From SSA to Synchronous Concurrency and Back

Hugo Pompougnac, Ulysse Beaugnon, Albert Cohen, Dumitru Potop-Butucaru

From SSA to Synchronous Concurrency and Back

Hugo Pompougnac (project-team KAIROS), Ulysse Beaugnon (Google), Albert Cohen (Google), Dumitru Potop-Butucaru (project-team KAIROS)

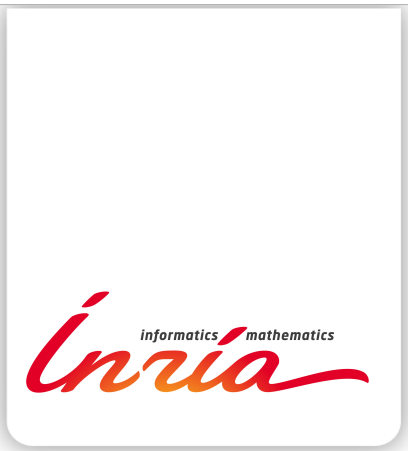
RESEARCH REPORT

N° 9380

December 2020

Project-Teams KAIROS

ISSN 0249-6399 ISRN INRIA/RR--9380--FR+ENG



From SSA to Synchronous Concurrency and Back

Hugo Pompougnac (project-team KAIROS), Ulysse Beaugnon (Google), Albert Cohen (Google), Dumitru Potop-Butucaru (project-team KAIROS)

Project-Teams KAIROS

Research Report n° 9380 — December 2020 — 23 pages

Abstract: We are interested in the programming and compilation of reactive, real-time systems. More specifically, we would like to understand the fundamental principles common to general-purpose and synchronous languages—used to model reactive control systems—and from this to derive a compilation flow suitable for both high-performance and reactive aspects of a modern control application. To this end, we first identify the key operational mechanisms of synchronous languages that SSA does not cover: synchronization of computations with an external time base, cyclic I/O, and the semantic notion of absent value which allows the natural representation of variables whose initialization does not follow simple structural rules such as control flow dominance. Then, we show how the SSA form in its MLIR implementation can be seamlessly extended to cover these mechanisms, enabling the application of all SSA-based transformations and optimizations. We illustrate this on the representation and compilation of the Lustre dataflow synchronous language. Most notably, in the analysis and compilation of Lustre embedded into MLIR, the initialization-related static analysis and code generation aspects can be fully separated from memory allocation and causality aspects, the latter being covered by the existing dominance-based algorithms of MLIR/SSA, resulting in a *high degree of conceptual and code reuse*. Our work allows the specification of both computational and control aspects of high-performance real-time applications. It paves the way for the definition of more efficient design and implementation flows where real-time ressource allocation drives parallelization and optimization.

Key-words: code generation, compiler optimization, concurrency, declarative languages, embedded systems, language design, language implementation, real-time systems, semantics, software engineering, synchronous languages

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

De SSA à la concurrence synchrone, aller-retour

Résumé : Nous traitons de la programmation et de la compilation de systèmes réactifs, temps-réel. En particulier, nous cherchons à comprendre les principes fondamentaux communs à la programmation généraliste et aux langages synchrones—utilisés pour modéliser les systèmes de contrôle—et de là nous dérivons une méthode de compilation adaptée aux aspects réactifs et haute performance d'une application moderne. À cette fin, nous commençons par identifier les mécanismes des langages synchrones que SSA n'implémente pas : la synchronisation des calculs avec une base de temps externe, les entrées-sorties cycliques, et la notion sémantique de *valeur absente*, qui permet la représentation naturelle de variables dont l'initialisation ne suit pas de simples règles structurelles. Ensuite, nous montrons de quelle manière la forme SSA, dans l'implémentation de MLIR, peut être étendue pour implémenter ces mécanismes et leur appliquer toutes les transformations et optimisations basées sur SSA. Nous illustrons ces mécanismes par la représentation et la compilation du langage synchrone, flot de données Lustre. Nous montrons que les problèmes d'analyse statique pour l'initialisation, de génération de code, peuvent être entièrement distingués des problèmes d'allocation mémoire et de causalité, ces derniers étant pris en charge par les algorithmes d'analyse de la dominance de MLIR/SSA, ce qui permet *un haut niveau de réutilisation du code et des concepts.* Notre travail permet la spécification d'applications temps-réel, du point de vue du contrôle comme du calcul. Il ouvre la voie à la définition de processus de conception et d'implémentation plus efficaces, où la parallélisation et l'optimisation procèdent de l'allocation des ressources temps-réel.

Mots-clés : génération de code, optimisation de compilateur, concurrence, langages déclaratifs, systèmes embarqués, conception de langage, implémentation de langage, systèmes temps réel, sémantique, génie logiciel, langages synchrones

1 Introduction

The Static Single Assignment (SSA) form [17, 18] has proven an extremely useful tool in the hands of compiler builders. First introduced as an intermediate representation (IR) meant to facilitate optimizations, it became a staple of optimizing compilers. More recently, its semantic properties—e.g. functional *determinism* while still allowing for limited *concurrency*—established it as a sound basis for High-Performance-Computing (HPC) compilation frameworks such as MLIR [14], where different abstraction levels of the same application[1] share the structural and semantic principles of SSA, allowing them to co-exist while being subject to common analysis and optimization passes (in addition to specialized ones).

But while compilation frameworks such as MLIR concentrate the existing know-how in HPC compilation for virtually every execution platform, they lack a key ingredient needed in the *high-performance embedded systems* of the future—the ability to represent reactive control and real-time aspects of a system. They do not provide first-class representation and reasoning for systems with a cyclic execution model, synchronization with external time references (logical or physical), synchronization with other systems, tasks and I/O with multiple periods and execution modes.

And yet, as we shall see in this paper, while the standard SSA form does not cover these aspects, it shares strong structural and semantic ties with one of the main programming models for reactive real-time systems: *dataflow synchrony* [3, 11], and its large and structured corpus of theory and practice of RTE systems design.

Contribution. Relying on this syntactic and semantic proximity, we extend the SSA-based MLIR framework to open it to synchronous reactive programming of real-time applications. We illustrate the expressiveness of our extension through the compilation of the pure dataflow core of the Lustre language. This allows us to model and compile all data processing, computational and reactive control aspects of a signal processing application.[2] In the compilation of Lustre, following an initial normalization phase, all data type verifications, buffer synthesis, and causality analysis can be handled using existing MLIR SSA algorithms. Only the initialization analysis specific to the synchronous model (a.k.a. clock calculus or analysis) requires specific handling during analysis and code generation phases, leading to significant code reuse.

The MLIR embedding of Lustre is non-trivial. As modularity based on function calls is no longer natural due to the cyclic execution model, we introduce a *node instantiation* mechanism. We also generalize the usage of the special *undefined/absent* value in SSA semantics [8] and in low-level intermediate representations such as LLVM IR [12]. We clarify its semantics and strongly link it to the notion of absence and the related static analyses (*clock calculi*) of synchronous languages [2].

Our extension remains fully compatible with SSA analysis and code transformation algorithms. It allows giving semantics and an implementation to all correct SSA specifications. It also supports static analyses determining correctness from a synchronous semantics point of view.

Outline. In Section 2 we formalize the SSA semantics. Section 3 extends SSA to allow reactive synchronous programming. Section 4.1 covers the embedding of Lustre into MLIR SSA and its compilation. In Section 4.2 we discuss our signal processing use case. We discuss related work in Section 5 before the conclusion in Section 6.

[1] Ranging from ML dataflow graphs and linear algebra specifications down to affine loop nests and optimized (tiled, vectorized...) low-level code.

