ARx: Reactive Programming for Synchronous Connectors

José Proença[1] and Guillermina Cledou[2]

[1] CISTER, ISEP, Portugal
[2] HASLab/INESC TEC, Universidade do Minho, Portugal
pro@isep.ipp.pt, mgc@inesctec.pt

Abstract. Reactive programming languages (RP) and Synchronous Coordination languages (SC) share the goal of orchestrating the execution of computational tasks, by imposing dependencies on their execution order and controling how they share data. RP is often realised as libraries for existing programming languages, lifting operations over values to operations over streams of values, and providing efficient solutions to manage how updates to such streams trigger reactions, i.e., the execution of dependent tasks. SC is often realised as a standalone formalism to specify existing component-based architectures, used to analyse, verify, transform, or generate code. These two approaches target different audiences, and it is non-trivial to combine the programming style of RP with the expressive power of synchronous languages.
This paper proposes a lightweight programming language to describe component-based Architectures for Reactive systems, dubbed *ARx*, which blends concepts from RP and SC, inspired mainly on the Reo coordination language and its composition operation, and with constructs tailored for reactive programs such as the ones found in ReScala. This language is enriched with a type system and with algebraic data types, and has a reactive semantics inspired in RP. We provide typical examples from both the RP and SC literature, illustrate how these can be captured by the proposed language, and implemented a web-based prototype tool to edit, parse, and type check programs, and to animate their semantics.

1 Introduction

This paper combines ideas from *reactive programming languages* (RP) and from *synchronous coordination languages* (SC) into a new reactive language that both enriches the expressiveness of typical reactive programs and facilitates the usage of typical synchronous coordination languages.

Reactive programming languages, such as Yampa [14], ReScala [11], and Angular[3], address how to lift traditional functions from concrete data values to streams of values. These face challenges such as triggering reactions when these streams are updated, while avoiding glitches in a concurrent setting (temporarily

[3] https://angular.io/

inconsistent results), distinguishing between continuous streams (always available) and discrete streams (publishing values at specific points in time), and avoiding the callback hell [15] resulting from abusing the observable patterns that masks interactions that are not explicit in the software architecture.

Synchronous coordination languages, such as Reo [2], Signal Flow Graphs [7], or Linda [9], address how to impose constraints over the interactions between software objects or components, restricting the order in which the interactions can occur, and where data should flow to. These face challenges such as how to balance the expressivity of the language—capturing, e.g., real-time [16], data predicates [18], and probabilities [3]—with the development of tools to implement, simulate, or verify these programs.

Both programs in Reactive Programming (RP) and Synchronous Coordination (SC) provide an architecture to reason about streams: how to receive incoming streams and produce new outgoing ones. They provide mechanisms to: (1) calculate values from continuous or discrete data streams, and (2) to constraint the scheduling of parallel tasks. RP is typically more pragmatic, focused on extending existing languages with constructs that manage operations over streams, while making the programmer less aware of the stream concept. SC is typically more fundamental, focused on providing a declarative software layer that does not address data computation, but describes instead constraints over interactions that can be formally analysed and used to generate code.

This paper provides a blend of both worlds, proposing a language—ARx—with a syntactic structure based on *reactive programs*, and with a semantics that captures the synchronisation aspects in *synchronous coordination programs*. It starts by providing an overview of the toolset supporting ARx in Section 3, available both to use as a web-service[4] or to download and run locally. The rest of the paper formalises the ARx language, without providing correctness results and focusing on the tools. It starts by presenting the core features of ARx in Section 4, introducing an intermediate language to give semantics to ARx of so-called stream-builders and providing a compositional encoding of ARx into stream-builders. Two extensions to ARx are then presented. The first is algebraic data types, in Section 5, making the data values more concrete, and the second is a reactive semantics in Section 6, introducing new constructs to ARx and to stream-builders, and new rules to the operational semantics of ARx.

2 Overview over reactive and synchronous programs

This section selects a few representative examples of reactive programs (RP) and of synchronous coordinators (SC). It uses a graphical notation to describe these programs, partially borrowed from Drechsler et al. [11], and explains the core challenges addressed by both approaches.

Reactive programs (RP) Figure 1 includes 3 example of reactive programs:

[4] http://arcatools.org/#arx

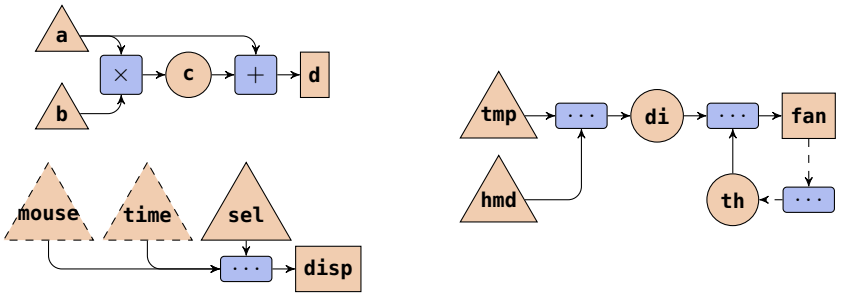

Figure 1: Example of typical Reactive Programs.

