

ERLING RENNEMO JELLUM, Norwegian University of Science and Technology, Trondheim, Norway
MARTIN SCHOEBERL, Technical University of Denmark, Lyngby, Denmark
EDWARD ASHFORD LEE, University of California, Berkeley, CA, USA
MILICA ORLANDIC, Norwegian University of Science and Technology, Trondheim, Norway

Modern cyber-physical systems often make use of heterogeneous systems-on-chip with reconfigurable logic to provide adequate computing power and flexible I/O. However, modeling, verifying, and implementing the computations spanning CPUs and reconfigurable logic are still challenging. The hardware and software components are often designed by different teams and at different levels of abstraction, making it hard to reason about the resulting computation. We propose to lift both hardware and software design to the same level of abstraction by using the Lingua Franca coordination language. Lingua Franca is based on a sparse synchronous model that allows modeling concurrency and timing while keeping a sequential model for the actual computation. We define hardware reactors as a subset of the reactor model of computation underlying Lingua Franca. We also present and evaluate reactor-chisel, a hardware runtime implementing the semantics of hardware reactors, and an extension to the Lingua Franca compiler enabling reactor-oriented hardware–software codesign.

CCS Concepts: • **Hardware → Hardware accelerators**; • **Computing methodologies → Concurrent programming languages**; • **Computer systems organization → Embedded software**;

Additional Key Words and Phrases: HW/SW Codesign, FPGA, models-of-computation

**ACM Reference format:**
Erling Rennemo Jellum, Martin Schoeberl, Edward Ashford Lee, and Milica Orlandic. 2024. Codesign of Reactor-Oriented Hardware and Software for Cyber-Physical Systems. *ACM Trans. Reconfig. Technol. Syst.* 17, 4, Article 55 (November 2024), 30 pages.
https://doi.org/10.1145/3672083

## 1 Introduction

With the release of the Zynq-7000 in 2011 by Xilinx and Altera's Cyclone V in 2012, a new field of computing was established. These so-called **System-on-Chip Field Programmable Gate Arrays**

This work was supported by the Research Council of Norway (RCN), Multi-Sensor Data Timing, Synchronization and Fusion for Intelligent Robots (Grant number 327538), and the Center of Excellence NTNU AMOS (Grant number 223254).
Authors' Contact Information: Erling Rennemo Jellum (Corresponding author), Norwegian University of Science and Technology, Trondheim, Norway; e-mail: erling.r.jellum@ntnu.no; Martin Schoeberl, Technical University of Denmark, Lyngby, Denmark; e-mail: masca@dtu.do; Edward Ashford Lee, University of California, Berkeley, Berkeley, CA, USA; e-mail: eal@berkeley.edu; Milica Orlandic, Norwegian University of Science and Technology, Trondheim, Norway; e-mail: milica.orlandic@ntnu.no.

ACM 1936-7414/2024/11-ART55
https://doi.org/10.1145/3672083

**(SoC FPGAs)** integrate multiple general-purpose and real-time CPUs, various peripherals, and reconfigurable logic (i.e., an FPGA) on the same silicon die. Installing small CPU cores in the FPGA logic, so-called soft cores, was already common. However, with the SoC FPGAs, the developers can access full-featured application processors capable of running various OS. The flexibility and performance offered by these platforms make them suitable for being at the heart of sensor fusion systems found in embedded systems in Internet of Things, medicine, robotics, and space. Xilinx and Altera went on to be acquired by AMD and Intel, respectively, continuing the legacy of the SoC FPGA.

However, these platforms pose serious challenges concerning how the computations spanning CPUs and FPGA should be modeled, verified, and implemented. At the core of this challenge are the fundamental differences in computational substrate offered by the CPUs and the FPGA. The CPUs offer an interface to the programmer (or compilers) called the **Instruction Set Architecture (ISA)**, which consists of a set of registers and instructions and a deterministic model describing how the state of the machine evolves when executing any sequence of instructions. The most common programming languages used to specify the behavior of these CPUs, like C, C++, and Python, are built on a sequential and imperative model [19]. This is convenient because the ISA is a sequential model, making the compilation process more straightforward.

The FPGA, on the other hand, offers an interface based on synchronous digital logic. The computation is expressed in terms of a set of memory elements whose state changes are synchronized by a clock signal. Although the most common languages for modeling synchronous logic, like Verilog and VHDL, are based on discrete-event models, the usage of the languages to synthesize **hardware (HW)** is synchronous [9]. This synchronous usage is also reflected by a newer HW language, Chisel [6], which even hides the clock from the developer.

A circuit description based on synchronous digital logic is not the lowest level abstraction for the FPGA. Such a description will pass through elaboration, synthesis, and place and route to generate a *routed netlist* before finally a *bitstream* is created. The bitstream contains the complete configuration of all the logic cells, switch boxes, and memories. However, the bitstream format is typically proprietary and the netlist description is rarely written by programmers directly.

There are three core differences between the algorithmic approach offered by the CPU and the synchronous approach offered by the FPGA:

(1) In the sequential model, there is a notion of order but not of time [29]. There is no portable way of specifying, e.g., in C or Python, that a certain instruction should be executed at an exact time in the future. Time is merely a side effect of a particular implementation. This is also matched by fundamental non-determinism in the timing of the CPUs themselves [22]. This makes ordinary CPUs and the sequential model a poor fit for solving timing critical problems like sensor synchronization [21]. In the synchronous model, on the other hand, all computation occurs (conceptually) at an exact timestamp drawn from an external clock. If this external clock is periodic and the synchronous hypothesis [10] is valid, this timestamp can be given a physical interpretation.

(2) The sequential model also lacks a notion of concurrency. For example, in C, there is no way to specify that two different functions should be executed concurrently. One has to execute before the other. The standard way of introducing concurrency is through threads; however, these have been shown to be an unsatisfactory abstraction for concurrency [28]. In the synchronous model, all computations triggered by the same global clock tick are concurrent, except when they have explicit dependencies.

(3) Finally, there is a fundamental difference in how the physical computing device is modeled. Jantsch makes the distinction between *hierarchy* and *abstraction* in models of computation [20]. Hierarchy is defined as the process of hiding information by partitioning a system into smaller parts.

Conversely, abstraction is defined as representing a system in a new way, using new primitives. Sequential imperative programming languages can thus be seen as hierarchical models over the ISA since they do not fundamentally change the way computation is represented. However, the gap from a bitstream to a synchronous program is an abstraction. In this article, we will introduce a **Model of Computation (MoC)** for SoC FPGAs based on abstraction.

A heterogeneous system composed of sequential algorithms on the CPU and an accelerator in the form of synchronous digital logic on the FPGA is hard to model and thus to reason about and verify. In practice, most designers make informal use of models of computation to simplify this. They might, for instance, implement a sequential interface to the FPGA, where the **software (SW)** signals the accelerator to start by writing to a "start" register and polls a "finished" register until the accelerator is done. In this case, we can reason about the accelerator as if it were a C function. If concurrency between SW and HW is desired, the accelerator interface could draw inspiration from the process networks. Either using asynchronous inputs and outputs, similar to a **Kahn Process Network (KPN)** [25], or rendezvous-based inputs and outputs, consistent with communicating sequential processes [18].

We propose, instead, a formal approach where there is a well-defined MoC underlying the language used to specify the computation spanning HW and SW. This merging of model and implementation leads to systems that are "correct-by-construction." We believe that such an MoC should have the following properties:

(a) Deterministic semantics.
(b) Ability to model concurrency at arbitrary granularity.
(c) Ability to model time.
(d) Event-driven with the ability to handle asynchronous inputs.

This article introduces HW reactors, a subset of the reactor MoC [37]. Reactor semantics is defined as the sequence of time-tagged events flowing between concurrent components called reactors. For a well-formed network of reactors, there is a unique legal sequence of such events (a). Reactors are concurrent objects, and the event-triggered reactions, encapsulating computation, have arbitrary granularity (b). The reactor model introduces time as a first-class citizen (c) and allows for limited non-determinism by incorporating asynchronous events (d).

We present `reactor-chisel`, a HW runtime for implementing reactor networks spanning both the CPUs and the reconfigurable logic of an SoC FPGA. We extend the open-source **Lingua Franca (LF)** language [35] with both a Chisel and a Codesign target. This enables reactor-oriented codesign of HW and SW. We evaluate the implementation in terms of resource utilization, maximum clock frequency, latency, and throughput on a set of simple test programs as well as a more realistic image processing example. The results are promising and highlight areas that need improvements and future research.

## 2 Background

### 2.1 MoCs

An MoC is a mathematical *abstraction* of a computing device [19]. The purpose of an abstraction is to hide information and provide a more convenient or economical way of specifying the intended behavior. There are multiple meta-models of MoCs, i.e., different ways of explaining what they are, i.e., their semantics, and how different MoCs can be compared. In the tagged signal [32] and Rugby [20] meta-models, MoCs are described in terms of how they can model concurrency and time. MoCs can also be explained in terms of dataflow and control flow [13]. The dataflow specifies how data are produced, communicated, and consumed, while the control flow deals with when, if, and how different elements process data.

MoCs can prove very useful in designing complex systems as they can allow for mechanized reasoning about the possible behaviors of the resulting system without analyzing the implementation details. MoCs are essential for verification, where typically, a system specification is written in some specification language with a well-defined underlying MoC. This kind of methodology suffers from what is known as "round-trip engineering" [44]. There are now at least two artifacts representing the system, the specification, and the implementation, and keeping these artifacts synchronized becomes a challenge. Another approach followed in this work is to create design languages based on well-defined MoCs. This elegantly avoids the round-trip problem because the specification and the implementation are the same program.

### 2.2 Synchronous Dataflow (SDF)

SDF belongs to the family of dataflow models of computation [14, 31]. Concurrent functional units called *actors* are connected by FIFO channels. The unit of communication in dataflow models is the *token* which is read from and written to the channels by the actors. Actors perform computation in atomic quanta called firings, where a number of tokens are *consumed* from the input channels and *produced* to the output channels. Since the firing is atomic, it is not enabled until all the tokens to be consumed are present at the input channels. SDF is a static dataflow model where the consumption and production rates are fixed and cannot change at runtime, for instance, based on the data carried by the tokens. An SDF graph can be represented as a set of balance equations, one for each channel. A non-zero solution to the balance equations is called a repetition vector and denotes a set of actor firings such that all actors fire and the SDF graph returns to its initial state. An SDF graph is *consistent* [30] if such a repetition vector exists. A consistent SDF graph can be executed indefinitely with bounded memory. The simplest form of SDF is the homogeneous SDF, where each actor consumes a single token from each input channel and produces a single token to each output channel. SDF graphs are *deterministic* in the sense that the order in which a set of actor firings is performed does not affect the final state of the graph.

SDF models are suitable for HW–SW codesign as they map easily both to SW and HW. SDF models are often used to model and implement signal-processing applications where the data rates are static and predictable. However, modern cyber-physical systems are often reactive, meaning they include dynamic, unpredictable, and asynchronous inputs. This is hard to model using SDF.

### 2.3 LF and the Reactor MoC

For such reactive systems, discrete-event MoCs like the reactor model [38] might be more appropriate. Reactors were formalized as part of Marten Lohstroh's PhD thesis [34]. LF [35] is a polyglot coordination language based on reactor semantics [36]. To informally introduce the reactor model, we will use the simple example LF program represented textually and graphically in Figure 1.