[2] A vocal pitch tuning vocoder.

```
<ssa_spec> ::= <function>+
 <function> ::= func <fun_name> <fun_iface> <fun_body>
<fun_iface> ::= (<type>*)->(<type>*)
<fun_body> ::= <block>+
    <block> ::= <blk_arg>:<blk_body>
  <blk_arg> ::= <block_name>(<tvar>*)
     <tvar> ::= <var>:<type>
<blk_body> ::= <op>* <term_op>
       <op> ::= (<var>*)=<op_id>(<tvar>*):<type>*
              | <var> = load(<tvar>):<type>
              | store(<tvar>,<tvar>)
<term_op> ::= cond_br <var> <blk_arg> <blk_arg>
              | br <blk_arg> | return(<tvar>*)
   <op_id> ::= <arith_op> | <bool_op> | call <fun_name>
```

Figure 1: SSA syntax

2 SSA syntax and semantics

The syntax and semantics of SSA form have been presented multiple times before, but we need to settle on one as a basis for extension. The particular SSA dialect we use is based on MLIR [14], which uses a continuation-passing style (CPS) for the specification of ϕ operators, and uses opcode names `br` and `cond_br` for unconditional and conditional branches, respectively.

2.1 Core SSA syntax

The syntax, presented in Fig. 1, consists of a minimal SSA syntax (in black) extended with the constructs needed to represent function-based modularity (in red), and with the `load` and `store` operations allowing the representation of memory side effects (in green).

Syntactic correctness must be complemented by the respect of a number of structural properties. No two blocks of a function may have the same name, and branching operations (`br` and `cond_br`) may only reference existing blocks with the correct number of arguments. Each variable is either output of exactly one operation, or argument of a single block header.[3]

No two functions may have the same name, and calling a function assumes that it exists and has the correct interface. The interface of a function consists in the input arguments of its first basic block (which are also the function arguments) and the inputs of its `return` operation, which are the function outputs. If multiple `return` operations exist in a function, they must have the same number of arguments. Every function has an interface declaration, which summarizes the types of input and output arguments.

All SSA blocks are terminated by a `return` or a branch operation. These operations are called *terminators*.

2.2 Example

To illustrate the specific properties of reactive systems, and the differences between Lustre and SSA we use the small example of Fig. 2. The Lustre node (top) and MLIR SSA function (bottom)

[3] Which under classical notation amounts to the variable being output of exactly one ϕ operator.

implement the same functionality, but we focus on the SSA function in this section. As we will see later, the latter could be the result of the compilation of the former.

Note that the MLIR SSA syntax requires that block identifiers start with "`^`" and that all variable names start with "`%`". It also provides more intuitive forms for the various operations: function calls specify the full function signature at their end; the comparison taking two input variables as arguments and producing one variable (the test result) has the syntax of line 22, which specifies the kind of test (`sge` for `>=`), the type of input data (`i32`), but not the type of the Boolean output, which is implicit (`i1`); operation `select` in line 23 outputs one of its data inputs based on the value of its Boolean test variable `%ck` (of type `i1`).

The blocks and the branching operations of an SSA function determine a sequential control flow graph (SCFG) structure, with the first block serving as control entry point. Inside each basic block, execution proceeds sequentially.

Our example is a stepper motor driver which receives rotation commands as increments in a 0.18° basis, but can only actuate (issue physical commands to the motor) for 1.8° at a time. For this reason, commands must be bufferized and only actuated when their number exceeds 10.

This behavior, typical of an embedded control system, involves a continuous interaction with the environment. In our example, this interaction is driven by the infinite loop formed by the SSA branching operations. Each iteration of the loop is an *execution cycle* during which the program reads its input `%inc` from its environment, performs computations, potentially actuates the motor and finally outputs `%pos_nxt`. Timely interaction with the environment is achieved through the function calls in violet, which are discussed in Section 3.

2.3 SSA operational semantics

For conciseness, we shall assume all variables (including Booleans) are semantically represented as integers. We also assume each operation of the program has a unique label, such as a program line number if we assume that no two operations share the same line.

Notations The cardinal of a set $\mathcal{S}$ is denoted $|\mathcal{S}|$. We use the OCAML notation for lists: $[]$ is the empty list, $h :: t$ the list of first element h and tail t. To represent partial valuations (of variables, memory) we use fully defined functions, adding the special value $\perp$—representing absence—to their codomain. We also denote $\perp$ any function that is constant $\perp$, regardless of its domain. *Int* is the domain of the integer type, and $\overline{Int} = Int \cup \{\perp\}$. Given a (mathematical) function f and a value x of its domain, $f[x \leftarrow y]$ is the function that is identical with f everywhere except on x, where it has value y.

$\mathcal{L}^{\mathtt{f}}$ and $\mathcal{V}^{\mathtt{f}}$ are respectively the sets of labels and of variables of an SSA function `f`, and $b_0^{\mathtt{f}}$ is its first basic block. The function containing basic block b is denoted $fun(b)$. The ordered set of inputs of basic block b is $in(b) \subseteq \mathcal{V}^{fun(\mathtt{b})}$, and $in(b)_i$ is the i^{th} input of b. The ordered set of variables assigned by operations of a block b is denoted $loc(b)$. The label of the first operation in block b is $fst(b)$. The operation associated with a label $l \in \mathcal{L}^{\mathtt{f}}$ is denoted $op(l)$. If $op(l)$ is not a terminator, then $nxt(l)$ is the label of the next operation in its block. The number of arguments of a `return` operation of function `f` is denoted O^f.

Execution state The execution state of an SSA *function* `f` is one of:

- An initial state $Start^{\mathtt{f}}(v_1, \ldots, v_{|in(b_0^{\mathtt{f}})|})$, where $v_i \in \overline{Int}$ are the actual parameters of the function.
- A final state $End^{\mathtt{f}}(w_1, \ldots, w_{O^f})$, where $w_i \in \overline{Int}$ are the outputs of `f` (received as input by `return`).

```
node stepper_drv(inc:int) returns (pos_nxt:int)
var pos,pos_inc,pos_tmp,cst:int; ck:bool;
let
  pos = 0 fby pos_nxt;
  pos_inc = pos+inc;
  pos_tmp = pos_inc-10;
  ck = (pos_tmp >= 0);
  pos_nxt = if ck then pos_tmp else pos_inc;
  cst = 0 when ck ;
  actuate(cst);
tel

func @stepper_drv()->() {
^start:
  %c0 = constant 0 : i32
  %c10 = constant 10 : i32
  br ^step(%c0:i32)
^step(%pos:i32):
  %inc = call @input_inc() : () -> (i32)
  %pos_inc = addi %pos, %inc : i32
  %pos_tmp = subi %pos_inc, %c10 : i32
  %ck = cmpi "sge", %pos_tmp, %c0 : i32
  %pos_nxt = select %ck,%pos_tmp,%pos_inc : i32
  cond_br %ck, ^act(%c0:i32), ^out
^act(%cst:i32):
  call @actuate(%cst) : (i32) -> ()
  br ^out
^out:
  call @output_pos_nxt(%pos_nxt) : (i32)->()
  call @tick() : () -> ()
  br ^step(%pos_nxt:i32)
}
```

Figure 2: Stepper motor driver in Lustre (top) and MLIR SSA (bottom). In green, control statements. In red, data processing operations. In violet, cyclic I/O and time synchronization. In blue, state manipulation.

$$(Start^{\mathtt{f}}(v_1,\ldots,v_{|in(b_0^{\mathtt{f}})|}),cs,m) \to (Run^{\mathtt{f}}(fst(b_0^{\mathtt{f}}),\bot[in(b_0^{\mathtt{f}})_k \leftarrow v_k \mid 1\le k\le |in(b_0^{\mathtt{f}})|]),cs,mem) \quad \textbf{(start)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = op\_id(w_1,\ldots,w_m)\text{“} \qquad (o_1,\ldots,o_n) = [\![op\_id]\!](s(w_1)\ldots,s(w_m))}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(nxt(l),s[v_i \leftarrow a_i \mid 1\le i\le n]),cs,m)} \quad \textbf{(opcall)}$$

$$\frac{op(l) = \text{“}\mathtt{br}\ bb(v_1,\ldots,v_{|in(bb)|})\text{“}}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(fst(bb),s[in(bb)_i \leftarrow s(v_i) \mid 1\le i\le |in(bb)|][loc(bb)_i \leftarrow \bot \mid 1\le i\le |loc(bb)|]),cs,m)} \quad \textbf{(goto)}$$