(top-left) A simple arithmetic computation, used by Bainomugisha et al. [5], with the program "$c = a \times b; d = a + c$" using reactive variables.

(right) A controller of a fan switch for embedded systems, used by Sakurai et al. [19], with the program "$di = 0.81 \times tmp + 0.01 \times hmd \times (0.99 \times tmp - 14.3) + 46.3$; $fan = di >= th$; $th = 75 + \mathsf{if}\ fan@last\ \mathsf{then}\ -0.5\ \mathsf{else}\ 0.5$".

(bottom-left) A GUI manager that selects which information to display, either from the continuous stream of mouse coordinates, or from the continuous stream of current time, with the program '$disp = \mathsf{if}\ sel\ \mathsf{then}\ \underline{mouse}\ \mathsf{else}\ \underline{time}$".

Consider the arithmetic computation example. It has 4 (reactive) variables, a, b, c, d, and the sources (depicted as triangles) may *fire* a new value. Firing a new value triggers computations connected by arrows; e.g., if b fires 5, the $\boxed{\times}$ operation will try to recompute a new product, and will update c, which in turn will *fire* its new value. So-called *glitches* can occur, for example, if a fires a value, and $\boxed{+}$ is calculated with the old value of c (before $\boxed{\times}$ updates its value). Different techniques exist to avoid glitches, by either enforcing a scheduling of the tasks, or, in a distributed setting, by including extra information on the messages used to detect missing dependencies. Languages that support reactive programming often includes operations to fire a variable (e.g., `a.set("abc")` in ReScala), to react to a variable update (e.g., `d.observe(println)` in ReScala), to ask for a value to be updated and read (e.g., `d.now` in ReScala), and to read or update a value without triggering computations. Hence the effort is in managing the execution of a set of tasks, while buffering intermediate results, and propagate updates triggered by new data.

Consider now the fan controller. It includes a loop with dashed arrows, capturing the variable *fan*@*last*, i.e., the previous value of *fan*. This is a solution to handle loops, which are either forbidden or troublesome in RP. Consequently, the system must know the initial value of *fan* using a dedicated annotation.

Finally, consider the GUI example. This includes dashed triangles, which denote continuous streams of data (often refer to as *behaviour* in functional RP, as opposed to *signal*). This means that updates to the mouse coordinates or to the time passing do not trigger a computation. Here *sel* can fire a boolean that will trigger data to flow from either *mouse* or *time* to *disp*. Furthermore,

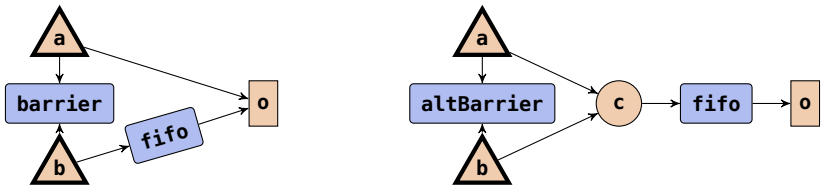

Figure 2: Example of typical Synchronous Coordinators: variations of an alternator.

the computation may not depend on all of its inputs, as opposed to the other operations seen so far. Hence, the composing operation depends, at each phase, on either *mouse* or *time*, and not on both.

Synchronous coordinators (SC) Synchronous coordinators provide a finer control over the scheduling restrictions of each of the stream updates, as illustrated in the two examples of Figure 2. These represent different coordinators that have two inputs, a and b, and alternate their values to an output stream o. In RP a similar behaviour could be captured by "$o = if(aLast)$ then b else a ; $aLast = \mathsf{not}(aLast@last)$". Using a synchronous coordinator, one can exploit synchrony and better control the communication protocol.

The coordinators on Figure 2 use the blocks `fifo`, `barrier`, and `altBarrier`, and may connect streams directly. Unlike in RP, these connections are *synchronous*, meaning that all streams involved in an operation must occur atomically. E.g., each stream can fire a single message only if the connected block or stream is ready to fire, which in turn can only happen if all their outputs are ready to fire. In the left coordinator, the top a can output a message only if it can send it to both o and the `barrier`. This `barrier` blocks a or b unless both a and b can fire atomically. The `fifo` can buffer at most one value, blocking incoming messages when full. The left coordinator receives each data message from both a and b, sending the message from a to c atomically and buffering the value from b; later the buffered message is sent to c, and only then streams a and b can fire again. The right coordinator uses a `altBarrier` that alternates between blocking a and blocking b, and it buffers the value temporarily to avoid o from having to synchronise with a or b.