The program in Figure 1 consists of three reactors related by connections and containment. At the top of the containment hierarchy is the main reactor, which in this example contains two reactor instances, one `Source` reactor and one `Sink` reactor. Such *contained* reactors are referred to as child reactors, and the *containing* reactor as the parent reactor. A reactor is a stateful concurrent component represented as a rectangle with rounded corners in the diagrams. A reactor communicates with other reactors through typed *ports*; in our example, the `Source` reactor has an output port `out`, which is connected to an input port `in` at `Sink`. The fundamental unit of computation is the *reaction*. These are represented as dark grey chevrons and can be thought of as *event* handlers. An event is the unit of communication in the reactor model. It consists of a value and a timestamp, referred to as tags in the reactor model. Like in the synchronous model, a reaction invocation is logically instantaneous. Events originate from triggers, which include fixed period *timers*, represented in the diagrams as clocks, input ports, represented as small triangles on

```
target C;
reactor Source {
    timer t(0, 100 msec)
    logical action a
    physical action p
    output out: int

    reaction(t) -> out {=
        // User C code here
    =}
    reaction(a) -> out {= =}
    reaction(p) -> a, out {= =}
}
```

```
reactor Sink {
    timer t(0, 50 msec)
    input in: int
    state s: int
    reaction(t) {= =}
    reaction(in) {= =}
}
main reactor {
    source = new Source()
    sink = new Sink()
    source.out -> sink.in
}
```

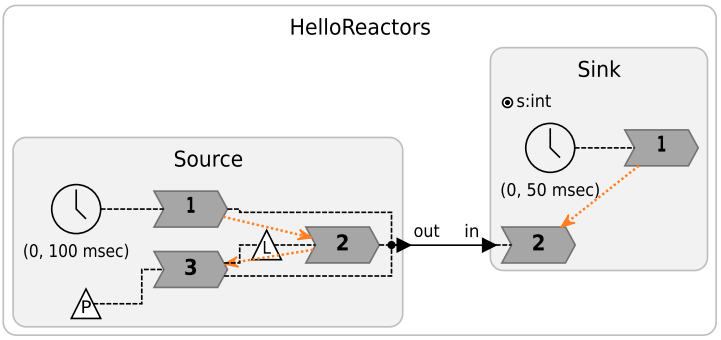



Fig. 1. An example LF program using the C target.

the borders of the reactor, and *actions*, represented as white triangles. Logical actions are denoted with an *L* and allow reactions to schedule future events at well-defined timestamps in the future. Physical actions, denoted with a *P*, allow asynchronous contexts, like an interrupt service routine, to schedule events into the system. Reactions may also produce events to output ports or schedule future events via logical actions. The actual computation taking place in a reaction is specified in the reaction body, which is expressed in a target language between two brackets {=. . . =}.

In SDF, actors are only fired when sufficient tokens are on the input channels. In the reactor model, reactions are only triggered when all observable events with the same tag have been produced. Reactions inside the same reactor can share state variables and are executed in a predefined order should they be triggered at the same tag. The reactor model is deterministic, assuming that the reaction bodies are deterministic.

The reactor model uses a superdense model of time and includes two timelines. The *logical* timeline can be compared to the simulated timeline of a VHDL circuit. The reactor model delivers deterministic computations relative to this logical timeline. However, the programs also exist on a real, *physical* timeline. There are four points of interaction between the physical and the logical timeline. (1) At startup, physical and logical time is synchronized. (2) Logical time can never exceed physical time. (3) Physical actions are scheduled relative to the current physical time, unlike logical actions, which are scheduled relative to the current logical time. (4) A *deadline* can be associated with a reaction. A deadline specifies the maximum difference between logical and physical time, i.e., the *lag*, allowed at the invocation of the reaction. The user must also specify a deadline miss handler. The ability to model the composition of static and dynamic dataflow (through timers and logical actions) with asynchronous events (physical actions) and real-time constraints (deadlines)

makes reactors an ideal model for cyber-physical systems. Bringing the key aspects of timing and concurrency into the semantics of the program also enables reusability and modularity of designs.

The reactor model also lends itself well to distributed execution. Distributed programs can coordinated centrally, with a scheme based on high-level architecture or decentralized with a scheme based on Ptides [7]. In recent work, LF has been extended with support for modal behaviors [42]. It also has support for running on bare-metal microcontrollers and can function as a real time operating system [23]. LF has been shown to provide high performance, comparable to other actor-oriented frameworks while introducing determinism [39].

## 3 Related Work

There have been multiple proposals for lifting HW and SW design to the same level of abstraction. In this section, we review a selection of notable related works.

### 3.1 Open Computing Language (OpenCL)

OpenCL [40] by the Khronos Group is a vendor-neutral framework for writing programs that execute on heterogeneous platforms, such as SoC FPGAs. It provides a uniform programming model for CPUs, GPUs, and FPGAs based on the C programming language. In the context of SoC FPGAs, OpenCL provides a convenient abstraction for HW–SW codesign, because both HW and SW are expressed in the same programming model; this makes it easier to experiment with the partitioning between HW and SW. The OpenCL compiler generates the boilerplate HW and SW facilitating the communication. However, the programming model is very general and requires explicit synchronization of data across the various memories used by different OpenCL kernels. This allows for non-deterministic programs. OpenCL is a **High-Level Synthesis (HLS)** framework. HLS automatically generates HW descriptions from high-level programming languages, such as C, OpenCL, or MATLAB. While HLS can reduce design time and increase productivity, design at the register-transfer level offers better fine-grained control and more optimization opportunities for the generated HW design. Unlike reactor-chisel, HLS is based on sequential languages and cannot specify timing and concurrency [15].

### 3.2 SYCL

SYCL [1] is a more recent, higher-level programming model built on top of OpenCL. While OpenCL is based on C and requires that each kernel be written in a separate file, SYCL uses C++ and allows a single-source model. SYCL also abstracts away much of the low-level device management and memory handling required in OpenCL. However, it inherits OpenCL's limitations with respect to determinism, timing, and concurrency.

### 3.3 Compaan/Laura

In Compaan/Laura [43], a computation is modeled in a MATLAB framework built on the KPNs MoC. A KPN consists of a network of processes connected by FIFO channels. The Compaan compiler will partition the KPN into a SW part and a HW part and the Laura compiler generates a VHDL description of the HW processes and a runtime implementing the control flow. The SW processes are implemented in C++. KPNs are closely related to the dataflow models that the HW reactor model is based on. However, reactors are more expressive than KPNs. A reactor can have encapsulated data-dependent production of output events. Through the runtime infrastructure, downstream reactors will be stalled until output events are produced, or guaranteed not to occur. Using KPNs, the data-dependent production logic must also be replicated in the downstream reactors so they can know whether the upstream will produce outputs or not.

### 3.4 ReconOS

ReconOS [4] extends the POSIX programming model into the FPGA domain. It makes it possible to design HW threads executing in the FPGA logic communicating with SW threads or other HW threads using inter-process communication primitives like FIFOs and pipes. This significantly lowers the barrier of entry for programmers wanting to develop applications on SoC FPGAs. The key difference between reactor-chisel and ReconOS is the underlying MoC. ReconOS is based on shared-memory concurrency which is a fundamentally non-deterministic MoC. `Reactor-chisel` is based on timed and deterministic message-passing instead.

### 3.5 ReconROS

ReconROS [33] is another framework addressing the desire for a unified way of designing HW and SW for robotics applications. This is achieved by integrating **Robot OS 2 (ROS2)** into their existing ReconOS project. ROS2 is a middleware for robotics applications centered around a publish-subscribe protocol. While an improvement over shared memory concurrency, publish-subscribe is also fundamentally non-deterministic [8].

### 3.6 ForSyDe

ForSyDe was proposed in the Ph.D. thesis of Ingo Sander in 2003 [41]. Its core idea is to raise the abstraction level of SW design based on several models of computations. The designer specifies the intended behavior of their system as a hierarchical network of processes communication with tagged signals. Each process is a function mapping process inputs to process outputs. A core tenant in ForSyDe is the so-called separation of concerns, where the concerns are computation and communication. The computation is expressed as functions and the communication is defined by an MoC. The ForSyDe library supports multiple MoCs like the synchronous MoC, KPN, and SDF. Our approach focuses on a single MoC and takes a different approach to the idea of a "separation of concerns." With LF, we use different language for expressing the concerns of communication and coordination, from the language used for expressing the computation.

### 3.7 Vitis Unified SW Platform

Vitis is a unified SW platform for HW–SW codesign on AMD/Xilinx devices [2]. Vitis bundles a series of tools that previously were distributed individually like Vivado, Vivado HLS, SDSoC, and Petalinux tools. Vitis provides a SW-centric workflow where HW designs can be compiled and linked with the Vitis v++ compiler. Vitis supports both HDL and HLS for describing the FPGA designs and C and C++ for the SW components. Vitis also supports OpenCL. The communication and synchronization between HW and SW are abstracted by the Xilinx Runtime Library which provides a HW-independent API to the accelerators. Compared to LF and HW reactors, Vitis can be viewed as an *ad hoc* approach. It offers multiple paradigms for HW–SW interaction but leaves it up to the developer to combine them in a predictable way.

### 3.8 Architecture Analysis and Design Language (AADL)

AADL is a modeling language for safety-critical embedded systems. AADL differs from the other frameworks discussed because it models both the computation and the execution platform. AADL also lets you specify nonfunctional requirements like execution time or memory footprint. An AADL model is a hierarchy of components that has both a textual and graphical representation. A component models either a SW or an execution platform entity. The SW components are data, threads, thread groups, subprograms, and processes. The execution platform components are memory, buses, processors, devices, virtual processors, and virtual buses. Inheritance enables

multiple implementations of the same component. Components may be organized into packages, which enables reusability. AADL is not built on a single concurrent MoC but allows the designer to implement a mixture of a shared variable MoC and a dataflow process network MoC. There are multiple tools, such as OSATE and RAMSES [12], which provide refinements and SW synthesis from AADL models. Kaolin is a development process and tool for mapping computation expressed in AADL onto FPGAs [11]. As with ForSyDe, AADL offers a more heterogeneous approach, while we focus on delivering the semantics of a single deterministic MoC.

### 3.9 SystemC

SystemC [3] is a system modeling language based on C++. It supports modeling both HW and SW and their interaction at multiple levels of abstraction. Like LF and the reactor model, SystemC is also based on a discrete-event model. However, SystemC is mainly targeted at HW–SW co-simulation, not actual implementation.

### 3.10 Bluespec Codesign Language (BCL)

The BCL was developed as part of Myron King's Ph.D. thesis [26]. It is a language based on the guarded atomic actions model, which can be compiled into both HW and SW. A BCL program consists of explicitly declared state variables and a set of guarded atomic actions on these state variables. Compilation of such programs into an efficient finite state machine is performed using the Bluespec compiler [5]. BCL introduces the ability to generate efficient C++ modules implementing the computation specified with guarded atomic actions. Moreover, a program can be arbitrarily partitioned between HW and SW, and the compiler will generate logic for communication and synchronization. A key difference between BCL and the `reactor-chisel` is that guarded atomic actions are a non-deterministic MoC. Atomic actions enabled simultaneously can be applied in any order. This is argued to be an advantage because it gives the compiler more freedom. Additionally, while BCL introduces a new language for expressing computation, LF only introduces a new syntax for coordination. The actual computation is expressed in a language already familiar to the user.

### 3.11 LEAP

LEAP is an FPGA runtime, aimed at making FPGA development easier and more familiar for SW developers [16]. Much like an OS, LEAP provides basic device abstractions, input/output, and memory management services, hiding the complex details of the underlying FPGA HW from the designer. LEAP consists of a set of Bluespec System Verilog libraries and a compilation toolchain. Our work differs from LEAP in two regards. First, LEAP is targeted at standalone FPGA designs and not HW/SW-codesign. Second, LEAP is based on the latency-insensitive paradigm which admits nondeterminism.

## 4 Reactor-Oriented HW–SW Codesign

In this section, we dive right into a set of programs exemplifying the capabilities of the proposed Chisel-target of LF. The underlying MoC is introduced later, in Section 5, and the details of the implementation of `reactor-chisel` in Section 6.