$$\frac{op(l) = \text{“}\mathtt{cond\_br}\ v\ bb1(v_1,\ldots,v_{|in(bb1)|})\ bb2(w_1,\ldots,w_{|in(bb2)|})\text{“} \qquad s(v)\notin\{0,\bot\}}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(fst(bb1),s[in(bb1)_i \leftarrow s(v_i) \mid 1\le i\le |in(bb1)|][loc(bb1)_i \leftarrow \bot \mid 1\le i\le |loc(bb1)|]),cs,m)} \quad \textbf{(ifthen)}$$

$$\frac{op(l) = \text{“}\mathtt{cond\_br}\ v\ bb1(v_1,\ldots,v_{|in(bb1)|})\ bb2(w_1,\ldots,w_{|in(bb2)|})\text{“} \qquad s(v)=0}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(fst(bb2),s[in(bb2)_i \leftarrow s(w_i) \mid 1\le i\le |in(bb2)|][loc(bb2)_i \leftarrow \bot \mid 1\le i\le |loc(bb2)|]),cs,m)} \quad \textbf{(ifelse)}$$

$$\frac{op(l) = \text{“}\mathtt{return}(v_1,\ldots,v_{Of})\text{“}}{(Run^{\mathtt{f}}(l,s),cs,m) \to (End^{\mathtt{f}}(s(v_1),\ldots,s(v_{Of})),cs,m)} \quad \textbf{(end)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = \mathtt{call}\ \mathtt{g}(w_1,\ldots,w_k)\text{“}}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Start^{\mathtt{g}}(s(w_1),\ldots,s(w_k)),Run^{\mathtt{f}}(l,s)::cs,m)} \quad \textbf{(call)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = \mathtt{call}\ \mathtt{g}(w_1,\ldots,w_k)\text{“}}{(End^{\mathtt{g}}(x_1,\ldots,x_n),Run^{\mathtt{f}}(l,s)::cs,m) \to (Run^{\mathtt{f}}(nxt(l),s[v_i \leftarrow x_i \mid 1\le i\le n]),cs,m)} \quad \textbf{(ret)}$$

$$\frac{op(l) = \text{“}w = \mathtt{load}(addr)\text{“} \qquad s(addr)\neq\bot}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(nxt(l),s[w \leftarrow m(s(addr))]),cs,m)} \quad \textbf{(load)}$$

$$\frac{op(l) = \text{“}\mathtt{store}(addr,v)\text{“} \qquad s(addr)\neq\bot}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(nxt(l),s),cs,m[s(addr) \leftarrow s(v)])} \quad \textbf{(store)}$$

Figure 3: SSA semantics. Memory access rules in green. Function call rules in red.

- A triple $Run^{\mathtt{f}}(pc,val)$ formed of the label pc of the operation to execute next (the program counter) and a partial valuation $val : \mathcal{V}^{\mathtt{f}} \to \overline{Int}$ of all the variables.

The execution state of an SSA *specification* is a triple (s,cs,m) formed of the state s of the function that is currently executing, a list cs of running states representing the call stack, and the current state $m : Int \to \overline{Int}$ of the memory. An *initial state* of the specification has the form $(Start^{\mathtt{f}}(\ldots),[],m)$, where m is the initial memory state and $\mathtt{f}$ the function that serves as execution entry point. A *final state* of the specification has the form $(End^{\mathtt{f}}(\ldots),[],m)$.

Program execution Transition rules are provided in Fig. 3. An *execution trace* of an SSA specification is any sequence of transitions starting in an initial state. Note that if $t = s_0 \to s_1 \to \ldots \to s_n \to \ldots$ is a trace, then any prefix $t_n = s_0 \to s_1 \to \ldots \to s_n$ is also a trace. We denote with $\le$ the prefix order on traces, meaning we can write $t_n \le t$.

Note that rules **(ifthen)** and **(ifelse)** define the behavior of a conditional branch only when the test variable is defined. When it is absent, execution cannot advance—it blocks. Execution also blocks when reaching a memory access rule with address $\bot$. Otherwise, execution can advance. This requires defining the semantics $[\![op\_id]\!]$ of all operations for $\bot$ inputs. Note that maximal traces (in the sense of $\le$) will either end in a final state, or when execution blocks, or never end.

Separation assumptions Our semantics separates, in both state expression and transition rules, the part corresponding to the SSA core from the extensions needed to represent function calls and memory. For instance, to consider only the core SSA semantics one simply has to remove call stack and memory terms from the specification state, and only consider the black rules of Fig. 3 (which do not access memory or call stack). Similarly, function call rules do not access the memory, and memory rules do not access the call stack.

2.4 Determinism and correctness

If we assume that all operations are deterministic (which in our formal framework amounts to assuming that $[\![op\_id]\!]$ are partial functions) the sequentiality of the SSA execution implies its determinism: For each initial state s there exists a unique trace starting in s that is maximal in the sense of $\leq$.

2.4.1 Dominance

But determinism is not sufficient. Correctness also requires that execution never blocks. To ensure this, we will enforce a stronger property: that all variables are different from $\perp$ whenever used in computations. Assume that variable v is assigned a value in basic block b, and that v is an input to operation o' of basic block b'. To ensure that v does not have value $\perp$ when o' is executed, one must ensure (as a necessary property) that any execution path reaching o' passes through the definition of v. This happens *if and only if* one of the following conditions is true:

D1 $b = b'$, and the definition of v precedes o' in b, either as block argument, or as output of an operation.

D2 $b \neq b'$, and any possible execution reaching b' necessarily passes through b.

While checking property **D1** is straightforward, determining that **D2** holds for any variable v and operation o using v is not tractable in the general case (Boolean satisfiability can be reduced to the decision of **D2**).

For this reason, SSA-based compilation always ensures **D2** by enforcing a sufficient property, named dominance, which can be checked using a low-complexity structural analysis of the SSA SCFG [7]. Dominance is considered part of SSA form correctness, along with the syntactic correctness and the structural properties of Section 2.1.

Together, these structural properties ensure the correctness of SSA specifications that do not access memory. When memory is used, these properties must be complemented by a proof of the fact that each memory location is initialized before it is read.

3 From SSA to synchronous concurrency

In this section we extend the syntax and semantics of SSA with the operational mechanisms needed to represent synchronous concurrency. The extension leaves the (non-reactive) SSA semantics of Fig. 3 unchanged, and allows the application of all SSA code transformations. Thus, building on MLIR's extensible syntax and semantics, it is possible to model and generate code for reactive real-time applications without changing the existing code base (only introducing additional behaviors).

3.1 Cyclic execution

To represent the behavior of embedded systems, which interact with their environment in a continual and timely fashion, all synchronous languages have *cyclic execution models*. The execution of a synchronous program is an *a priori* infinite *sequence of execution cycles*. Execution cycles being non-overlapping, they form a *logical time base*: each operation happens in exactly one cycle which can be identified by its index. But synchronous logical time is not only a descriptive notion used during analysis. It is meant to allow the synchronization of cycle execution onto external time bases. For instance, periodic real-time execution is typically enforced by synchronizing cycle triggering (the logical time base) with a periodic HW timer.

```
<type> += in(<type>) | out(<type>)
  <op> += (<var>?) = tick(<tvar>*)
        | <var> = sync(<tvar> <tvar>+)
        | <var> = input(<var>):<type>
        | (<var>?) = output(<var>:<var>):<type>
        | <var> = undef:<type>
<op_id> += inst <fun_name> <inst_id>
```

Figure 4: SSA syntax extensions for synchronous reactive programming. Grayed non-terminals are those of Fig. 1. "+=" extends an already-defined non-terminal.

The SSA form also allows the representation of cyclic behaviors under the form of cyclic SCFGs, as exemplified in Fig. 2. However, what constitutes an *execution cycle* is not clearly identified, nor the mechanisms for synchronizing execution cycles with an external time base, nor how to constrain an operation to happen in a specific cycle. To allow SSA-based embedded real-time programming, we must allow the specification of all these aspects *in a way that will be preserved by SSA-based code transformations and optimizations.*

Tying the definition of execution cycles to the structural elements of SSA (the basic blocks) is tempting, but unfeasible, as basic blocks are not preserved by common optimizations. Another tempting approach is to extend the syntax *and semantics* of basic blocks and/or branching statements to explicitly identify some of them as execution cycle barriers. However, such an approach would require changes to much of existing SSA-related compiler code (such as dominance analysis or optimization algorithms), which we want to avoid.