Remarks In SC data streams can fire only once, and do not store this value unless it is explicit in the coordinator. In RP, when a stream fires a value, this value is stored for later reuse – either by the sender or by the computing tasks, depending on the implementation engine. Also, the notion of synchronisation, describing sets of operations that occur atomically, is not common in RP, since RP targets efficient implementations of tasks that run independently.

The term *reactive* has also been applied in the context of *reactive systems* and *functional reactive systems*. The former addresses systems that react to incoming stimuli and are responsive, resilient, elastic and message driven, described in the *Reactive Manifesto* [1]. The latter is a specific take on reactive program-

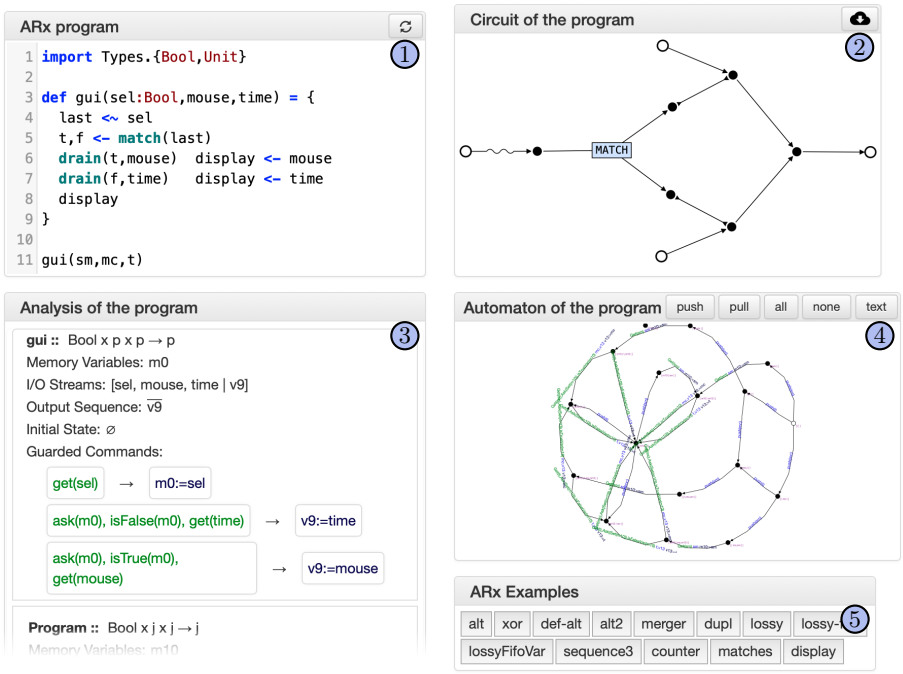

Figure 3: Screenshot of the widgets in the online tool for ARx programs.

ming based on functions over streams that distinguish (continuous) behaviour from (discrete) events [12]. Early work on synchronous languages for (real-time) reactive systems has been focused on safety-critical reactive control system, and includes synchronous programming and (synchronous) dataflow programming [5]. Similarly to synchronous coordination, synchronous languages such as Esterel [6] and StateCharts [13], assume that reactions are atomic and take no time, simplifying programs and allowing their representation as finite state machines which can be later translated into sequential programs.

3 ARx toolset

We implemented an open-source web-based prototype tool to edit, parse, and type check ARx programs, and to animate their semantics.[5] These tools are developed in Scala, which is compiled to JavaScript using ScalaJS.[6] This section starts by giving a quick overview on how to use the tools, using as running example a version of the GUI manager from Figure 1 in ARx. The toolset includes several widgets, depicted in Figure 3: ① the editor to specify the program, ② the

[5] http://arcatools.org/#arx
[6] https://www.scala-js.org

architectural view of the program, ③ the type and semantic analysis of the program, ④ a finite automaton capturing the reactive semantics of the program, and ⑤ a set of predefined examples.

Most of the syntax in ① in introduced in Section 4. Variables, such as `mouse` and `time`, denote streams of data; new lines are ignored; and the statements "`display←mouse display←time`" mean that the stream `display` merges the data from `mouse` and `display`. Extensions to the core syntax include (1) algebraic data types (Section 5), deconstructed with `match` (line 5), and (2) a reactive variable introduced by the arrow $\leftsquigarrow$ in line 4 (Section 6).