### 4.1 Blinky

The Hello World of embedded systems is the Blinky program. Figure 2 shows a Blinky program, written in the Chisel-target of LF. The Blinky reactor has an output port called `led` with datatype `Bool()`. It is marked "external," meaning that it is just routed directly to the top level of the design. A timer named `t` with a 500-millisecond period is defined as well as a state variable `count`. The reactor contains a single reaction triggered by `t`. The reaction functionality is defined by

```
target Chisel
reactor Blinky {
  @external
  output led: {=Bool()=}
  state count: {=UInt(32.W)=}
  timer t(0, 500 msec)
  reaction(t) -> led {=
    lf_set(led, ~lf_get(out))
    lf_write(count, lf_read(count)+1.U)
  =}
}

main reactor {
    blinky = new Blinky()
}
```

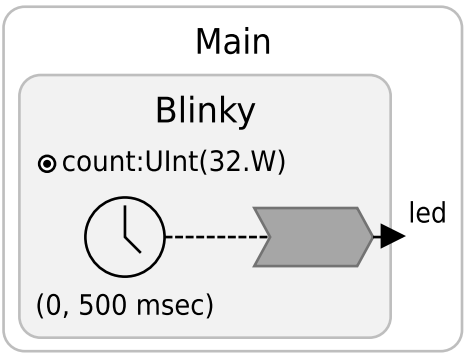



Fig. 2. A simple Blinky example using the Chisel-target.

a reaction body between the curly brackets {=... =}. The reaction body is a sequential circuit written in Chisel, which by default is enabled for a single clock cycle but might execute for an arbitrary number of clock cycles. A simple API is provided for reading values from the input ports, `lf_get(inputPort)`, reading from state variables, `lf_read(stateVar)`, writing to state variables, `lf_write(stateVar, value)`, and setting output port values, `lf_set(outputPort, value)`. There are also API functions for reading the current logical time, `lf_time_logical()`, and the current physical time, `lf_time_physical()`.

This program will be compiled into a Chisel design where a top-level pin is toggled every 500 logical milliseconds. The timers in LF are *logical*, which means that they specify the order and simultaneity of events, not a precise physical time at which the events should occur. Since reactions with data dependencies execute in a deterministic order, and reactions might execute for an arbitrary number of cycles, the exact physical time that a reaction will be enabled is not necessarily deterministic. This subtle distinction between the logical timeline, as specified in LF syntax, and the physical timeline which is dependent on both the HW platform and the sequential circuits embedded in the reactions have implications for the appropriate level for granularity computations should be modeled with reactors.

### 4.2 Vector Addition

Due to HW runtime overhead, we should not expect circuits modeled at clock cycle level granularity, e.g., with reactions triggered each clock cycle, to be realizable in physical time. `Reactor-chisel` is more suitable for coarse-grained computations, where, for instance, an entire accelerator is embedded in a single reactor. Such a reactor, executing on the FPGA, can also be composed with reactors written in other targets without changing the semantics. This makes it ideal for targeting SoC FGPAs. For this purpose, we have added a "Codesign-target" to LF which allows mixing SW reactors written in C++ with HW reactors written in Chisel.

Listing 1 shows such a program spanning both the CPUs and the FPGA of an SoC FPGA. It is the classical VectorAdd example where the vector addition of two long vectors is off-loaded to the FPGA. The reactor called `SW` is to be run on the CPUs and will, every 10 milliseconds, output two 2,048-element arrays to the `Accel` reactor, which is to be run on the FPGA. The `Accel` reactor has a single reaction triggered when there are tokens on both inputs. It reads out all elements from each port, sums them, and writes them to the output port. By setting the `reactionDone` wire, the reaction body can execute for an arbitrary number of cycles.

**Listing 1:** A LF Programming Using the Codesign Target That Allows Mixing C++ and Chisel

```
1  target Codesign
2  reactor Software {
3      output op1: uint32_t[2048]
4      output op2: uint32_t[2048]
5      input res: uint32_t[2048]
6      timer t(0, 10 msec)
7
8      reaction(t) -> op1, op2 {=
9        // Cpp program writing two 2048 word arrays to each output port
10     =}
11     reaction(res) {=
12       // Cpp program using the result
13     =}
14 }
15 reactor Accel {
16     @array(length=2048)
17     input op1:{=UInt(32.W)=}
18     @array(length=2048)
19     input op2:{=UInt(32.W)=}
20     @array(length=2048)
21     output res:{=UInt(32.W)=}
22     reaction(op1, op2) -> res {=
23       val readPort1 = lf_get_array(op1, 0.U, 2048.U)
24       val readPort2 = lf_get_array(op2, 0.U, 2048.U)
25       val writePort = lf_set_array(res, 0.U, 2048.U)
26       val cnt = RegInit(0.U(32.W))
27       val fire = writePort.ready && readPort1.valid && readPort2.valid
28       when(fire) {
29         writePort.valid := true.B
30         readPort1.ready := true.B
31         readPort2.ready := true.B
32         writePort.bits.data := readPort1.bits.data + readPort2.bits.data
33         cnt := cnt + 1.U
34       }
35       reactionDone := cnt === 2048.U // Builtin signal
36     =}
37 }
38 main reactor {
39     sw = new Software()
40     @fpga
41     accel = new Accel()
42     sw.op1, sw.op2, accel.res -> accel.op1, accel.op2, sw.res
43 }
```

This program shows how easy it is to offload computation from the CPU to the FPGA. The `@array` attribute on Lines 15, 17, and 18 informs the compiler that there is an *array* of tokens being communicated. Connections between such ports are buffered in BRAM if it is between HW reactors and off-chip DRAM if it is between a SW reactor and a HW reactor. `Reactor-chisel` includes **Direct Memory Access (DMA)** modules that efficiently handle the interleaving of memory requests and responses for multiple ports. The Codesign target aims to abstract away all the communication and synchronization details between the CPU and the FPGA and leave the developer only with the task of implementing the actual computation going on in the event-triggered reactions.

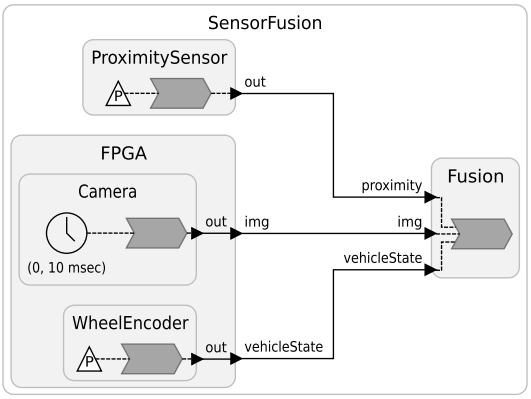



Fig. 3. A sensor fusion application spanning FPGA and CPU.

### 4.3 Multi-Sensor Fusion

The `Accel` HW accelerator in the `VectorAdd` program was purely reactive and had no locally originating tokens. While the Codesign target allows efficient modeling and implementation of such simple codesigns, it is also much more expressive.

Imagine acquiring, preprocessing, and fusing messages from multiple sensors. Figure 3 shows an LF diagram of a program fusing three sensors. A proximity sensor delivers sensor readings asynchronously through an interrupt service routine to the SW, a camera is triggered and processed at 100 Hz from the FPGA as well as a wheel encoder sensor which also triggers asynchronously and is connected to the FPGA. Asynchronous events are modeled in LF through the use of physical actions.

This LF program has a semantic that is quite hard to achieve by using *ad hoc* programming in C++ and VHDL. This program dictates that the `Fusion` reactor will always see sensor messages in their timestamp order. This is a crucial property for many sensor fusion systems. Out-of-order handling of sensor messages can, in the case of sensors measuring continuous quantities, e.g., an accelerometer or tracking of the position of features in an image, lead to serious estimation errors [17]. For sensors measuring discontinuous quantities, like a wheel encoder that indicates whether the vehicle is moving forward or backward and a proximity sensor telling whether an object is close to the vehicle or not, out-of-order handling of messages can lead to the system passing through inconsistent states [7].

Using `reactor-chisel`, control logic is synthesized to the FPGA and the CPU such that they share a common notion of time through the exchange of timestamps. An event arising in the `WheelEncoder` will stall the SW, preventing it from handling any other events with a later timestamp until the `WheelEncoder` event has propagated through the system and has been fully handled. This coordination is bidirectional, and the FPGA cannot handle events until dependent events in the SW reactors are handled. This coordination mechanism will be further elaborated in Section 6.9.

To the authors' knowledge, this kind of tight bi-directional coordination between computation in SW and HW is novel and unlocks a new way of doing HW–SW codesign.

## 5 HW Reactors MoC

In the same way that VHDL and Verilog define synthesizable subsets, we propose a synthesizable subset of Lohstroh's original reactor model [37]. This subset lends itself well to a dataflow-like implementation. In this section, we will give an informal description of this subset. When there

might be ambiguity, we will refer to the components of the HW reactor model as *HW* reactors and *HW* reactions; if there is no ambiguity, we will refer to them as reactors and reactions.

HW reactors are similar to ordinary reactors. They encapsulate timers, physical actions, state variables, ports, and event-triggered HW reactions, as well as contained HW reactors. The ports of HW reactors can be connected through connections yielding a directed graph of HW reactors which constitutes a program.

The fundamental unit of communication in the HW reactor model is a timestamped *token*. Although similar to the events of the reactor model, we refer to them as tokens since this is the terminology used in dataflow models and the HW reactor subset is arguably closer to dataflow than discrete-event.

HW reactions are the fundamental components in our model, and they resemble actors in homogenous SDF. In the reactor model, reactions are triggered by events from *any* trigger; on the other hand, HW reactions are triggered by the presence of a single token on *all* triggers and sources. In the reactor model, a reaction firing *may* produce an event to any output port marked as an effect. In the HW reactor model, it produces a token to *all* output ports. This requires us to add the notion of an absent token. If a HW reaction does not produce a token to a certain output port, then the runtime produces an absent token there.

Input ports "fork" tokens to all dependent HW reactions. In the same way, output ports "join" tokens from all HW reactions that have the port as an anti-dependency. Connections buffer tokens between ports.

There are four ultimate sources of tokens: timers, built-in triggers like startup and shutdown, physical actions, and ports at the top-level HW reactor, which is connected to external reactors, e.g., running in SW. Every time one of the sources generates a token, all other sources must also generate a token, possibly an absent token. This means that each token source will output a sequence of tokens with identical tags. For timers, this sequence can be computed as a static schedule of a repeated hyperperiod. For physical actions and inputs from external reactors, the absence or presence of tokens must be checked at each step in the hyperperiod. Tokens from physical actions and external reactors might also arrive at tags that are not present in the static schedule, in which case all the timers will fire with an absent token. Physical actions are restricted to only scheduling tokens without any additional offset. Physical actions can thus be queued in an FIFO queue, which is cheaper in HW than a sorted priority queue.

There is a subtle distinction between the logical timeline, defined by the tags of the tokens flowing between HW reactors, and the physical timeline, defined by the clock signal driving the FPGA. The tags of the tokens can also be understood as *reaction indices*. They denote the order in which the reactions should be executed. A runtime should try to minimize the difference between the logical and the physical timeline; however, the semantics of the program are defined relative to the logical timeline, not the physical.

Lohstroh's original reactor model also includes logical actions, which are not included in the HW reactor model. Logical actions are very expressive and hard to capture in static dataflow. Suppose a reaction declares a logical action as an effect. In that case, the reaction may schedule an arbitrary number of future events with timestamps only limited by a possible minimum spacing parameter. This means that any LF program with a logical action might need an unbounded event queue. Logical actions can be thought of as equivalent to the `after` statement in VHDL which is not part of the synthesizable subset for the very same reason.