3.1.1 The `tick` operation

The solution we chose is the introduction of a new operation `tick` to identify execution cycle barriers. To order operations with respect to these barriers, `tick` relies on two SSA mechanisms: dominance and memory access ordering.

Dominance-based ordering. The syntax of `tick`, provided in Fig. 4, shows that it can take as arguments any number of variables, which allows specifying that operations producing these variables are executed before the cycle barrier. It produces an optional variable `%s` of type `none`, which is the unit type of MLIR SSA.

```
1  func @periodic1()->() {
2  ^reset:
3    %x0 = call @init() :()->tensor<64xi8>
4
5    br ^step(%x0:tensor<64xi8>)
6  ^step(%x1:tensor<64xi8>):
7    %x2 = addi %x1, %x1: tensor<64xi8>
8    %s = tick(%x2:tensor<64xi8>)
9    %x3 = sync(%x2:tensor<64xi8>, %s:none)
10   br ^step(%x3:tensor<64xi8>)
11 }
```

```
1  func @periodic2->() {
2  ^reset:
3      %x0 = alloc(): memref<64xi8>
4      call @init(%x0) : memref<64xi8> -> ()
5      br ^step
6  ^step:
7      call @do_addi(%x0): memref<64xi8> -> ()
8      tick()
9
10     br ^step
11 }
```

Figure 5: Synchronizing computations w.r.t. execution cycle barriers (`tick`) to enforce periodic execution. Dominance-based (left) vs. side-effect-based (right).

Like for the unit type of functional programming or the *pure signals* of synchronous programming [16], a variable of type `none` represents *pure synchronization*. It carries no information but requires operations reading it to happen after the operation producing it. To facilitate the specification of such ordering constraints, we also introduce the `sync` operation which allows transferring the dependences of one or more variables onto another. The operation takes as input two or more variables. It copies the value of its first input onto its output as soon as all inputs have arrived.

As Fig. 5(left) shows, this way of enforcing ordering is particularly useful early in the compilation process, when aggregate n-dimensional data are represented with abstract types (such as the *tensors* of MLIR) manipulated with side-effect-free operations.

Ordering by side-effects We also assume that `tick` has unspecified side-effects, which means it cannot commute (during SSA code transformations) with operations that read or write memory. This way of enforcing ordering is particularly useful later in the code generation process, once buffer allocation of aggregate data has been done. This is the case in Fig. 5(right), which could be an implementation of the program in Fig. 5(left). Note that in both cases SSA code transformations (e.g. loop unrolling) can be applied freely, without changing the ordering of computations w.r.t. execution cycle barriers.

Structural properties To ensure that the execution of a *reactive SSA function* is an infinite sequence of execution cycles, we will require that it does not contain `return` operations, and that each potentially unbounded cycle of the SSA specification contains at least one `tick` operation.

3.1.2 Compilation

The transformation of a standard SSA form specification into executable sequential code is a well-understood process. However, the introduction of `tick` fundamentally changes the SSA semantics by requiring a cyclic interaction with the environment/scheduler.

The way this interaction is traditionally implemented in the compilation of synchronous languages [16, 4, 3] is illustrated in Fig. 6. The control flow of the source code (function `@n` in our case) is completely restructured in order to produce a semantically equivalent program with a single `tick` operation (function `@n_drv`). To do this, the control and data state of the original function *between execution cycles* must be explicitly represented. In our example, this state is transmitted by the arguments of `^step` and consists of:

- A Boolean value (of type `i1`) to determine whether in the next cycle we need to start `@f` or `@g`.
- A value of type `i32` allowing the transmission of the output of `@f` to `@g`.

At the beginning of each cycle the state is decoded, the needed computations are triggered, then a new state is encoded and transmitted to the next cycle. This process is usually represented with a separate *step function*, in our case `@n_step`.

Classical synchronous language compilers will usually produce just this step function and the data structure describing the application state. When used in conjunction with the dataflow modularity described in Section 3.3, this approach allows modular code generation[4] [4]. However, the driver function (in our case `@n_drv`), and in particular the implementation of `tick`, are usually not generated, being considered too implementation-dependent.

[4]One step function per hierarchic synchronous module, the state representation of a module including that of sub-modules it hierarchically includes, and its step function calling the step functions of sub-modules.

```
1  func @n()->() {
2  ^start:
3    %x = call @f(): ()->(i32)
4    tick()
5    call @g(%x): (i32)->()
6    tick()
7    br ^start
8  }
```

```
1  func @n_step(i1,i32)->(i1,i32){
2  ^start(%s:i1,%x:i32):
3    cond_br %s, ^true,^false
4  ^true:
5    %x1 = call @f(): ()->(i32)
6    %false = constant 0: i1
7    return %false,%x1: (i1,i32)
8  ^false:
9    call @g(%x): (i32)->()
10   %true = constant 1:i1
11   return %true,%x: (i1,i32)
12 }
13 func @n_drv()->() {
14 ^start:
15   %s0 = constant 0 : i1
16   %x0 = constant 0 : i32
17   br ^step(%s0,%x0: i1,i32)
18 ^step(%s:i1, %x:i32):
19   %s1,%x1 = call @n_step(%s,%x): (i1,i32)->(i1,i32)
20   tick()
21   br ^step(%s1,%x1: i1,i32)
22 }
```

Figure 6: Synchronous languages approach to code generation. Source code on the left, output on the right.

This compilation approach has been long tested in practice [2, 11], where it has shown its strengths (most notably modularity), but also its limits. The limits are mainly related to the one-size-fits-all generated code with a single step function and a state representation that must cover the needs of *all* execution cycle transitions. In our example, the state value of type `i32` is computed and transmitted between every two successive cycles, even though it is semantically produced (output of `@f`) only in odd cycles. A good measure of the inefficiency of the resulting code is given by the significant amount of work on its optimization, and in particular on the optimization of its state representation[16, 10, 9]. It is important to note that, once the generation of the step function performed, classical compiler optimizations are confined to the scope of the step function, and optimizations involving multiple execution cycles must be specifically designed for each particular language and state encoding.

Our compilation method cannot follow this example and systematically restructure control to obtain a loop with a single `tick`. Not only because of the potential efficiency loss, but because in many cases *the implementation must have a different structure*. For instance, in avionics MIF/MAF applications [9] the implementation must have a structure similar to that of Fig. 6(left), where a global periodic pattern (the global loop, named *major frame*, or MAF) is split by the `tick` operations into time intervals of equal length (the *minor frames*, or MIFs), each containing a different code.

To allow the implementation, without restructuring, of any reactive SSA graph satisfying the structural properties defined above, we propose a return to the fundamentals of reactive systems design, by making explicit the interaction with the system scheduler. In our compilation approach, a reactive SSA specification such as function `@n` of Fig. 6(left) is seen as a sequential process running under a cooperative multi-tasking scheduler. Each time the execution of `@n` reaches a `tick` operation, the execution context (state) of `@n` is saved and control is given back to the scheduler. When the scheduler determines that a new execution cycle must be triggered, it restores the state of the process and gives it back the control. When this happens, execution of the `tick` operation terminates, and control is given in sequence.

This operational mechanism, whose formal semantics is defined in Section 3.5, can be easily implemented on various execution platforms ranging from low-level timers on bare metal platforms [9] to POSIX system services such as `longjmp` (as we do in Section 4.2) and to RTOS services such as `PERIODIC_WAIT` in the avionics-oriented IMA/ARINC 653 standard [1]. Most important, SSA code transformations and optimizations can be applied at their full strength while generating the executable code of the sequential processes.

Note that our proposal does not exclude the classical approach of compiling synchronous languages. Instead, it is complementary, allowing the modeling of implementation aspects that were previously not covered by code generation, and by allowing more expressiveness in the implementation.

3.2 Cyclic I/O

An embedded system will continually interact with its environment, cyclically reading inputs and writing outputs. In implementations, this is typically done by reading and writing memory-mapped registers that can be represented with volatile C variables, or by calling dedicated I/O functions. Synchronous languages abstract away such implementation-dependent mechanisms under the form of *input and output variables* that can be read or written at each cycle, with two constraints related to the synchronous model:

- An output variable can be written at most once per execution cycle.
- All reads of an input variable during an execution cycle must return the same result.