The semantics of an ARx program is given by a guarded command language, which we call stream builders (Section 4.2), following the ideas from Dokter and Arbab's stream constraints [10]. An instance of this intermediate language is illustrated in ③, and includes not only stream variables from ARx, but also memory variables (e.g., `m0`). These guarded commands include the guards (1) `get(mouse)` to denote a destructive read from the `mouse` stream, (2) `isFalse(m0)` as a predicate introduced in our first extension, and `ask(m0)` is a non-destructive read introduced in our second extension.

Stream builders have an operational semantics: they evolve by consuming input streams and memory variables, and by writing to output streams. Furthermore, the reactive extension in Section 6 adds an extra step to signal the interest in writing-to or reading-from a stream. This reactive semantics is animated in an automata view, depicted in ④. Note that this automata grows quickly, but it is usually unnecessary, as the stream builders act as a compact and symbolic representation of the automata.

4 Core ARx

4.1 ARx: Syntax

A program is a statement, according to the syntax in Figure 4. Expressions are either *terms* t, or names of so-called *stream builders* bn parameterised by a sequence of variables $\overline{x}$. In turn, terms can be *stream variables* x, *data values* d, or *function* names parameterised by variables $\overline{x}$.

So far we leave open the notions of *stream builders* and *functions*. Stream builders will be introduced in Section 4.2, and will give semantics to ARx programs. Functions are assumed to be deterministic and total with an interpretation $\mathcal{I}$ that maps closed terms to values; in Section 5 we will restrict to constructors of user-defined algebraic data types, as in our prototype implementation.

Regarding the remaining constructions, a *statement* is either an assignment a, a stream expression e, a builder definition d, or a parallel composition of statements s s. An *assignment* assigns a stream expression e to a non-empty sequence of stream variables $\overline{x}$. A builder definition **`def`** introduces a name bn associated to a new stream builder of a given block of statements s.

Statement	$s ::= a \mid e \mid d \mid s\ s$	Stream Expression	$e ::= t \mid bn(\overline{x})$
Assignment	$a ::= \overline{x} \leftarrow e$	Term	$t ::= x \mid d \mid fn(\overline{x})$
		Builder Definition	$d ::= \textbf{def}\ bn(\overline{x}) = \{s\}$

Figure 4: ARx' basic syntax, where bn ranges over names of stream builders, fn ranges over names of functions, and x over stream variables.

Examples The examples below assume the existence of stream builders `fifo` and `barrier`,[7] and the function `ifThenElse` with some interpretation. Consider the `alternator` definition, capturing the program from the left of Figure 2. This has two input streams as parameters: `a` and `b`, which must fire together because of the `barrier`. Their values are redirected to `c`: the one from `a` flows atomically, and the one from `b` is buffered until a follow-up step. The stream `c` is the only output of the definition block.

```
def alternator(a,b) = {
  barrier(a,b)  c←a
  c←fifo(b)     c
}
```

```
def gui(sel,mouse,time) = {
  display ←
    ifThenElse(sel,mouse,time)
}
```

Stream builders are typed, indicating the data types that populate the each input and output stream. Our implementation uses a type-inference engine that unifies matching streams and uses type variables. In our example from Figure 3, the inferred type of the `gui` builder is `Bool` $\times$ `p` $\times$ `p` $\rightarrow$ `p`, meaning that its first argument has type `Bool`, and all other ports must have the same type `p`. The type system also imposes all input stream variables inside **`def`** clauses to be parameters. We leave the type rules out of the scope of this paper, which focuses on the tools and on the semantics of ARx.

4.2 Stream builders: Syntax

Programs in ARx express functions from streams to streams, and describe the (non-strict) order between consuming and producing values. ARx's semantics is given by stream builders, defined below, which are closely inspired on Dokter and Arbab's stream constraints [10].

Definition 1 (Stream builder). *A* stream builder *sb follows the grammar:*

$$
\begin{array}{rcll}
sb & ::= & \overline{upd} \wedge [\overline{gc}] & \text{(stream builder)} \\
gc & ::= & \overline{guard} \rightarrow \overline{upd} & \text{(guarded command)} \\
guard & ::= & \mathsf{get}(v) \mid \mathsf{und}(v) \mid t & \text{(guard)} \\
upd & ::= & v := t & \text{(update)}
\end{array}
$$

where v is a variable name, t is a term, and $\overline{x}$ is a sequence of elements from x.

[7] `barrier` is known as `drain` in our tools.

Each variable v can represent an *input stream* or an *output stream* (or both), as in ARx, or an *internal memory*. Terms t are the same as in ARx, but with variables also over internal memory, and we write $\mathsf{fv}(t)$ to denote the set variables in t. Let s be a variable pointing to a stream and m a memory variable. Intuitively, a guard $\mathsf{get}(s)$ means that the head of s, noted $s(0)$, is ready to be read; a guard $\mathsf{get}(m)$ means that the memory variable m is defined; and a guard $\mathsf{und}(x)$ means that x has no value yet or is undefined.