To summarize, our underlying model is close to homogenous SDF. A computation is thought of as a graph of reactions fired when there are tokens at all inputs. The computation is driven forward by the token sources generating sequences of tokens and absent tokens. At every tag where a token is present, all reactions in the program will be fired, possibly with absent tokens.

## 6 Reactor-Chisel

In the following section, we will explain how such a model can be implemented efficiently on an SoC FPGA. We introduce `reactor-chisel` which is a HW runtime, meant for executing on the FPGA, implementing the semantics of the HW reactors model and enabling the correct execution of the programs presented in Section 4. We refer to `reactor-chisel` as a "runtime" because in function it resembles a runtime underlying languages, such as Java, Python, and Golang. However, due to the fundamental difference between FPGAs and CPUs, the implementation is fundamentally different. `Reactor-chisel` is organized as "glue logic" wrapping user-written modules. This glue logic is mainly concerned with abstracting away the communication between different modules. To ensure the deterministic composition of modules, a centralized scheduler module is also part of the runtime which distributes tokens to the user-written modules. The tokens act as a virtual clock signal, enabling them in a pre-defined order. Through **Control and Status Registers (CSRs)**, the scheduler module also coordinates with a SW runtime ensuring well-defined ordering of execution of SW and HW modules.

We will use the example program in Listing 2 represented with a diagram in Figure 4 as a running example. This program contains most of the LF primitives currently supported by `reactor-chisel`. When passing such a program to the LF Compiler (`lfc`), a HW design using `reactor-chisel` as a library will be generated. In the following, we will introduce the different parts of this generated design.

### 6.1 Tokens

Tokens are the fundamental unit of communication in a HW reactor program. All connections, dependencies, and anti-dependencies, indicated by solid and dashed lines in Figure 4, will be compiled to interfaces that exchange tokens.

Figure 5 shows the class hierarchy of tokens in `reactor-chisel`. All tokens include a tag (i.e., a timestamp representing the logical time of the token) and a bit indicating whether this is an absent token or not. The default in `reactor-chisel` is to use a 32-bit signed integer in nanosecond resolution to represent time, this will not cause any issues unless the generated schedule has a hyperperiod that spans more than 2 seconds (more on hyperperiods later). The tag will be optimized away unless the user-written modules are explicitly reading the tag. In Figure 5, there are three types of tokens. Pure tokens carry no values and are used for triggers from timers, physical actions with no type, and precedence ports between the reactions. Single tokens are used for tokens carrying a single value of arbitrary Chisel type. Single tokens should only be used with narrow data types since they will be compiled into interfaces with bit widths sufficient to carry the entire data type. For bigger data types, such as an image frame, array tokens are more appropriate. Array tokens encapsulate arrays of values that are buffered in RAMs between reactors. This introduces additional read latency which depends on the type of RAM (either BRAM or DRAM).

### 6.2 Token Interfaces

Tokens are communicated between reactions contained in various reactors. Figure 6 shows the four *channels* used for reading and writing tokens and the four different *interfaces* we compose out of these channels. A channel is a uni-directed group of signals and an interface is a group of such channels. All channels, and thus also interfaces, are parameterized by the token type. `TokenReadMaster` and `TokenReadSlave` are the interfaces used to read, i.e., receive, tokens. They are identical except for the direction of the channels. A `TokenReadReq` channel is used to communicate a read request from the master to the slave. If the token is an `ArrayToken`, the request includes an address and a read size. In the case of a `PureToken` or a `SingleToken`, this channel is unused. A `TokenReadResp` channel is used by the slave to respond to requests. In the case of `SingleTokens` and `PureTokens`,

**Listing 2:** An Example Program to Illustrate the Different LF Primitives Supported By `ReactorChisel`

```
target Chisel

reactor Source {
  timer t(0, 1 usec)
  output outSingle: {=UInt(8.W)=}
  @array(length=8)
  output outArray: {=UInt(8.W)=}
  @external
  input inExt: {=UInt(1.W)=}
  state cnt: {=UInt(8.W)=}
  reaction(t) -> outSingle {=
    // User-written Chisel code here
  =}
  reaction(startup) inExt -> outArray {=
    // User-written Chisel code here
  =}
}
reactor Sink {
  @array(length=8)
  input inArray: {=UInt(8.W)=}
  input inSingle: {=UInt(8.W)=}
  @external
  output extOut: {=UInt(1.W)=}
  output out : {=UInt(32.W)=}
  physical action phy: {=UInt(8.W)=}
  reaction(inArray) {=
    // User-written Chisel code here
  =}
  reaction(phy, inArray) -> out {=
    // User-written Chisel code here
  =}
  reaction(shutdown) -> extOut {=
    // User-written Chisel code here
  =}
  contained = new Contained()
  inArray -> contained.inArray
}
reactor Contained {
  @array(length=8)
  input inArray: {=UInt(8.W)=}
  reaction(inArray) {=
    // User-written Chisel code here
  =}
}
main reactor {
  src = new Source()
  sink = new Sink()
  src.outSingle, src.outArray -> sink.inSingle, sink.inArray
}
```

the slave will present any available token on the `TokenReadResp` channel, without the need for a prior request. `TokenWriteMaster` and `TokenWriteSlave` are the interfaces used to write, i.e., send, tokens. A `TokenWriteReq` channel is used by the master to request sending a token. It includes an address and a write size for `ArrayTokens`, whereas `SingleToken` and `PureToken` do not use this channel. A `TokenWriteDat` channel is used for transmitting the actual data within the token. All channels handshake with ready/valid signals. In addition to the channels, there is also a fire signal.

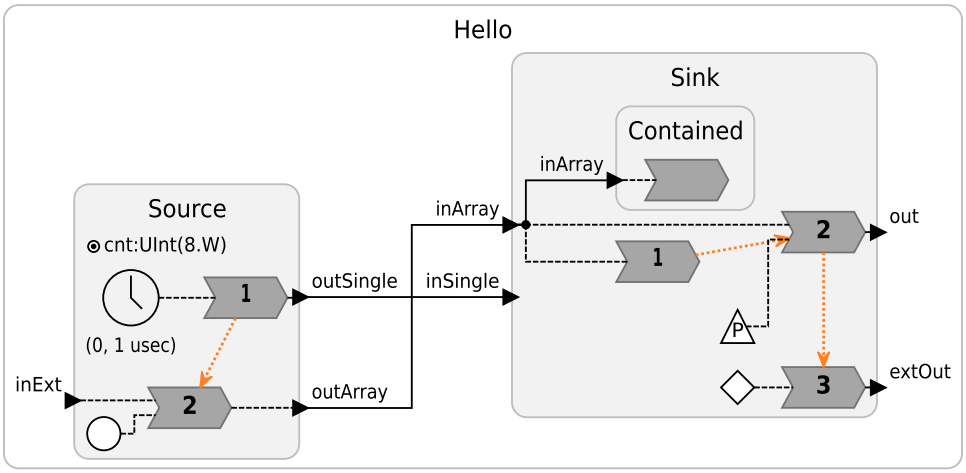



Fig. 4. The auto-generated diagram of the program in Listing 2.

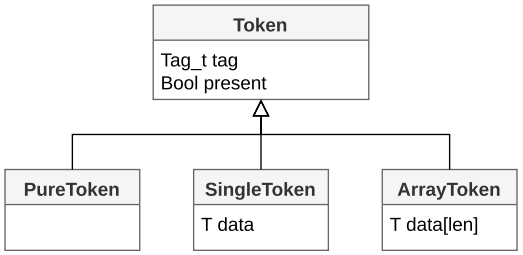
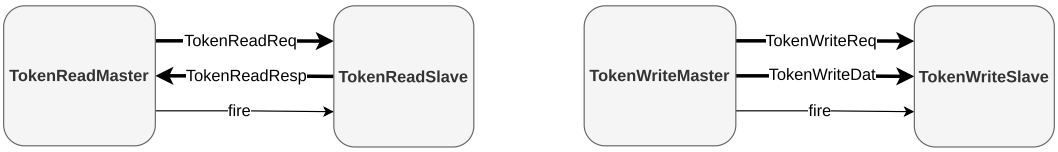



Fig. 5. The Chisel class hierarchy of tokens.




(a) Token read channels (b) Token write channels

Fig. 6. The four channels used to read and write tokens. We refer to `TokenReadMaster`, `TokenReadSlave`, `TokenWriteMaster`, and `TokenWriteSlave` as *interfaces* which we display with rounded rectangles.

It is used by the `TokenReadMaster` to signal that the firing of any connected component, typically a reaction, has been completed and that the tokens have been consumed. The fire signal is also used by the `TokenWriteMaster` to signal that the writer has finished writing the tokens.

### 6.3 Reaction

Reactions are the fundamental unit of computation. In the HW reactor model, a computation is thought of as a directed graph of reactions. The programmer's task is mainly that of designing the computations encapsulated by each reaction. These computations are also referred to as reaction-bodies and are expressed in a target language between the {=...=} brackets in the LF programs, e.g., at Line 12 in Listing 2. `Reactor-chisel` currently only supports reaction bodies written in

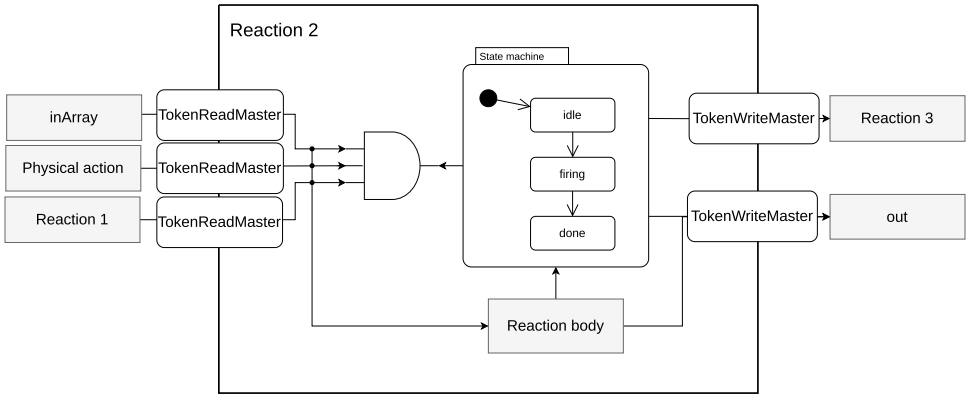



Fig. 7. The Chisel module implementing Reaction 2 in the Sink reactor of Figure 4.

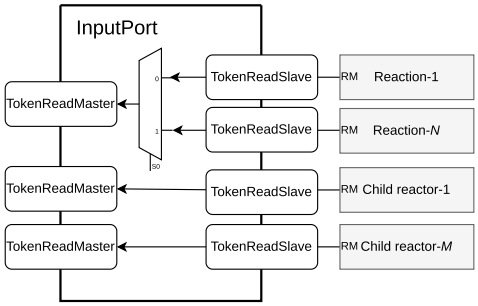
Chisel, but any HW design language that compiles to Verilog, including various HLS languages, could be supported.

In `reactor-chisel`, a `Reaction` is a parameterized HW module which, among other things, encapsulates the user-written reaction body. A `Reaction` has a `TokenReadMaster` interfaces to each of its dependencies and a `TokenWriteMaster` for each anti-dependency. Possible dependencies are triggers (like timers, physical actions, and startup/shutdown), input ports of the parent reactor as well and precedence ports from other reactions within the same reactor with higher precedence. The possible anti-dependencies are the output ports of the parent reactor and precedence ports to reactions with lower precedence. The core feature of a `Reaction` is a finite state machine that controls when the user-written reaction body shall be enabled.