For instance, in Fig. 2, the input `inc` of the Lustre program is read at each execution cycle.

By comparison, when not considering memory side effects, MLIR functions interact with their environment only twice:

- At the beginning of their execution, to read the value of their input arguments, which then remains constant during the execution of the function.
- When reaching a `return` operation, when the function completes.

These assumptions enable common SSA optimizations such as loop-invariant code motion. Volatile accesses to memory locations can sometimes be represented, like in the low level dialects of MLIR (LLVM, SPIRV), but they represent a particular low-level implementation, excluding others.

The solution we chose to represent cyclic I/O is based on the representation of input and output *channels* with function arguments of the special types `in(t)` and `out(t)`, where `t` is the type of data transmitted by the channel. Access to channel variables is done exclusively using the `input` and `output` operations, whose syntax is provided in Fig. 4. Operation `input` has a single argument of input channel type. Each time it is executed, `input` samples the channel for a value of the correct type, which is placed in the output variable. An `output` operation has two inputs: one output channel and a second variable (of the corresponding non-channel type) whose value will be written to the output.

To specify ordering relations between I/O operations and other operations, we use the same mechanisms discussed for `tick` in Section 3.1.1. On both `input` and `output` we assume unspecified side-effects, which prevents reordering with other operations that access memory and function calls. Both I/O operations also allow dominance-based ordering. For this reason, `output` has an (optional) output of type `none`.

Structural properties It is required that only functions representing reactive behaviors use channel variables or the operations `input` and `output`. It is also required that reactive functions have only arguments of I/O channel type, and that no channel variable is output of an operation.

```
func @n(in(i32), out(i32))->() {
^start(%i:in(i32), %o:out(i32)):
  br ^step
^step:
  %x = input(%i):i32
  output(%o:%x):i32
  tick()
  br ^step
}
```

```
func @n(()->(i32), (i32)->())->() {
^start(%i:()->(i32), %o:(i32)->()):
  br ^step
^step:
  %x = call_indirect %i (): ()->(i32)
  call_indirect %o (%x): (i32)->()
  call @tick() : ()->()
  br ^step
}
```

Figure 7: Code generation for I/O operations

It is also required that at most one `input` or `output` operation is performed on a given I/O variable between two instances of `tick`. This semantic property is usually enforced by requiring the respect of structural properties, e.g. by requiring that every SCFG path between two operations on the same channel contains a `tick` operation.

Compilation As explained above, various low-level mechanisms can be used to implement the operations `input` and `output`, the most typical being volatile variables and calls to I/O functions. For portability, our compiler takes the second approach, by:

- Transforming each function argument of type `in(t)` into a function argument of type `()->(t)`, i.e. a reference to a function that takes no argument and produces one result of type `t`.
- Transforming each function argument of type `out(t)` into a function argument of type `(t)->()`.
- Each `input` and `output` operation is transformed into a call to the corresponding function argument. Note the use of `call_indirect`, a version of operation `call` allowing calling a function transmitted by reference.

Fig. 7 provides a small example: a reactive program with one input channel and one output channel that copies at each cycle its input on the output.

When the value produced by `input` or taken as input by `output` is an object stored in memory,[5] much care must be exerted to avoid memory errors (accessing unallocated memory and memory leaks). In our implementation, we will assume that all memory-stored objects produced by an `input` operation are allocated and deallocated by the environment, with a lifetime finishing at the end of the current execution cycle. We also assume that memory-stored objects given as argument to an `output` operation are allocated and deallocated by the function, and that the environment no longer uses them when the next execution cycle begins.

3.3 Modularity

The modularity of SSA is that of sequential procedural programming. The modules of an SSA specification are the functions, which interact through *function calls*.

By comparison, the formal models underpinning synchronous languages are concurrent. In the most general settings, such as Esterel's constructive semantics [16], *the execution of two sub-modules of a specification can advance concurrently, synchronizing and communicating with each other in both senses*. Determining that the execution of such a specification does not block (a process known as *causality analysis*) is in general undecidable, if integer data are allowed, and NP-hard (untractable in practice) if the input language uses only Boolean variables. Furthermore,

[5] Like the variables of type `memref` in MLIR.

the implementation of such general specifications can be very inefficient due to intricate semantic rules.

For this reason, synchronous language compilers have early on imposed simple structural constraints allowing fast and modular compilation: In each module, the computations of an execution cycle must form an acyclic dependency graph, allowing fast scheduling and code generation. In this acyclic graph, to allow the separate compilation of a sub-module, its computations must be grouped together as a single graph node, meaning that they can be performed *atomically*.

To allow the representation of this mechanism in our SSA extension, we introduce the notion of *instance* of a reactive function `@f`, which is a process (with separate state) executing function `%f` under the system scheduler. Instances are uniquely identified (with lists of strings, in our implementation). We assume that the first reactive function of a specification has an instance identified with the empty list of strings `[]` that receives control when the system starts. All other instances are inductively defined and possibly given control during execution using the operation `inst` (syntax in Fig. 4): if `i` is an instance of function `@f` which contains operation "`inst @g str`", then instance `str::i` of function `@g` is automatically defined.

We provide in Fig. 8 an example of submodule instantiation. The system has two reactive functions and two instances: the implicit `[]` instance of function `@main` and one instance `["a"]` of function `@sum`. Instance `[]` reads its unique input from the environment on odd cycles, triggers one tick of instance `["a"]` in every cycle (giving it as input the value of `%i`) and outputs the last output of `["a"]` in even cycles. Instance `["a"]` of `@sum` has a state in which it accumulates the sum of inputs it receives on odd cycles. On even cycles, it computes and outputs variable `%o`.

Structural constraints The number and types of input and output variables of an `inst` operation must match the signature of the reactive function it instantiates.

Compilation The compilation method based on conversion to step functions will implement sub-module instantiation using function calls [4]. When conversion to a step function is not desired, we transform each instance into a process running under a cooperative scheduler. This mechanism extends that of Section 3.1.2 by clarifying how the scheduler passes the control between instances. When reaching an `inst` operation, the inputs and the control are transmitted to the instance, which then executes until reaching a `tick` operation, at which point it saves its state and returns the outputs to the caller, according to the rules of Section 3.5.

3.4 Signal absence

Consider the simple example in Fig. 6(left) and its implementation on the right. Note that the translation slightly changes the semantics of the program: On the left, the variable `%x` is transmitted only from odd cycles to even cycles. But in the translation result `%x` has been added to the program state, which is computed and transmitted at every cycle. Thus, in the `^false:` basic block of `@n_step`, under SSA semantics, we need to return a value for `%x`, even if the source program does not require it.

In this case, the choice is natural: we maintain the previous value, which may later allow an encoding of the state in (persistent) memory. But using a constant of type `i32` or a random value instead would have been correct, as this value will never be used.

But the choice is less obvious in other cases, such as for communication variables. In Fig. 8, function `@sum` outputs values only on even cycles, meaning that the value of `%x1` in function `@main` is never correctly initialized (yet, the specification is overall deterministic, because this *absent/undefined* value is never used in computations).

Such situations are common in the synchronous modeling of multi-periodic systems, which explains why *absence* prominently figures in the semantics of all synchronous languages [2].

```
1  func @main(in(i32), out(i32))->(){
2  ^start(%ic:in(i32), %oc:out(i32)):
3    br ^step
4
5  ^step:
6    %i = input(%ic):i32
7    %x1 = inst @sum "a" (%i:i32)
8    tick()
9    %x2 = inst @sum "a" (%i:i32)
10   output(%oc:%x2):i32
11   tick()
12   br ^step
13 }
```

```
1  func @sum(in(i32),out(i32))->(){
2  ^start(%ic:in(i32),%oc:out(i32)):
3    %s0 = constant 0 : i32
4    br ^step(%s0:i32)
5  ^step(%s:i32):
6    %i = input(%ic):i32
7    %s1 = addi %s, %i : i32
8    tick(%s1:i32)
9    %o = call @f(%s1):(i32)->(i32)
10   output(%oc:%o) : i32
11   tick()
12   br ^step(%s1:i32)
13 }
```

Figure 8: Submodule instantiation example

To allow its explicit representation in SSA without breaking the structural correctness rules of SSA, we introduce operation `undef` (syntax in Fig. 4) which explicitly leaves its output variable absent/undefined. We could use it, for instance, in Fig. 6, function `@n_step`, basic block `^false`, in order to leave the second return argument uninitialized.