A stream builder consists of an initial update that initialises memory variables and a set of guarded commands of the form $g \to u$ that specify a set of non-deterministic rules, such that the update u can be executed whenever guard g holds. When ***executing*** an update from a guarded command with a guard that succeeds, each $s := t$ sets $s(0)$ to the value obtained from t, each $m := t$ sets m to be the value from t; every stream s with $\mathsf{get}(s)$ in the guard is updated to s' (the head of s is consumed), and every memory m with $\mathsf{get}(m)$ in the guard—and not set in the update—becomes undefined. As a side-remark, these constructions get, und, and $v := t$ are analogue to the operations *get*, *nask*, and *tell*, respectively, over shared tuple-spaces in Linda-like languages, well studied in the literature in that context [8].

We further restrict which streams can be used in updates and guards based on whether they are input streams I—that can be read—or output streams O—that can be built. This is formalised by the notion of well-definedness below. Intuitively, in addition to the previous restrictions, the initial updates can be only over memory variables and use terms with no variables, i.e., memory variables can only be initialized with data values rather than streams; and for every guarded command, undefined variables cannot be "gotten", both guards and terms in updates can only use defined memory and input stream variables, and assignments in guarded commands cannot have cycles (which can occur since input and output stream variables may intersect). We use the following notation: given a stream builder sb, we write $sb.I$, $sb.O$, and $sb.M$ to denote its input streams, output streams, and memory variables; and we write $\mathsf{get}(\overline{v})$ (and analogously for und) to denote a set of occurrences of $\mathsf{get}(v)$, for every $v \in \overline{v}$.

Definition 2 (Well-defined stream builder). *Let sb be the stream builder:*

$$\overline{v_{init} := t_{init}} \wedge [\mathsf{get}(\overline{v_{get}}), \mathsf{und}(\overline{v_{und}}), \overline{t_{guard}} \to \overline{v_{out} := t_{out}}, \ldots].$$

We say sb is well-defined *if $\overline{v_{init}} \subseteq sb.M$ and $\mathsf{fv}(t_{init}) = \emptyset$, and if for every guarded command, the following conditions are met:*

$$\overline{v_{get}} \cap \overline{v_{und}} = \emptyset \qquad \overline{v_{get}} \cup \overline{v_{und}} \subseteq sb.I \cup sb.M \qquad \overline{v_{out}} \subseteq sb.O \cup sb.M$$
$$\mathsf{fv}(t_{guard}) \subseteq \overline{v_{get}} \qquad \mathsf{fv}(\overline{t_{out}}) \times \overline{v_{out}} \text{ is acyclic} \qquad \mathsf{fv}(t_{out}) \subseteq \overline{v_{get}}$$

Examples We omit the initial updates of a stream builder when empty, and write $\mathtt{builder}\langle v_1, v_2, \ldots\rangle = \overline{upd} \wedge \overline{gc}$ in the examples below to define a stream builder $\mathtt{builder}$ as $\overline{upd} \wedge \overline{gc}$ over variables $\{v_1, v_2, \ldots\}$, using the convention that i

denotes an input stream, o denotes an output stream, and m denotes a memory.

$$
\begin{array}{rcl}
sb_{add}\langle i_1, i_2, o\rangle & = & [\mathsf{get}(i_1), \mathsf{get}(i_2) \ \rightarrow \ o := i_1 + i_2] \\
sb_{xor}\langle i, o_1, o_2\rangle & = & [\mathsf{get}(i) \ \rightarrow \ o_1 := in \ \ , \ \ \mathsf{get}(i) \ \rightarrow \ o_2 := in] \\
sb_{fifo}\langle i, o, m\rangle & = & [\mathsf{get}(i), \mathsf{und}(m) \ \rightarrow \ m := i \ \ , \ \ \mathsf{get}(m) \ \rightarrow \ o := m] \\
sb_{fifoFull42}\langle i, o, m\rangle & = & m := 42 \wedge sb_{fifo}\langle i, o, m\rangle \\
sb_{barrier}\langle i_1, i_2\rangle & = & [\mathsf{get}(i_1), \mathsf{get}(i_2) \ \rightarrow \ \emptyset] \\
sb_{alternator}\langle i_1, i_2, o, m\rangle & = & \begin{bmatrix} \mathsf{get}(a), \mathsf{get}(b), \mathsf{und}(m) & \rightarrow c := a, m := b \\ \mathsf{get}(m) & \rightarrow c := m \end{bmatrix}
\end{array}
$$

Informally, the sb_{add} stream builder receives values from two streams and outputs its sum. At each round, it atomically collects a value from each input stream and produces a new output value. In sb_{xor} there are two non-deterministic options at each round: to send data to one output stream or to a second output stream. In sb_{fifo} the two options are disjoint: if m is undefined only the left rule can be triggered, and if m is defined only the right rule can be triggered, effectively buffering a value when m is undefined, and sending m when m is defined (becoming undefined again). The formal behaviour is described below. Later in the paper, we will present a composition operator for stream builders, allowing $sb_{alternator}$ to be built out simpler builders.