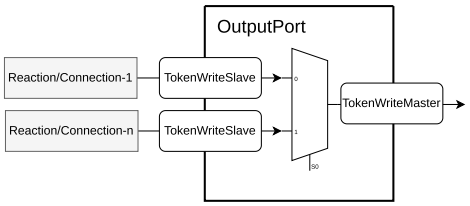
Figure 7 shows Reaction 2 from the `Sink` reactor in Figure 4. Notice that this reaction has three dependencies, each connected through a `TokenReadMaster` interface. It is the port called `inArray`, a physical action as well as a precedence port from Reaction 1. The `Reaction body`, indicated in the figure, is a user-written Chisel module and is copied directly from the LF program. Everything else is part of the runtime and code-generated. In Listing 2, all reaction bodies are empty. A state machine controls the execution. Whenever there are tokens present on all the input channels, and all the output channels are ready to accept tokens, the reaction will enter a state where the reaction body is enabled. It will execute until a special signal, called `reactionDone`, which defaults to "true" but can be overwritten from the reaction body, is asserted, at which point we will move to the "done" state. In the "done" state, all output channels that have not received any token receive an absent token.

### 6.4 Ports

Ports are modules that enable reactions to communicate tokens. In `reactor-chisel`, each port is implemented by a set of `TokenReadMaster` or `TokenWriteMaster` interfaces, depending on whether it is an input or output port. An `InputPort` allows multiple reactions to read through a single `TokenReadMaster`, conversley, an `OutputPort` does the same for outputs. Contained reactors can also have their input ports connected to an input port of the parent, as is the case with the `Contained` reactor in Figure 4. This contained reactor will also be connected to the `InputPort`. The `InputPort` works like an arbiter allowing multiple reactions within the same reactor sharing a single `TokenReadMaster` interface. Reactions at different levels of the containment hierarchy will get parallel `TokenReadMaster` interfaces. Figure 8(a) shows a generic `InputPort` module connecting $N$ reactions from the same reactor and $M$ contained reactors to the input port of the




(a) An InputPort multiplexes the read signals from the reactions within the same reactor and also forwarding read signals from reactions in child reactors.




(b) An OutputPort multiplexing the write signals from reactions and connections.

Fig. 8. InputPorts and OutputPorts are arbiters connecting reactions and contained reactors to the input and output ports of the parent reactor.

parent reactor. This results in $M + 1$ parallel "read channels" through the input port. The number of read channels associated with an input port can only be decided by traversing down the containment hierarchy and finding all reactions connected to the input port in question. Figure 8(b) shows a generic `OutputPort` module. It multiplexes the `TokenWriteMaster` interfaces from reactions and connections. There is never more than a single "write channel." This is due to the reactor semantics that do not allow reactions at different levels of the containment hierarchy to connect to the same output port.

### 6.5 Connections

In `reactor-chisel`, a `Connection` is a memory, typically situated between an output port of one reactor and the input port of another. A `Connection` will expose a single `TokenWriteSlave`, typically connected to an output port, and several `TokenReadSlave` matching the number of read channels of the connected input port. Figure 9 shows a `Connection` connecting to two reactors with a generic memory for storing tokens. `Connections` are parameterized by the type of token it carries. `SingleTokens` are stored in flip-flops, while `ArrayTokens` uses BRAMs. A `Connection` can only have a single writer, but multiple readers. This is mirrored by the fact that input ports of reactors might have multiple read channels, while output ports only can have a single write channel. In Figure 9, two duplicate memories are written to by an output port, while being read from, independently, by two input ports, a simple state machine transitions on the fire signals on the `TokenWriteSlave` and `TokenReadSlaves`.

### 6.6 State Variables

In LF, state variables are persistent data associated with a reactor. They retain their value between reaction invocations, and they can be read and written from all reactions in the reactor. This is unlike any sequential elements instantiated in the reaction bodies, which will be reset between each reaction invocation. In `reactor-chisel`, state variables are implemented as memories with read and write ports connected to all the reactions within the same reactor. Due to the mutual exclusion between these reactions, all of these ports can be multiplexed onto a single read and a single write port connected to the actual memory. In our current implementation, state variables are stored in flip-flops.

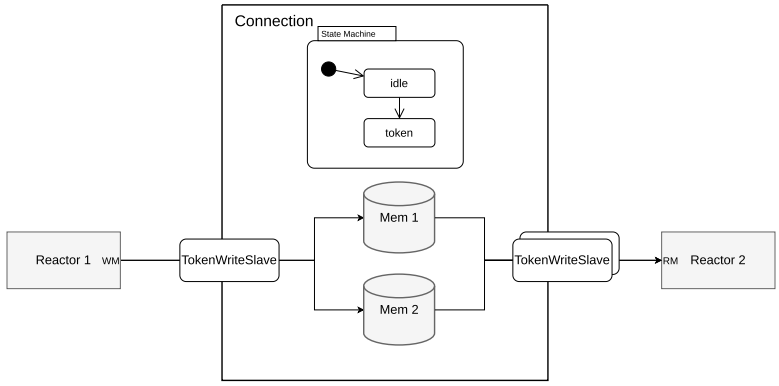



Fig. 9. A Connection is written to by the output port of a reactor and read from by the input port of another reactor. RM and WM are shorthand for TokenReadMaster and TokenWriteMaster.

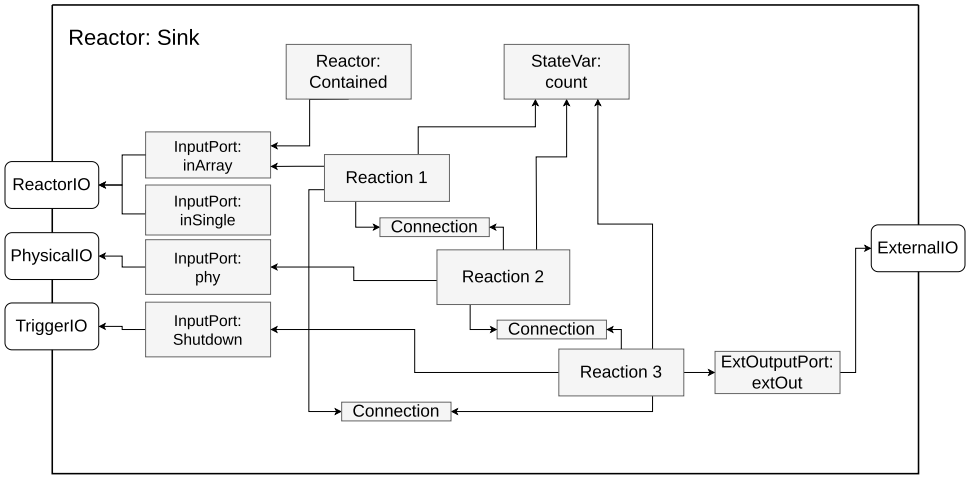



Fig. 10. The code-generated implementation of Sink from Figure 4.

### 6.7 Reactor

As mentioned, it is the reactions that are the fundamental unit of computation and where the main control logic is synthesized. Reactors are best thought of as containers that allow for multiple reactions sharing access to the same state variables. Figure 10 shows the generated `Sink` reactor from Figure 4. It contains three reaction modules each connected through an `TokenReadMaster`, to a set of `InputPorts` and `Connection` modules. Note that the direction of the arrow indicates the master-slave relationship, not the flow of tokens. For example, the arrow from the reactor `Contained` to the input port `inArray` indicates that `Contained` has a `TokenReadMaster` interface which is connected to a `TokenReadSlave` interface at `inArray`. The tokens flow in the opposite direction. All reactions are connected with read and write ports to the state variable `count`. The precedence relationship, shown with dashed orange lines in Figure 4, is implemented through the `Connection` modules separating the reactions. Notice also the `Connection` from `Reaction 3` and `Reaction 1`. This ensures that `Reaction 1` does not start processing its next token until `Reaction 3` is done with the first. This connection is initialized with a single pure token.

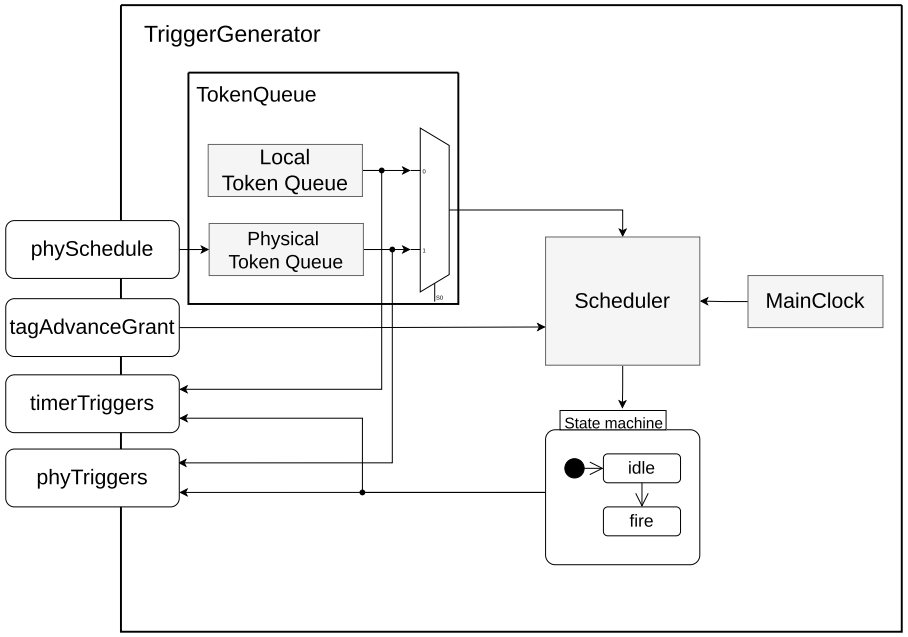



Fig. 11. The TriggerGenerator module in reactor-chisel.

A reaction will fire when there are tokens available on all of its `TokenReadMaster` interfaces. When it fires, the user-written Chisel design will execute until it is done. Any write port not used by the user-written design will have an absent token written to it. This will include the generated precedence connections. In this way, the reactions within the same reactor are executed strictly in order.

The `InputPort` modules are connected to one of three IO bundles (shown in rounded rectangles at the border of the module). `ReactorIO` contains all the actual LF input and output ports, e.g., `inArray` and `inSingle`. `PhysicalIO` contains the physical actions, in this case, the phy dependency. Finally `TriggerIO` contains all timers and built-in triggers like startup and shutdown. There is also an `ExternalIO` which contains ports annotated with `external`. The `ExternalIO` bundle allows reactions to directly read and write to wires routed to the top level of the design.

### 6.8 Triggers

The tokens that enable reactions and are communicated via ports and connections, ultimately originate from triggers which are all managed by a central module called `TriggerGenerator` shown in Figure 11. This module exposes a `timerTriggers` interface and a `phyTriggers` interface. Both are vectors of `TokenWriteMaster` interfaces, containing one interface per timer and physical action in the program. The startup and shutdown triggers are also emitted through the `timerTriggers` interface. `TriggerGenerator` also accepts scheduling of physical actions through its `phySchedule` interface which is a vector of `TokenWriteSlave` interfaces, one per physical action. The purpose of the `TriggerGenerator` is to output sequences of tokens on the `timerTriggers` and `phyTriqggers` interfaces that correspond to the timers in the LF program and the asynchronous scheduling of physical actions. This is achieved through the two "queues" in `TokenQueue`, one for physical actions (`PhysicalTokenQueue`), and one for timers and built-in triggers (`LocalTokenQueue`). Both expose an output signal called the **Next Event Tag (NET)** which is the tag of their next token, as well as a

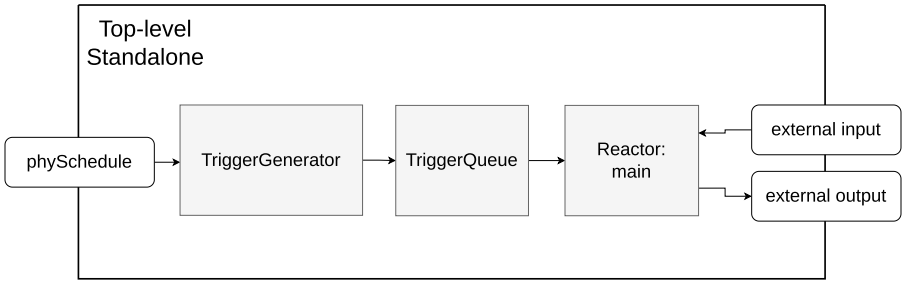



Fig. 12. The top level of a standalone LF program executing in HW.