For scalar values, `undef` is **compiled** into the `llvm.undef` value of LLVM IR [12]. For aggregate data allocated to memory, it translates into a lack of initialization after allocation.

Operation `undef` has not one, but two **semantic** interpretations, both covered by the rule of Fig. 9:

Absence In this case, $\alpha = \bot$, the semantic value introduced in Section 2.3.

Undefinedness In this case, $\alpha = *$, where $*$ is interpreted as a random integer value.

Both interpretations are fundamental. The second one is that of the implementation (and of LLVM IR), where the hardware locations always store values of type *Int* (even if we do not know which ones). In this interpretation, the execution of an SSA specification will always start with a random, yet fully defined, memory state, and SSA correctness ensures that, under semantic execution, no $\bot$ value is ever used in computations or stored to memory.

$$\frac{op(l) = \text{“}v = \texttt{undef}\text{“}}{(Run^{\mathsf{f}}(l,s),cs,m) \rightarrow (Run^{\mathsf{f}}(nxt(l),s[v \leftarrow \alpha]),cs,m)} \textbf{ (undef-}\alpha\textbf{)}$$

Figure 9: Semantics of $\bot$

$$\frac{\mathcal{I}(i) = (s, cs, si, so) \quad (s, cs, m) \to (s', cs', m')}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (s', cs', si, so)], m' >} \textbf{(local)}$$

$$\frac{\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{“}v = \mathtt{input}(ic)\text{”}}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s[v \leftarrow si(ic)]), cs, si, so)], m >} \textbf{(in)}$$

$$\frac{i = [] \quad \mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{“}\mathtt{tick}\text{“}}{< i, \mathcal{I}, m > \overset{so}{\underset{si'}{\Rightarrow}} < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s), cs, si', \alpha)], m >} \textbf{(s-tick-}\alpha\textbf{)}$$

$$\frac{\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{“}\mathtt{output}(oc : v)\text{”}}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s), cs, si, so[oc \leftarrow s(v)])], m >} \textbf{(out)}$$

$$\frac{\mathcal{I}(i' :: i) = (Run^{\mathtt{g}}(l', s'), cs', si', so') \quad \mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{“}(v_1, \ldots, v_n) = \mathtt{inst\ nd}\ i'\ (w_1, \ldots, w_k)\text{”}}{< i, \mathcal{I}, m > \Rightarrow < i' :: i, \mathcal{I}[(i' :: i) \leftarrow (Run^{\mathtt{g}}(l', s'), cs', \bot[in_l^{\mathtt{nd}} \leftarrow s(w_l) \mid 1 \leq l \leq k], \alpha)], m >} \textbf{(inst-}\alpha\textbf{)}$$

$$\frac{\mathcal{I}(i' :: i) = (Run^{\mathtt{g}}(l', s'), cs', si', so') \quad op(l') = \text{“}\mathtt{tick}\text{“} \quad \mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{“}(v_1, \ldots, v_n) = \mathtt{inst\ nd}\ i'\ (w_1, \ldots, w_m)\text{”}}{< i' :: i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(l, s[v_i \leftarrow so'_i \mid 1 \leq i \leq n]), cs, si, so)], m >} \textbf{(tick)}$$

Figure 10: SSA semantics extension for reactive synchronous systems. In red, modularity rules.

The first interpretation is that of the abstract interpretation meant to determine whether uninitialized values are used in computations. In this interpretation, the execution of an SSA specification will start with $\perp$ memory state. The
execution of `undef` operations, access to uninitialized memory locations, or use of $\perp$ function arguments will set to $\perp$ variables that can afterwards be used in computations.

The two interpretations are related by a strong abstraction result. To define it, we organize $\overline{Int}$ as a Scott domain under the partial order defined by $\perp \leq x$ for all $x \in Int$. This partial order is extended pointwise on the variable states of an SSA function and on memory states. It is also naturally extended to function states by $Start^{\mathtt{f}}(v_1, \ldots, v_k) \leq Start^{\mathtt{f}}(w_1, \ldots, w_k)$ and $End^{\mathtt{f}}(v_1, \ldots, v_k) \leq End^{\mathtt{f}}(w_1, \ldots, w_k)$ if $v_i \leq w_i$ for all i, and $Run^{\mathtt{f}}(l, s) \leq Run^{\mathtt{f}}(l, s')$ if $s \leq s'$. It is also extended to call stacks by $[s_1, \ldots, s_k] \leq [s'_1, \ldots, s'_k]$ if $s_i \leq s'_i$ for all i. With these definitions, we can extend pointwise the partial order $\leq$ to execution states of an SSA specification.

Theorem Consider a non-reactive SSA specification.[6] Assume that all operation identifiers have monotonous semantics $[\![op\_id]\!] : \overline{Int} \to \overline{Int}$. Let $s^a_0 \to \ldots \to s^a_n$ be an execution trace under the abstract semantics of `undef`, and $s^c_0 \to \ldots \to s^c_m$ an execution trace under its low-level semantics such that $s^a_0 \leq s^c_0$. Then: 1) if $m < n$, the low-level trace can be continued to a trace of length n, and 2) $s^a_i \leq s^c_i$ for all $1 \leq i \leq min(m, n)$.

Proof sketch Each of the semantic rules of Figures 3 and 9 is monotonous in the sense of the Scott domain partial order on specification states. Then, each execution step will preserve the abstraction relation $s^a_i \leq s^c_i$ between abstract and low-level (concrete) states.

A corollary of this theorem is that guarantees obtained through analyses under the abstract semantics[7] remain valid under low-level (implementation) semantics.

3.5 Formal semantics of reactive extensions

The final step of our SSA extension is the definition of the formal semantics of reactive SSA specifications formed of one or more instances of reactive SSA functions.

Given a reactive function `f`, we denote with $in^{\mathtt{f}}$, respectively $out^{\mathtt{f}}$ the ordered set of input channel variables of `f`. Given an instance i, we denote with $r(i)$ its reactive function.

The **execution state** of a reactive SSA specification is represented with triples $< i, \mathcal{I}, m >$, where m is the shared memory state, i is the currently active instance identifier, and $\mathcal{I}$ is a map associating to each instance identifier the instance state. The state of instance i is $\mathcal{I}(i) = (s, cs, si, so)$, where s is the state of the function `f` that is currently executing, cs is the call context, $si : in^{r(i)} \to \overline{Int}$ is the state of the input channels of i, and $si : out^{r(i)} \to \overline{Int}$ is the state of the output channels.

Under these notations, the operational semantic rules are provided in Fig. 10. The (local) rule transforms non-reactive SSA transitions of the instances (denoted with $\to$ and defined in Figures 3 and 9) into transitions (denoted with $\Rightarrow$) of the reactive system. All other rules involve reactive operations (interaction with the scheduler). Rules (in) and (out) deal with instance I/O. Rule (s-tick-α) is the only one that interacts with the environment by performing I/O and possibly time synchronization. Note that, like the (undef-α) rule of Fig. 9, it has two interpretations ($\alpha = \perp$ or $\alpha = *$). The rules (inst-α) and (tick) (in red) implement modularity.

[6] That may use `undef`, but not `tick`, cyclic I/O, or synchronous modularity.

[7] That a program does not block or that a given variable or memory location has a specific (defined) value at a specific point during execution.

4 Evaluation

In the previous section, our objective was to extend the SSA form with synchronous reactive features in the most general way that remains fully compatible with traditional SSA semantics and algorithmics (thus allowing implementation without changes to the existing codebase).

In this section, we evaluate the ability of this extension to support the specification and compilation of *realistic* applications with both reactive and HPC aspects. The reactive aspects of such applications must be specified in a high-level synchronous dialect, not directly in our low-level SSA extension. We show that the dataflow core of the Lustre synchronous language[8] can be embedded as a new *dialect*, named `lus`, into the SSA-based MLIR compilation framework [14]. This allows the specification (in MLIR) of applications where the reactive aspects are modeled at the Lustre abstraction level, while data processing is modeled at the abstraction level of other dialects such as `affine` (affine loops), `linalg` (linear algebra), or `tf` (TensorFlow graphs).