4.3 Stream builders: operational semantics

Consider a stream builder $sb = \overline{init} \wedge \overline{gc}$ with an interpretation $\mathcal{I}$ of closed terms as data values. The semantics of sb is given by the rewriting rule below of a mapping σ_m, from memory variables to their data values, with initial configuration $\langle \overline{init} \rangle$, using the following notation: $t[\sigma]$ captures the substitution on t of σ, $\mathsf{dom}(\sigma)$ is the domain of σ, $\sigma - X$ is the map σ excluding the keys in X, $\sigma_1 \cup \sigma_2$ is the map σ_1 extended with σ_2, updating existing entries, and $\mathsf{gets}(g)$ returns the variables within get constructs in the guard g. We use σ_i and σ_o to represents the current mapping from (the first element of) input and output stream variables to data values, respectively.

Definition 3 (Guard satisfaction). *Given a guard g and the current state of a system, given by σ_m and σ_i, the satisfaction of g, denoted $\sigma_m, \sigma_i \models g$, is defined as follows.*

$$
\begin{array}{ll}
\sigma_m, \sigma_i \models \mathsf{get}(\overline{v}) \ \textit{if} \ \ \overline{v} \subseteq \mathsf{dom}(\sigma_m) \cup \mathsf{dom}(\sigma_i) & \sigma_m, \sigma_i \models t \ \textit{if} \ \ \mathcal{I}(t[\sigma_m][\sigma_i]) = \textit{true} \\
\sigma_m, \sigma_i \models \mathsf{und}(\overline{v}) \ \textit{if} \ \ \overline{v} \cap \sigma_m = \emptyset & \sigma_m, \sigma_i \models \overline{g} \ \textit{if} \ \ \forall_{g_i \in \overline{g}} \cdot \sigma_m, \sigma_i \models g_i
\end{array}
$$

Definition 4 (Operational semantics). *The semantics of a stream builder $sb = \overline{init} \wedge \overline{gc}$ is given by the rule below, with an initial configuration $\langle \overline{init} \rangle$.*

$$
\frac{(g \rightarrow u) \in \overline{gc} \qquad \sigma_o = \left\{ v \mapsto d \ \mid (v := t) \in u, \ v \in sb.O, d = \mathcal{I}(t[\sigma_m][\sigma_i]) \right\} \qquad \sigma_m, \sigma_i \models g \qquad \sigma'_m = \left\{ v \mapsto d \ \mid (v := t) \in u, v \in sb.M, d = \mathcal{I}(t[\sigma_m][\sigma_i]) \right\}}{\langle \sigma_m \rangle \xrightarrow{\sigma_i, \sigma_o} \langle (\sigma_m - \mathsf{gets}(g)) \cup \sigma'_m \rangle}
$$

Intuitively, $\langle\sigma\rangle \xrightarrow{\sigma_i,\sigma_o} \langle\sigma'\rangle$ means that, for every variable i and data value d such that $\sigma(i) = d$, the system reads the values d as the next value in the stream i, and produces the next value of each stream $o \in \mathsf{dom}(\sigma_o)$ as being $\sigma_o(o)$. The internal state is captured by σ_m that stores values of memory variables in $sb.M$, which is updated based on σ_i. Furthermore, the system can only evolve by executing an *active* guarded command. Intuitively, a guarded command is active if the current state of memory (σ_m) and input stream (σ_i) variables satisfy the guard g, such that: each term guard has an interpretation that evaluates to *true*; all required input stream variables coincide with the set of defined input stream variables; all required memory variables are contained in the defined memory variables; and all required undefined memory variables are indeed undefined. For example, the following are valid runs of the stream builders in Section 4.2.

$$sb_{alternator}: \ \langle\emptyset\rangle \xrightarrow{\{in_1\mapsto 5, in_2\mapsto 8\},\{out_1\mapsto 5\}} \langle m \mapsto 8\rangle \xrightarrow{\emptyset,\{out\mapsto 8\}} \langle\emptyset\rangle$$

$$sb_{add}: \ \langle\emptyset\rangle \xrightarrow{\{in_1\mapsto 3, in_2\mapsto 2\},\{out\mapsto 3+2\}} \langle\emptyset\rangle \xrightarrow{\{in_1\mapsto 2, in_2\mapsto 7\},\{out\mapsto 2+7\}} \langle\emptyset\rangle$$

4.4 Composing Stream Builders

The composition of two stream builders yields a new stream builder that merges their initial update and their guarded commands, under some restrictions. I.e., the memory variables must be disjoint, some guarded commands can be included, and some pairs of guarded commands from each stream builder can be combined, formalised below in Definitions 5 and 6 after introducing some preliminary concepts. In the following we use the following auxiliary functions: $\mathit{out}(gc)$ returns the output streams assigned in the RHS of gc, $\mathit{in}(gc)$ returns the input streams inside *get* statements of gc, and $\mathit{vars}(gc)$ returns $\mathit{out}(gc) \cup \mathit{in}(gc)$.