*trigger vector* indicating which triggers are present at this tag. The token queue with the lower NET is forwarded to the outputs of the `TokenQueue` module. The `Scheduler` module contains the firing logic. It accepts an NET from the `TokenQueue`, an external **Tag Advance Grant (TAG)**, and the current time from the `MainClock`. In the case of standalone FPGA designs based on the Chisel-target, the TAG is hardwired to a value greater than any tag (a forever tag). When the `Scheduler` decides to fire, the trigger vector from the `TokenQueue` is forwarded to the timerTriggers and `phyTriggers` interfaces and the `TokenQueue` can move to the next step in schedule.

The `PhysicalTokenQueue` is an FIFO queue implemented using the `Queue` primitive from the Chisel3 standard library. The `LocalTokenQueue` is technically not a queue, but rather two fixed schedules assembled at compile time. A hyperperiod is computed based on the period and offsets of the timers in the program. The schedules last a single hyperperiod, containing a trigger vector for each step. The *initial* schedule contains the trigger vectors to be outputted on the very first run through the hyperperiod. Due to the possibility of timers with no period, this round might be different than the rest, encoded in the *periodic* schedule. There can also be a trigger vector associated with the termination of the program if there are any reactions triggered by the built-in shutdown trigger.

The semantics of a set of interconnected HW reactors are defined with respect to the tags of the tokens communicated between them. This is ultimately made possible by the `TriggerGenerator` module which synchronizes the creation of tokens.

Figure 12 brings everything together for "standalone" programs, i.e., programs that run exclusively in HW. It shows how the `TriggerGenerator` is connected through a `TriggerQueue` to the main reactor. The trigger queue is simply a queue for the tokens and gives some flexibility in the face of bursty behavior. The `TokenWriteSlave` interfaces associated with `phySchedules` is routed to the very top level of the design together with any port marked as `external`.

### 6.9 HW–SW Codesign

So far, we have only discussed how `reactor-chisel` enables designing FPGA accelerators with reactor semantics. However, one of the key benefits of `reactor-chisel` is that it enables deterministic coordination with external reactors, e.g., running on the CPUs of an SoC FPGA. We leverage the open-source project fpga-tidbits [24] for creating the platform-independent communication channels between HW and SW using either CSRs or shared memory.

Consider again the `VectorAdd` program in Listing 1. This program consists of two SW reactors, `SW` and `main`, and one HW reactor, `Accel`. Dataflowing on the connections between HW reactors and SW reactors are either sent through memory-mapped CSRs synthesized to the FPGA or through shared memory. Ports carrying single tokens will use CSRs while ports carrying array tokens use the shared memory. For instance, in the `VectorAdd` program in Listing 1, all the ports carry

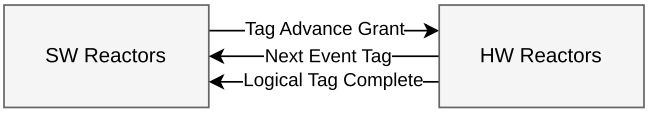



Fig. 13. Coordination signals between HW reactors and SW reactors.

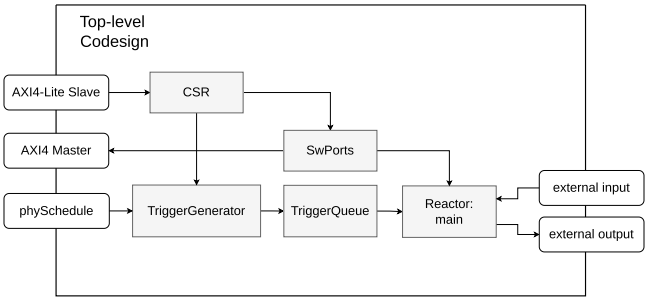



Fig. 14. The top-level FPGA module generated when using the Codesign-target.

array tokens. `Reactor-chisel` includes DMA modules which lets the reactions access the tokens in off-chip DRAM as if they were ordinary array tokens buffered in on-chip BRAM. If there are multiple array connections from SW to HW, `reactor-chisel` will multiplex the memory requests to off-chip DRAM onto a single physical memory interface.

The execution of reactors in SW and HW must be synchronized to correctly deliver reactor semantics. We employ a subset of the coordination scheme used in distributedLF programs [8]. Three signals are needed between the HW reactors on the FPGA and the SW reactors on the CPU as seen in Figure 13. (1) The NET signal informs the CPU of the tag of the next token on the token queue in the FPGA. (2) The **Logical Tag Complete (LTC)** signal informs the CPU of the tag of the latest completed token in the FPGA. (3) The TAG is sent from the CPU to the FPGA and grants the FPGA permission to advance its time until a specific tag.

This coordination interface is implemented through a set of CSRs. The `TriggerGenerator` introduced in Section 6.8 handles the control logic from the FPGA side. It does not release the next token until it has received a TAG from the SW that is greater than its NET. Moreover, as a token is released, the tag is queued such that it can be dequeued and written to the LTC status register when it has fully propagated through the HW reactions.

Figure 14 shows the generated top-level design on the FPGA when using the Codesign-target of LF. The CSRs are read and written to, by the SW reactors, through an AXI4Lite slave interface. Array connections are facilitated through shared memory, accessed through an AXI4 master interface. External inputs and outputs can be connected to other designs on the FPGA, or directly to IO pins. Finally, scheduling of physical actions can be done from external FPGA designs which in turn can be connected to IO pins.

On the SW side, we synthesize a special reactor in charge of communicating and synchronizing with the HW reactors. Figure 15 shows the synthesized reactor for the `VectorAdd` program in Listing 1. The reactor called `_FpgaWrapper` is generated based on the reactor `Accel` from Listing 1. Notice that it duplicates all of its ports. There are three reactions. Reaction 1 handles the startup (the startup trigger is represented by a white circle), this involves programming the bitstream of `Accel` onto the FPGA and initializing the shared memory regions used for HW–SW communication. Reaction 2 is triggered by the input port and contains most of the logic. It forwards data from the `SW` reactor by writing it to shared memory and passing the address to the `Accel` reactor through a

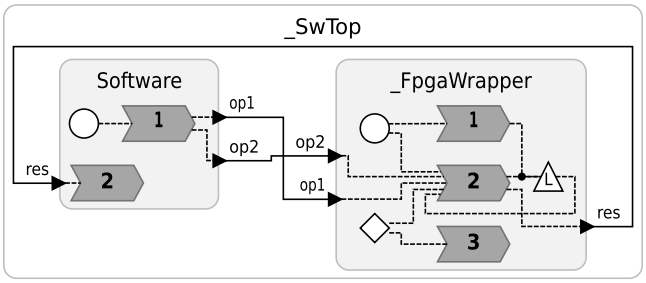
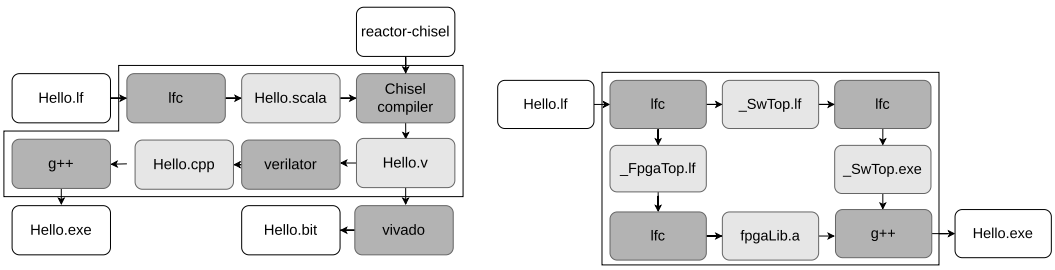



Fig. 15. Diagram of synthesized SW reactor for Listing 1, handling communication and synchronization with the HW reactors.




(a) Compilation of Lingua Franca programs for the Chisel target

(b) Iterative compilation of Lingua Franca programs for the Codesign target

Fig. 16. Compiling LF designs for SoC FPGA with the Chisel-target and the Codesign-target.

CSR. Data coming from the `Accel` reactor, through the `res` port, is copied from the FPGA memory region and outputted over the duplicate `res` port. This reaction is also triggered by a logical action which is scheduled whenever there are tokens originating from the HW reactors. Finally, Reaction 3 is triggered by shutdown (represented by a white diamond). Here we do cleanup and free any allocated memory. This synthesized reactor relies on fpga-tidbits for communicating with the FPGA or doing co-simulation of HW and SW.

### 6.10 Compilation

To generate these circuits, we extend the `lfc` with two new target languages. The Chisel-target supports generating code for HW reactors written in Chisel. The Codesign-target enables the compilation of LF programs consisting of both SW reactors written in C++ and HW reactors written in Chisel.

Figure 16 shows the steps involved in compiling LF programs for SoC FPGAs. white rectangles are the initial input and final outputs, grey rectangles are intermediate artifacts, dark grey rectangles are compilers and tools, and the large white rectangle encircles all the steps performed automatically when invoking `lfc`. First, we look at compiling standalone circuits shown in Figure 16(a). We begin in the top left corner with the program `Hello.lf` passed to `lfc`. This step transforms an input LF program into a Chisel program (`Hello.scala`). This generated Chisel program is then compiled, together with the `reactor-chisel` runtime, by the Chisel compiler, which eventually emits a Verilog description (`Hello.v`). This Verilog description can either pass through `verilator`,

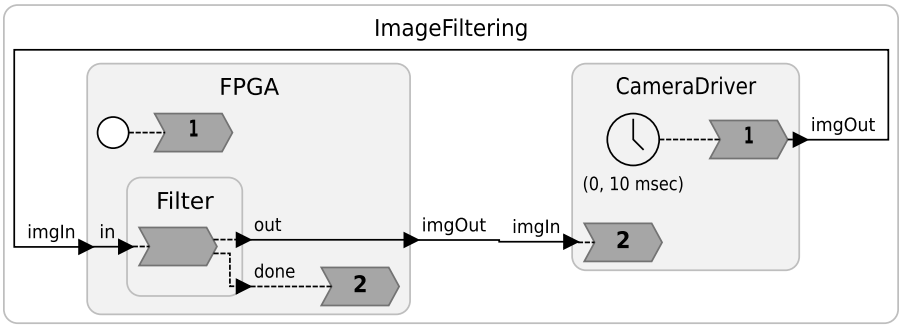



Fig. 17. A simple HW/SW image processing application.

generating a C++ emulation of the circuit (`Hello.cpp`) before a C++ compiler like `g++` can generate an executable (`Hello.exe`). This is the default behavior of our toolchain. A single click turns the LF program into an executable emulation of the circuit. When the user eventually wants to target real HW, the generated Verilog must be passed through a proprietary tool like `Vivado`, which will result in a bitstream (`Hello.bit`) that can be programmed onto an FPGA.