During compilation, reactive statements of the `lus` dialect are *lowered*[9] into a mix of structured control flow (MLIR dialect `scf`) and the reactive SSA constructs introduced in Section 3 and grouped in a new dialect named `sync`. Operations of the `sync` dialect are later converted into low-level SSA (dialects `std` and `llvm`) and calls to external OS primitives following the compilation rules defined in Section 3.

The structured control flow, along with the data types and data processing code, are progressively lowered using the existing transformations of MLIR, which do not affect reactive semantics. Among others, these transformations allow *buffer synthesis*, i.e. the transformation of the abstract aggregate data used with `lus`-level synchronous specification (e.g. tensors) into memory objects along with allocation and deallocation operations, in a way that ensures the absence of both memory leaks and accesses to unallocated memory.

The result is a **fully functional specification and compilation framework for reactive high-performance applications**, which we evaluated on a non-trivial signal processing (vocoder) application.

4.1 Embedding Lustre in MLIR

The synchronous language that has reached the most widespread use is Lustre [11, 3]. For space reasons, we only consider here its pure dataflow core—the SN-Lustre dialect of [5]—into which full Lustre can be translated.

A Lustre program, like that of Fig. 2(top), is called a *node*. It specifies a dataflow graph of statements connected through dataflow variables. Each variable is either an input of the node, or it is output of exactly one statement (*single assignment property*). Lustre follows a *pure dataflow* paradigm, with no use of load/store memory (no side-effects).

The semantics of Lustre is (intuitively) best described as a mix of dataflow operational and declarative aspects.

Operational interpretation As synchronous programs, Lustre nodes have a cyclic execution model. At each execution cycle, the list of statements of a node is traversed *once*, in an order compatible with the dependences determined by the variables. When traversed, a statement will read the value of its input variables, possibly perform some internal computations, and assign a value to its output variables. As the semantics of a node is not affected by the syntactic order of its statements, we can always assume (as a normal form assumption) that the statements already are in traversal order, as in Fig. 2(top).

[8] Chosen for its practical importance, as well as for its simplicity.
[9] I.e. transformed into code at a lower abstraction level.

```
1  node s(i:int)returns(s:int)
2  var ck:bool;i1,s1:int;
3  let
4
5    ck = (i>0) ;
6    i1 = i when ck ;
7    s1 = 0 fby s ;
8    s  = s1 + i1 ;
9
10 tel
```

```
1  node sn(i:int)returns(s:int)
2  var ck:bool;i1,s1,si,so:int;
3  let
4    si = 0 fby so;
5    ck = (i>0);
6    i1 = i when ck;
7    s1 = si when ck;
8    s  = s1 + i1;
9    so = merge ck s (si whenot ck);
10 tel
```

Figure 11: Time modularity (left) and its normalization

Dependences are defined as follows: Variable `y` being produced by statement p and used in statement c determines a dependency $p \to c$ in all cases except one: when c is a statement of the form "`x = k fby y`". In this case, $p \to c$, as a form of anti-dependency ensuring that the value is read before being overwritten. This special handling of `fby` allows it to read the value assigned to its input variable *in the previous execution cycle*. This value is then assigned to the output of `fby`, allowing its use in the current cycle. In Lustre, this is the only mechanism allowing the specification of a *state* passed between execution cycles.

The value assigned to a variable in a cycle can be the special value $\perp$ introduced in Section 2.3 to represent *absence*. Making a variable absent inside an execution cycle is done using the subsampling statement `when`. When statement "`x = y when c`" is traversed and the Boolean `c` has value true, `x` is assigned the value of `y`. If `c` is false or $\perp$, `x` is assigned value $\perp$. Absence is the dataflow mechanism of specifying conditional execution: A function call with $\perp$ input variables will not execute its function, but instead assign all outputs to $\perp$. This mechanism, specific to the dataflow programming paradigm, is used in lines 9 and 10 of Fig. 2 to specify that function `actuate` is executed only in cycles where variable `ck` is true. The counterpart of `when` is statement "`x = merge c y z`". Upon traversal, if `c` is true (resp. false), `x` takes the value of `y` (resp. `z`). Otherwise, `x` takes the value $\perp$.

Declarative aspects (clock constraints) The purely operational interpretation defined above matches the standard Lustre semantics [3, 4, 5] on correct Lustre programs where all `fby` statements are executed at every cycle.

However, Lustre allows a form of logical time modularity during specification, by allowing subsets of statements (including `fby` statements) to be executed under specific Boolean conditions. This is the case in the node of Fig. 11(left), which incrementally computes the sum of its *positive* input values. The statements in lines 7 and 8, which implement the computation of the sum, are executed only in cycles where `ck` is true. However, no operational mechanism constrains the execution of the `fby` statement.

Instead, each Lustre statement defines a *clock constraint*—a predicate relating the present/absent status of its input and output variables (and possibly their value) inside an execution cycle. In our example, two such clock constraints apply:

- Inputs and outputs of a function call or `fby` are either all $\perp$, or all have a value different from $\perp$.
- The inputs of a `when` are either both present, or both absent. Output is present *iff* the condition is true.

Determining the execution condition of each statement consists in solving the system of con-

```
node @s2(%i:i32)->(int){

  %c0=constant 0:i32

  %si=fby %c0,%so:i32

  %ck=cmpi "sgt",%i,%c0:i32
  %i1=when %ck,%i:i32
  %s1=when %ck,%si:i32

  %s=addi %s1,%i1:i32

  %s2=when not %ck,%si:i32
  %so=merge %ck,%s,%s2:i32
  lus.yield %s:i32

}
```

```
func @s2(in(i32),out(i32))->(){
^start(%ic:in(i32),%sc:out(i32)):
  %c0=constant 0:i32
  br ^step(%c0:i32)
^step(%si:i32):
  %i = input(%ic):i32
  %ck=cmpi "sgt",%i,%c0:i32
  cond_br %ck,^aux(%i:i32,%si:i32), ^out(%si:i32)

^aux(%i1:i32,%s1:i32):
  %s=addi %s1,%i1:i32
  output(%sc:%s):i32
  br ^out(%s:i32)
^out(%so:i32):
  tick()
  br ^step(%so:i32)
}
```

Figure 12: Lowering `lus` to `sync` in MLIR

straints associated to all statements. This process,[10] called *clock inference* and performed in formal settings called *clock calculi*, has been the subject of extensive prior research [4, 11, 2].

The `lus` dialect We have embedded the Lustre language constructs into MLIR. The `lus` representation of the Lustre node in Fig. 11(right) is provided in Fig. 12(left), with the `lus` keywords highlighted in blue. The MLIR definition of nodes is derived from that of SSA functions by replacing keyword `func` with `node` and requiring the compiler to check that a single basic block is present, terminated with operation `lus.yield` (which identifies the output variables).

While MLIR is SSA-based and its transformations mostly require the respect of dominance, it also allows specification under the weaker single assignment property, thus allowing the representation of cyclic dataflow graphs like the one in Fig. 12(left).

At the level of `lus`, we have implemented the clock inference algorithm. Along with data type correctness, single assignment, and causal correctness, the success of clock inference guarantees the correctness of a Lustre program. As the first two properties are automatically checked by the existing MLIR infrastructure, only causal correctness remains to be checked.

Normalization Once clock inference performed, we can transform the original program so that the operational interpretation correctly simulates it. This transformation, exemplified in Fig. 11, consists in ensuring that all `fby` statements are executed at every cycle (through the introduction of `when` and `merge` statements, in red in Fig. 11). We have implemented this normalization step in MLIR.

Lowering of `lus` to `sync` Once normalization performed, all `fby` operations can be collectively replaced with the continuation passing encoding of state specific to MLIR SSA. In Fig. 12(right) this is done using the basic branching mechanism of SSA, but the lowering phase we implemented in MLIR produces structured control flow operations (`for`). Once the state reencoding performed, determining the causal correctness of the program amounts to an SSA dominance check, performed automatically by MLIR.

The remaining lowering steps are the transformation of I/O variables of the `lus` node into I/O channels and explicit `input` and `output` operations, and the synthesis of imperative coditional control to ensure that each operation is only executed in cycles prescribed by clock inference. In

[10] Which will fail for incorrect programs.