The composition of stream builders follows the same principles as the composition of, e.g., constraint automata (CA) [4]. But unlike in CA and in most Reo semantics, it supports explicitly many-to-many compositions. I.e., a builder with an input stream i can be composed with another builder with the same input stream i, imposing the associated guarded commands to be together. Similarly, a builder with an output stream o can be combined with another one with the same output stream o, although only one builder can write to o at a time. A builder with an input stream x can be composed with another with an output stream with the same name x, making x both an input and an output in further compositions. The composition rules were carefully designed to keep the composition commutative and associative, which we do not prove in this paper.

We start by introducing the following auxiliary predicates used in the composition, using gc_i to range over guarded commands and I_1 to range over stream variables (meant to be from the same stream builder as gc_1).

$$\begin{aligned}
\mathit{matchedOuts}(I_1, gc_1, gc_2) &\equiv I_1 \cap \mathsf{out}(gc_2) \subseteq \mathit{in}(gc_1)\\
\mathit{matchedIns}(I_1, gc_1, gc_2) &\equiv I_1 \cap \mathit{in}(gc_2) \subseteq \mathit{vars}(gc_1)\\
\mathit{exclusiveOut}(gc_1, gc_2) &\equiv \mathit{out}(gc_1) \cap \mathit{out}(gc_2) = \emptyset\\
\mathit{noSync}(I_1, gc_2) &\equiv I_1 \cap \mathit{vars}(gc_2) = \emptyset
\end{aligned}$$

The predicate $\mathit{matchedOuts}(I_1, gc_1, gc_2)$ means that any input stream in I_1 that is an output of gc_2 must be read by gc_1, i.e., must be an input stream used by gc_1. Its dual $\mathit{matchedIns}(I_1, gc_1, gc_2)$ is not symmetric: it means that any input stream in I_1 that is an input gc_2 must either be written-to or read-by gc_1. This reflects the fact that input streams replicate data, and that input streams may also be output streams that may also be used to synchronise. The predicate $\mathit{exclusiveOut}(gc_1, gc_2)$ states that gc_1 and gc_2 do not share output streams, reflecting the fact that only one rule can write to a stream at a time. The last predicate $\mathit{noSync}(I_1, gc_2)$ states that gc_2 will not affect any of the input streams in I_1. Intuitively this means that gc_2 may read-from or write-to streams from another builder sb_1 if they can also be written by sb_1, but not if they are read by sb_1.

The composition of guarded commands and of stream builders is defined below, based on these predicates.

Definition 5 (Composition of guarded commands ($gc_1 \bowtie gc_2$)). *For $i \in \{1,2\}$, let $gc_i = \mathsf{get}(\overline{v_{gi}}), \mathsf{und}(\overline{v_{ui}}), \overline{t_{gi}} \to \overline{v_{oi} := t_{oi}}$ be two guarded commands. Their composition yields $gc_1 \bowtie gc_2$ defined below.*

$$\mathsf{get}((\overline{v_{g1}} \cup \overline{v_{g2}}) - (\overline{v_{o1}} \cup \overline{v_{o2}}))\,,\ \mathsf{und}(\overline{v_{u1}} \cup \overline{v_{u2}})\,,\ \overline{t_{g1}} \cup \overline{t_{g2}} \ \to\ \overline{v_{o1} := t_{o1}} \cup \overline{v_{o2} := t_{o2}}$$