The compilation steps for the Codesign-target are more complicated and shown in Figure 16(b). We start in the top left corner with `Hello.lf`, but this time, it uses the Codesign-target and includes SW and HW reactors. When `lfc` is passed such a program, it will generate two new projects. The HW reactors will be extracted to a new project organized in `_FpgaTop.lf`; this uses the normal Chisel-target and is passed through `lfc` again through a process similar to the one in Figure 16(a). However, it outputs a static library (`fpgaLib.a`), not an executable. The SW reactors are put in a separate project with the reactor synthesized for HW-SW communication, shown in Figure 15. This is organized in `_SwTop.lf` which uses the LF C++ target. This file is passed through `lfc` again and finally linked with `fpgaLib.a` to produce the final executable. If we are targeting emulation, then the final executable includes the verilator emulation of the circuit. If we are targeting a real FPGA, the final executable only includes the drivers for interacting with the FPGA.

We make use of fpga-tidbits to enable this seamless transition between emulation in verilator and deployment on real HW.

## 7 Evaluation

We evaluate `reactor-chisel` on a set of test programs in terms of resource utilization, latency, and throughput. We synthesize our designs using Vivado 2021.2 targeting the xc7z020clg484-1 part, with a 5-nanosecond clock period. All benchmarks and test programs are available online together with documentation at Github.[1]

### 7.1 Image Processing Example

We begin with a more qualitative evaluation of a real-time image processing example shown in Figure 17. It consists of two top-level reactors. `CameraDriver` executes in SW and is triggered every 10 milliseconds. It fetches a new image from the camera, in our implementation, it just reads an image from the filesystem, and writes it to an output port connected to the `FPGA` reactor which executes on the FPGA. The `FPGA` reactor contains another reactor called `Filter` which performs the image processing algorithm. There are also two other reactions which are used for instrumentation. For simplicity, we chose a grayscale filter as the image processing workload.

[1] https://github.com/erlingrj/reactor-chisel-benchmarks

Table 1. Comparing a Handwritten Image-Processing Accelerator with a Reactor-Chisel-Based Implementation

| | Handwritten | `Reactor-chisel` |
|---|---|---|
| LUTs | 835 | 1,087 |
| Flip-flops | 1,018 | 1,285 |
| Fmax (MHz) | 221 | 216 |
| Latency (clock cycles) | 576,024 | 614,423 |
| HW loc | 130 | 77 |
| SW loc | 87 | 49 |

LUTs, lookup tables.

The filter is implemented by doing a weighted sum of the three input channels [27], split into two stages.

A "handwritten" implementation of the same application was also done using C++ and Chisel. The C++ program must manually set up the shared memory area and write the image into a buffer allocated there, before passing the address and size to the CSR and drive a `start` register high. The accelerator waits on the start signal before using a DMA to fetch the image from shared memory. The image is streamed through the filter, one pixel at a time, and the resulting grayscale pixel is written back through the same DMA. The handwritten implementation is also based on fpga-tidbits [24] for generating CSRs, DMAs and SW drivers. Arguably, this is a fair comparison since `reactor-chisel` itself is based on fpga-tidbits. Thus both versions will experience similar overheads from the implementation of DMAs and CSRs which is outside the scope of this work. The same grayscale filter module is used in both examples. Only the coordination fabric is different.

A quantitative comparison is done in Table 1. Clearly, `reactor-chisel` introduces an overhead in terms of resource utilization, critical path, and runtime execution. However, we do believe that it is an acceptable tradeoff for reduced complexity of the overall project. We see a significant decrease in the number of lines of code in both HW and SW components. The key advantage of LF and `reactor-chisel` is that it brings the entire computation, spanning CPUs and FPGA into a single program based on a single MOP. This enables the developer to precisely specify, at a high level of abstraction, the intended behavior across CPU and FPGA. Moreover, being a declarative language, LF enables, auto-synthesis of diagrams like the one in Figure 17. This is tremendously helpful in understanding the semantics of the computation. Moreover, the timed semantics and the static topology of the reactors open the door for interesting compiler optimizations. For instance, in Figure 17, we can easily infer, at compile time, that the two connections will carry an image frame every 10 millisecond. If there are more connections, the compiler could figure out which to multiplex onto the same AXI ports and which to give dedicated ports by identifying at which tags the different connections might carry tokens. Such compiler optimizations are left for future work.

### 7.2 Resource Utilization

Table 2 shows the resource utilization for a set of programs using the Chisel-target, aimed for standalone execution on the FPGA, i.e., no synchronization with CPU or off-chip DRAM. The test programs include minimal reaction bodies that only do enough to avoid being optimized away. We are thus trying to measure solely the overhead by the runtime. The names of the tests are meant to illustrate what they are measuring. For instance, `ReactionN` measures the overhead of $N$ reactions.

The `Reactions1` test represents the smallest program we can envision. It thus gives an idea of the baseline overhead introduced by the runtime. Going from one to two reactions in the same

Table 2. Resource Utilization for Standalone Circuits Using the Chisel-Target

| Test | LUTs | Flip-flops | BRAMs (Kb) | Fmax (MHz) |
|---|---|---|---|---|
| ArrayConnection | 139 | 136 | 72 | 334.90 |
| Hierarchy | 65 | 117 | 0 | 323.73 |
| PhysicalAction | 139 | 171 | 0 | 224.92 |
| PhysicalAction2 | 153 | 193 | 0 | 251.32 |
| PhysicalAction10 | 210 | 215 | 0 | 239.92 |
| Reactions1 | 62 | 119 | 0 | 319.18 |
| Reactions2 | 79 | 128 | 0 | 309.21 |
| Reactions3 | 91 | 134 | 0 | 328.84 |
| Reactions10 | 165 | 180 | 0 | 303.49 |
| Timers1 | 62 | 119 | 0 | 319.18 |
| Timers2 | 127 | 140 | 0 | 315.96 |
| Timers3 | 146 | 178 | 0 | 319.80 |
| Timers10 | 375 | 304 | 0 | 286.86 |

reactor costs about 20 **lookup tables (LUTs)** and 10 flip-flops. The resource cost is not linear and a further increase from 2 to 10 reactions only incurs a 12 LUT and 7 flip-flop cost, on average. The addition of more reactions does not have a substantial effect on the critical path.

Increasing the number of timers in a program is slightly more expensive. On average, adding a timer to the design costs 31 LUTs and 20 flip-flops. It is more expensive to go from a single timer to a two timers due to optimizations performed for single-timer designs.

The `PhysicalAction` test shows the cost of introducing physical actions. As described in Section 6.8, physical actions are scheduled onto a separate token queue, whose head is compared against the head of the ordinary token queue. Adding a single physical action adds 77 LUTs and 52 flip-flops, compared to a program with a single timer. Adding more physical actions is cheaper.

In the `Hierarchy`, we introduce an empty wrapper reactor around the program from `Reactions1`. There is virtually no additional cost showing that the hierarchy introduced by reactors is a zero-cost abstraction.

Finally, in `ArrayConnection`, we have a test program consisting of a source and a sink reactor that communicates array tokens. Each array token is 32-bit wide with a length of 2,048 elements. While a normal connection carry tokens consisting of a single element which is buffered in flip-flops, an *array connection* carries an array of elements which are then buffered in BRAMs. Refer to Section 6.5 for a more complete description. In the `ArrayConnection` test program, we have two reactors communicating over a 2,048-element, 32-bit wide array connection. As expected, the results shows a usage of 64 Kb BRAMs.

Table 3 shows the resource utilization of programs using the Codesign-target. This means that the resulting designs include DMA modules, CSRs, AXI Lite slave for giving access to the CSRs, AXI Master ports to main memory.

In `SingleConnection1` and `Singleconnection2`, we have a single SW reactor executing on the CPU and a single HW reactor executing on the FPGA and one and two normal connections between them, respectively. We can consider `SingleConnection1` the "baseline" overhead for the Codesign-target with the order of 300 LUTs and 500 flip-flops. This is mainly consumed by the CSRs, which account for 86% of the LUTs and 69% of the flip-flops. Adding another single token

Table 3. Resource Utilization for HW–SW Designs Using the Codesign-Target

| Test | LUTs | Flip-flops | BRAMs (Kb) | Fmax (MHz) |
|---|---|---|---|---|
| ArrayConnections1 | 497 | 661 | 0 | 240.85 |
| ArrayConnections2 | 814 | 891 | 0 | 239.52 |
| SingleConnection1 | 308 | 520 | 0 | 253.94 |
| SingleConnection2 | 320 | 596 | 0 | 257.80 |
| WithLocalEvents | 421 | 576 | 0 | 234.58 |
| WithoutLocalEvents | 308 | 520 | 0 | 253.94 |

connection from SW to HW adds, 12 LUTs and 76 flip-flops. Again, this is almost exclusively spent on CSRs.

In `WithLocalEvents` and `WithoutLocalEvents`, we look at the cost of going from a purely reactive HW reactor which receives all its tokens from SW to a HW reactor that also generates tokens through a timer. This costs an additional 113 LUTs and 56 flip-flops. This is mainly used in the read/write logic of the CSRs and the `TriggerGenerator`.

`ArrayConnections1` and `ArrayConnections2` show the overhead of array connections between HW reactors and SW reactors. Array connections between two HW reactors are implemented in on-chip BRAMs, while array connections between SW and HW connections are implemented in off-chip DRAM. This is evident from the results in Table 3 showing zero BRAM usage for these connections.

Relative to a single token connection, an array connection costs 189 LUTs and 141 flip-flops. Going from a single array connection to two array connections adds 317 LUTs and 230 flip-flops. For every array connection between HW and SW, a DMA module is created. If there are more than two, then a multi-channel memory system is created which allows multiple read requests and read responses to be multiplexed onto a single memory port. All of these components are part of the FPGA-tidbits project.

In general, the communication across HW and SW adds some overhead, both in terms of resources and maximum clock frequency. This would also be the case for *ad hoc* designs using, e.g., AMD/Xilinx IP blocks for CSRs, AXI Lite slave, AXI4 Master, and DMAs. We leave it for future works to investigate how the underlying framework, i.e., FPGA-tidbits, can be improved to reduce resource utilization.

### 7.3 Latency

Table 4 shows the results of the latency benchmarks which measure the reaction invocation latencies for various program topologies. It is measured by comparing the logical and physical time, i.e., the lag, at the time the reaction is triggered. These benchmarks use the Chisel-target, meaning that we are measuring latencies on the FPGA fabric.

`TimerLatency1` shows that the invocation latency for a single reaction triggered by a timer is zero clock cycles. We achieve this by releasing tokens from the token queue a few clock cycles too early to account for the pipeline stages. This is a static optimization. Reactions triggered deeper in the containment hierarchy will have more pipeline stages between them and the TriggerGenerator and will thus experience different invocation latencies. This highlights the subtle difference between the logical and physical time points at which these reactions are invoked.

Table 4. Reaction Invocation Latency for Programs Using the Chisel-Target

| Test | Latency (clock cycles) |
|---|---|
| ArrayConnectionLatency1 | 5 |
| ArrayConnectionLatency2 | 260 |
| ArrayConnectionLatency3 | 515 |
| MutexLatency | 4 |
| TenStagePipeline | 27 |
| TimerLatency1 | 0 |

Table 5. Token Throughput for Programs Using the Chisel-Target

| Test | Throughput (clock cycles per event) |
|---|---|
| DeepPipe | 7 |
| ShallowPipe | 7 |
| TenParallelPipes | 7 |

`MutexLatency` shows that a second reaction in the reactor, triggered by the same timer, will experience an additional four clock cycles latency due to mutual exclusion. The first reaction executes in a single cycle and there are three additional cycles of overhead.

`ArrayConnectionLatency1` shows that it takes five cycles between an upstream reaction writing a single value to an array connection and a downstream reaction is triggered and has fetched the value. In `ArrayConnectionLatency2`, the upstream reaction writes 256 words to the array connection, here the downstream is triggered after 260 clock cycles. Finally, in `ArrayConnectionLatency3`, the upstream reaction writes 256 words and the downstream reads out all 256 words. Here the latency is 515 clock cycles. This highlights the semantics of array connections. The upstream must complete all of its writes before the downstream can start reading. Thus, the reading does not start until 256 cycles have passed. We are working on a FIFO token that can get around this issue.