Fig. 12(right) this imperative control is implemented using core SSA branching statements, but our MLIR implementation uses structured control flow operations.

This completes the definition of our compilation method. Note that, in the implementation of our compiler, stock MLIR algorithms are used for all data type specification and implementation, causality analysis, and memory allocation. We have only had to implement analysis and lowering steps specific to the synchronous model of computation.

4.2 The pitch tuning vocoder use case

We illustrate the expressiveness our proposal and evaluate its effectiveness on a complex real-time sound processing application—a voice pitch tuning vocoder. The application is naturally modeled under a synchronous paradigm. It must cyclically sample the sound input and the other control inputs, update the control state, perform one step of the pitch tuning algorithm, and (if required by the current control status) drive the sound output.

The application has a complex data processing pipeline, involving Hann filering to reduce spectral leakage, move to frequency domain using the Fast Fourier transform (FFT), move to polar coordinates to separate magnitude from phase, the additive phase synthesis algorithm that performs the actual pitch change, move back to Cartesian coordinates and then to the time domain using the inverse FFT, and a final step of Hann filtering and additive accumulation. All these aspects are specified using the abstract aggregate types of MLIR (tensors) and operations of the `linalg`, `scf`, and `std` dialects.

The control of this pipeline is also complex. It involves multiple sliding windows and multiple feedback loops. In addition to the pipeline control, the application also features control related with the user interface, silencing the output (which involves shutting down unneeded pipeline steps) and altering the pitch correction. All of these are intuitively specified using the `lus` dialect.

Our compilation flow automatically performs buffer allocation and synthesizes low-level control ensuring the correctness and safety of the implementation. Various optimizations can be applied in the process, including loop fusion.

5 Related work

We advance the state-of-the-art in SSA by providing the syntactic constructs, structural rules, and semantics extensions allowing the representation of synchronous reactive behavior. This overall objective is fully original.

However, particular aspects of our extension have been covered in previous work, in particular predicated execution [6, 15] and the use of semantic absence/undefinedness [12, 8]. In previous SSA work, predicated execution is *explicit*: the predicates are represented at all points where they exert control. We allow an *implicit* predication specification, akin to dataflow semantics, where the absence of inputs determines absence of execution. This requires a clear definition of correctness, which is absent in [6].

Our treatment of semantic absence/undefinedness is also different from that proposed for LLVM [12]. Our objective is not to ensure that non-deterministic code (involving undefined values) preserves its set of traces unchanged under various optimizations. Instead, it is to guarantee that well-defined values are preserved by various optimizations, even in the presence of undefined behaviors affecting other variables. We attain this goal through the abstraction theorem of Section 3.4.

We also extend previous work on the compilation of synchronous languages [16, 4, 2]. Existing synchronous language compilers profoundly restructure the code in order to match an execution model (which, as explained in Section 3.1.2, is in most cases sequential function calls, even though

multi-threaded implementations have also been proposed [13]). We fully avoid this by ensuring that every reactive SSA specification can be executed *as is*, and be subject to any correct SSA transformation. Synchronous-specific code transformations are still possible, but not mandatory.

More important maybe, instead of advancing the compilation of synchronous languages as a separate research field, we show that it can be seen as an extension of classical compilation, allowing the reuse of fundamental techniques for type checking, causality analysis, buffer allocation *etc.*

6 Conclusion

We presented an embedding of the Lustre synchronous reactive language into SSA, extending the MLIR framework to allow synchronous programming and code generation. We illustrated it by capturing the data processing, computational and reactive control aspects of a signal processing application. Our MLIR extension remains fully compatible with SSA-based analysis and transformations, while also capturing the synchronous composition of concurrent state machines, logical and physical time synchronization, and compilation passes specific to synchronous languages, such as clock calculus, causality analysis, bounded-memory and clock-directed code generation.

While there had long been connections between the semantics of functional languages, SSA and dataflow synchronous languages, this paper describes the concrete extension of SSA capturing the necessary elements for reactive control system. Furthermore, retaining all SSA-based compilation algorithms and reusing existing code unaware of the the specific synchronous concurrency semantics allows a tighter integration of reactive system modeling frameworks with the computationally intensive and general-purpose capabilities of MLIR- and LLVM-based frameworks for HPC and AI.

Since synchronous concurrency is highly popular in safety-critical environments, one important future direction is to fully formalize the core MLIR components our extension relies upon, and to prove its correctness. A related direction consists in exploring syntax and refinements for state-machines and control automata expressed in the Esterel synchronous language as well as hardware design languages such as Bluespec and Chisel. It is also important to further investigate which SSA-based algorithms can benefit the compilation flow of reactive control systems and how domain-specific methods for synchronous languages can enable greater automation and guarantees in the composition and memory management of concurrent systems in general.

References

[1] Arinc 653: Avionics application software standard interface. part 1 - required services. revision 3, 2010.

[2] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., LeGuernic, P., and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE 91*, 1 (Jan 2003).

[3] Bergerand, J., Caspi, P., Pilaud, D., Halbwachs, N., and Pilaud, E. Outline of a real time data flow language. In *Proceedings RTSS* (San Diego, CA, USA, December 1985).

[4] Biernacki, D., J.-L. Cola c., Hamon, G., and Pouzet, M. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings LCTES* (2008).

[5] Bourke, T., Brun, L., Dagand, P.-E., Leroy, X., Pouzet, M., and Rieg, L. A formally verified compiler for lustre. In *Proceedings PLDI* (2017).

[6] Carter, L., Simon, B., Calder, B., Carter, L., and Ferrante, J. Predicated static single assignment. In *Proceedings PACT* (1999).

[7] Cytron, R., Ferrante, J., Rosen., B., Wegman, M., and Zadeck, F. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (1991).

[8] Demange, D., and Y. Fernández de Retana, D. P. Semantic reasoning about the sea of nodes. In *Proceedings CC* (Vienna, Austria, 2018).

[9] Didier, K., Potop-Butucaru, D., Iooss, G., Cohen, A., Souyris, J., Baufreton, P., and Graillat, A. Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware. *ACM Trans. Archit. Code Optim. 16*, 3 (2019), 24:1–24:27.

[10] Gérard, L., Guatto, A., Pasteur, C., and Pouzet, M. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In *Proceedings LCTES* (2012).

[11] Halbwachs, N. A synchronous language at work: the story of Lustre. In *Proceedings Memocode* (Verona, Italy, 2005).

[12] Lee, J., Kim, Y., Song, Y., Hur, C.-K., Das, S., Majnemer, D., Regehr, J., and Lopes, N. Taming undefined behavior in llvm. In *Proceedings PLDI* (2017).

[13] Li, X., and von Hanxleden, R. Multithreaded reactive programming-the kiel esterel processor. *IEEE Transactions on Computers 61*, 3 (2012).

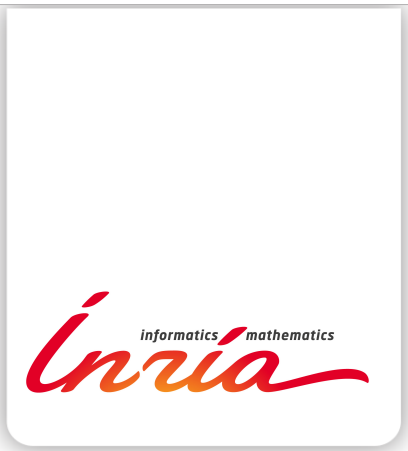
[14] Multi-level intermediate representation compiler framework (MLIR). `https://mlir.llvm.org/`, Retrieved on 11/17/2020.

[15] Ottenstein, K., Ballance, R., and MacCabe, A. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not. 25*, 6 (June 1990), 257–271.

[16] Potop-Butucaru, D., Edwards, S., and Berry, G. *Compiling Esterel*. Springer, 2007.

[17] Rosen, B., Wegman, M., and Zadeck, F. Global value numbers and redundant computations. In *Proceedings POPL* (Jan 1988).

[18] Static single assignment book (in progress). `http://ssabook.gforge.inria.fr/latest/book.pdf`, Retrieved on 11/17/2020.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399