Lastly, in `TenStagePipeline`, we have a pipeline of 10 reactors, each containing a single reaction executing in a single clock cycle. We measure the time needed to traverse from the first to the last stage to 27 clock cycles. This is four cycles per hop, consistent with the result in `MutexLatency`.

### 7.4 Throughput

Our throughput benchmarks measure the number of clock cycles spent per single token for various designs in the Chisel-target (Table 5). We do this by generating tokens at the highest frequency, i.e., a timer with a single clock cycle period, and measuring how many clock cycles are needed to process a fixed amount of tokens. The processing consists of pipelines of reactors, and each reactor contains a single reaction executed in a single cycle. In `DeepPipe`, `ShallowPipe`, and `TenParallelPipes`, we show that in such a pipeline design, we can handle a new token every seven clock cycles. This delay is mainly caused by two factors. First, the state machine that controls the execution of a reaction wastes one cycle before enabling the reaction and another after. If a reaction body executes one clock cycle, then the total execution will be three clock cycles. The second contributing factor is that a reaction is not enabled until its immediate downstream has completed execution. This is to protect

the memory in the connections from concurrent access from upstream and downstream. Both of these overheads are addressable through further optimizations. We could allow the upstream to start execution but block it if it tries writing to its output port. We are also working on the concept of a FIFO token that resembles array tokens. However, FIFO tokens are buffered in the connections using FIFOs instead of RAM. This would enable the downstream reaction to be triggered when the first write occurs, rather than the completion of the upstream. Such optimizations are left for future work.

## 8 Conclusion

We have presented the HW reactor model, a subset of the reactor MoC and a convenient abstraction level for doing HW–SW codesign for cyber-physical systems. We have also presented `reactor-chisel`, a Chisel implementation of a HW runtime delivering the semantics of the HW reactor model. We have also extended the polyglot coordination language LF with a Chisel-target and a Codesign-target, enabling HW–SW to codesign on the reactor level. We have shown several example programs illustrating the ease of which HW–SW codesign is performed in the Codesign-target of LF. We evaluated our methodology on a set of benchmarks showing relatively low resource utilization and promising latency and throughput numbers. The work is open-source and available on Github.[2]

## Acknowledgment

We would also like to acknowledge the reviewers for their help in improving the article and the contributors to the LF project for enabling this work.

## References

[1] Khronos. 2021. SYCL. Retrieved July 11, 2023 from https://www.khronos.org/sycl/

[2] AMD. 2023. Vitis Unified Software Platform Documentation: Embedded Software Development. Retrieved July 11, 2023 from https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded

[3] Accellera Systems Initiative. 2012. IEEE standard for standard systemc language reference manual. *IEEE Standards* 1666–2011.

[4] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. 2014. ReconOS: An operating system approach for reconfigurable computing. *IEEE Micro* 34, 1 (2014), 60–71. DOI: https://doi.org/10.1109/MM.2013.110

[5] Arvind Arvind, Rishiyur S. Nikhil, Daniel Rosenband, and Nirav Dave. 2004. High-level synthesis: An essential ingredient for designing complex ASICs. In *Proceedings of the International Conference on Computer Aided Design (ICCAD '04)*.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the Design Automation Conference (DAC '12)*. ACM/EDAC/IEEE, 1216–1225.

[7] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Hokeun Kim, Shaokai Lin, Christian Menard, and Edward A. Lee. 2023. Risk and mitigation of nondeterminism in distributed cyber-physical systems. In *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '23)*. ACM, New York, NY, 1–11. DOI: https://doi.org/10.1145/3610579.3613219

[8] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Rohan Tabish, Hokeun Kim, Shaokai Lin, Christian Menard, Cong Liu, and Edward A. Lee. 2022. Xronos: Predictable coordination for safety-critical distributed embedded systems. arXiv:2207.09555. Retrieved from https://arxiv.org/abs/2207.09555

[9] Albert Benveniste and Gérard Berry. 1991. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE* 79, 9 (1991), 1270–1282.

[10] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. Retrieved from http://citeseerx.ist.psu.edu/view

[2] https://github.com/erlingrj/reactor-chisel

[11] Dominique Blouin, Gilberto Ochoa-Ruiz, Yvan Eustache, and Jean-Philippe Diguet. 2015. Kaolin: A system-level AADL tool for FPGA design reuse, upgrade and migration. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '15)*. 1–8. DOI: https://doi.org/10.1109/AHS.2015.7231166

[12] Etienne Borde, Smail Rahmoun, Fabien Cadoret, Laurent Pautet, Frank Singhoff, and Pierre Dissaux. 2014. Architecture models refinement for fine grain timing analysis of embedded systems. In *Proceedings of the 2014 25nd IEEE International Symposium on Rapid System Prototyping*. 44–50. DOI: https://doi.org/10.1109/RSP.2014.6966691

[13] Jeronimo Castrillon, Karol Desnos, Andrés Goens, and Christian Menard. 2022. *Dataflow Models of Computation for Programming Heterogeneous Multicores*. Springer Nature, Singapore, 1–40. DOI: https://doi.org/10.1007/978-981-15-6401-7_45-2

[14] Jack B. Dennis. 1974. *First Version Data Flow Procedure Language*. Report MAC TM61. MIT Laboratory for Computer Science.

[15] S. A. Edwards. 2006. The challenges of synthesizing hardware from C-like languages. *IEEE Design & Test of Computers* 23, 5 (2006), 375–386. https://doi.org/10.1109/MDT.2006.134

[16] Kermin Fleming and Michael Adler. 2016. The LEAP FPGA operating system. *FPGAs for Software Programmers* (2016), 245–258.

[17] Håvard Grip. 2021. Suviving an In-Flight Anomaly: What Happened on Ingenuity's Sixth Flight. NASA Mars Helicopter Blog. Retrieved from https://mars.nasa.gov/technology/helicopter/status/305/surviving-an-in-flight-anomaly-what-happened-on-ingenuitys-sixth-flight/

[18] C. A. R. Hoare. 1978. Communicating sequential processes. *Communications of the ACM* 21, 8 (1978), 666–677.

[19] Axel Jantsch. 2003. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.

[20] Axel Jantsch and Ingo Sander. 2005. Models of computation and languages for embedded system design. *IEE Proceedings—Computers and Digital Techniques* 152 (April 2005), 114–129. DOI: https://doi.org/10.1049/ip-cdt:20045098

[21] Erling R. Jellum, Torleiv H. Bryne, Tor A. Johansen, and Milica Orlandic. 2022. The syncline model-analyzing the impact of time synchronization in sensor fusion. In *Proceedings of the IEEE Conference on Control Technology and Applications (CCTA '22)*. 1446–1453. DOI: https://doi.org/10.1109/CCTA49430.2022.9966179

[22] Erling R. Jellum, Shaokai Lin, Peter Donovan, Chadlia Jerad, Edward Wang, Marten Lohstroh, Edward A. Lee, and Martin Schoeberl. 2023. InterPRET: A time-predictable multicore processor. In *Proceedings of the Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week '23)*. ACM, New York, NY, 331–336. DOI: https://doi.org/10.1145/3576914.3587497

[23] Erling R. Jellum, Shaokai Lin, Peter Donovan, Efsane Soyer, Fuzail Shakir, Torleiv Bryne, Milica Orlandic, Marten Lohstroh, and Edward A. Lee. 2023. Beyond the threaded programming model on real-time operating systems. In *Proceedings of the 4th Workshop on Next Generation Real-Time Embedded Systems (NG-RES '23) (Open Access Series in Informatics (OASIcs)*, Vol. 108. Federico Terraneo and Daniele Cattaneo (Eds.), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:13. DOI: https://doi.org/10.4230/OASIcs.NG-RES.2023.3

[24] Erling R. Jellum, Yaman Umuruglu, Milica Orlandic, and Martin Schoeberl. 2023. FPGA-tidbits: Rapid prototyping of FPGA accelerators in chisel. In *Proceedings of the 26th Euromicro Conference on Digital System Design (DSD '23)*. 153–160. DOI: https://doi.org/10.1109/DSD60849.2023.00031

[25] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. North-Holland Publishing Co., 471–475.

[26] Myron D. King. 2013. *A Methodology for Hardware-Software Codesign*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[27] Kaushal Kumar, Ritesh Kumar Mishra, and Durgesh Nandan. 2020. Efficient hardware of RGB to gray conversion realized on FPGA and ASIC. *Procedia Computer Science* 171 (2020), 2008–2015. DOI: https://doi.org/10.1016/j.procs.2020.04.215

[28] Edward A. Lee. 2006. The problem with threads. *Computer* 39, 5 (2006), 33–42. DOI: https://doi.org/10.1109/MC.2006.180

[29] Edward A. Lee. 2009. Computing needs time. *Communications of the ACM* 52, 5 (2009), 70–79. DOI: https://doi.org/10.1145/1506409.1506426

[30] E. A. Lee and D. G. Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers* C-36, 1 (1987), 24–35. DOI: https://doi.org/10.1109/TC.1987.5009446

[31] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proceedings of the IEEE* 75, 9 (1987), 1235–1245. DOI: https://doi.org/10.1109/PROC.1987.13876

[32] Edward A. Lee and Alberto Sangiovanni-Vincentelli. 1998. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems* 17, 12 (1998), 1217–1229. Retrieved from http://ptolemy.eecs.berkeley.edu/publications/papers/98/framework/

[33] Christian Lienen, Marco Platzner, and Bernhard Rinner. 2020. ReconROS: Flexible hardware acceleration for ROS2 applications. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT '20)*. 268–276. DOI: https://doi.org/10.1109/ICFPT51103.2020.00046

[34] Marten Lohstroh. 2020. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html

[35] Marten Lohstroh, Edward Lee, Soroush Bateni, Christian Menard, Peter Donovan, Clément Fournier, Hou Seng (Steven) Wong, Alexander Schulz-Rosengarten, Erling R. Jellum, Hokeun Kim, Matt Weber, Shaokai Lin, and Anirudh Rengarajan. 2024. *Lingua Franca*. Retrieved from https://github.com/lf-lang/lingua-franca

[36] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 4 (May 2021), Article 36, 1–27. DOI: https://doi.org/10.1145/3448128

[37] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2020. Reactors: A deterministic model for composable reactive systems. In *Cyber Physical Systems. Model-Based Design*. Roger Chamberlain, Martin E. Grimheden, and Walid Taha (Eds.), Springer International Publishing, Cham, 59–85.

[38] Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. 2019. Actors revisited for time-critical systems. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*, 152:1–152:4.

[39] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon, and Edward A. Lee. 2023. High-performance deterministic concurrency using lingua franca. *ACM Transactions on Architecture and Code Optimization* 20, 4 (August 2023), 1–29. DOI: https://doi.org/10.1145/3617687 Just Accepted.

[40] Aaftab Munshi. 2009. The opencl specification. In *Proceedings of the IEEE Hot Chips 21 Symposium (HCS '09)*. IEEE, 1–314.

[41] Ingo Sander. 2003. *System Modeling and Design Refinement in ForSyDe*. Thesis.

[42] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Edward A. Lee, and Soroush Bateni. 2023. Polyglot modal models through lingua franca. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. 337–342.

[43] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. 2004. System design using Khan process networks: The Compaan/Laura approach. *Proceedings Design, Automation and Test in Europe Conference and Exhibition* 1 (2004), 340–345.

[44] Wikipedia. 2023. Round-Trip Engineering — Wikipedia, The Free Encyclopedia. Retrieved September 19, 2023 from http://en.wikipedia.org/w/index.php?title=Round-trip%20engineering{&}oldid=1172040452

Received 22 September 2023; revised 15 April 2024; accepted 14 May 2024