Definition 6 (Composition of stream builders ($sb_1 \bowtie sb_2$)). *For $i \in \{1,2\}$, let $sb_i = \overline{init_i} \wedge [\overline{gc_i}]$ be two stream builders. Their composition yields $sb = sb_1 \bowtie sb_2 = (\overline{init_1} \cup \overline{init_2}) \wedge [gcs]$, where $sb.O = sb_1.O \cup sb_2.O$, $sb.I = sb_1.I \cup sb_2.I$, $sb.M = sb_1.M \cup sb_2.M$, and gcs is given by the smallest set of guarded commands that obeys the rules below, which are not exclusive.*

$$(\textsc{Com1})\ \frac{gc \in \overline{gc_1} \quad \mathit{noSync}(sb_2.I, gc)}{gc}$$

$$(\textsc{Com2})\ \frac{gc \in \overline{gc_2} \quad \mathit{noSync}(sb_1.I, gc)}{gc}$$

$$(\textsc{Com3})\ \frac{\forall i, j \in \{1,2\}, i \neq j :\quad gc_i \in \overline{gc_i} \quad \mathit{matchedOuts}(sb_i.I, gc_i, gc_j) \quad \mathit{matchedIns}(sb_i.I, gc_i, gc_j) \quad \mathit{exclusiveOut}(gc_i, gc_j)}{gc_1 \bowtie gc_2}$$

Intuitively, any guarded command can go alone, unless it must synchronise on shared streams. Any two guarded commands can go together if their synchronization is well-defined and do not perform behaviour that must be an exclusive choice. Observe that the composition of two well-defined stream builders (c.f. Definition 2) may not produce a well-defined stream builder (e.g. cyclic assignments), in which case we say that the stream builders are incompatible and cannot be composed.

Example The sequential composition of two `fifo` builders is presented below, annotated with the rule name that produced it. The ~~get(b)~~ guard was dropped during composition (Definition 5), but included here to help understanding. The last two guarded commands, in gray, denote scenarios where the middle stream

$$
\begin{array}{rll}
[\![bn(\overline{x})]\!]_\Gamma = \left(sb\left[\overline{x}/\overline{x_I}, \overline{y}/\overline{x_O}, \overline{z}/\overline{rest}\right], \overline{y}\right) & \begin{cases} \Gamma(bn) = (sb, \overline{x_I}, \overline{x_O}) \\ \overline{rest} = \mathsf{fv}(sb) - \overline{x_I} - \overline{x_O} \\ \text{for some fresh } \overline{y}, \overline{z} \end{cases} \\
[\![t]\!]_\Gamma = \left(\left[\overline{\mathsf{get}(\mathsf{fv}(t))} \to o := t\right], o\right) & \begin{cases} \text{for some fresh } o \\ I = \mathsf{fv}(t), O = \{o\}, M = \emptyset \end{cases} \\
[\![x]\!]_\Gamma = ([\mathsf{get}(x) \to o := x], o) & \begin{cases} \text{for some fresh } o \\ I = \{x\}, O = \{o\}, M = \emptyset \end{cases} \\
[\![\overline{x} \leftarrow e]\!]_\Gamma = (sb[\overline{x}/\overline{x_O}], \emptyset) & \{\ [\![e]\!]_\Gamma = (sb, \overline{x_O}) \\
[\![c_1\ c_2]\!]_\Gamma = (sb_1 \bowtie sb_2, \overline{x_O} \cdot \overline{y_O}) & \begin{cases} [\![c_1]\!]_\Gamma = (sb_1, \overline{x_O}) \\ [\![c_2]\!]_\Gamma = (sb_2, \overline{y_O}) \end{cases} \\
\left[\!\!\left[\begin{array}{c} \mathsf{def}\ bn(\overline{x}) = \{c_1\} \\ c_2 \end{array}\right]\!\!\right]_\Gamma = [\![c_2]\!]_{\Gamma[bn \mapsto (sb, \overline{x}, \overline{x_O})]} & \{\ [\![c_1]\!]_\Gamma = (sb, \overline{x_O})
\end{array}
$$

Figure 5: Semantics: encoding of builder compositions as a stream builder.

b remains open for synchronization. These are needed to make the composition operator associative, but can be discarded when hiding the internal streams like b. This is not explained here, but is implemented in our prototype tool. Following a similar reasoning, the stream builder $sb_{alternator}$ can be produced by composing the stream builders $sb_{barrier}\langle a, b\rangle$, $sb_{fifo}\langle b, c, m\rangle$, and $sb_{sync}\langle a, c\rangle$, which has no internal streams.

$$
sb_{fifo}\langle a, b, m_1\rangle \ \bowtie sb_{fifo}\langle b, c, m_2\rangle =
$$

$$
\left[\begin{array}{rll}
\mathsf{get}(a), \mathsf{und}(m_1) \to m_1 := a & & (\textsc{Com}1) \\
\sout{\mathsf{get}(b)}, \mathsf{get}(m_1), \mathsf{und}(m_2) \to b := m_1\ ,\ m_2 := b & & (\textsc{Com}3) \\
\mathsf{get}(a), \mathsf{und}(m_1), \mathsf{get}(m_2) \to m_1 := a\ ,\ c := m_2 & & (\textsc{Com}3) \\
\mathsf{get}(m_2) \to c := m_2 & & (\textsc{Com}2) \\
\mathsf{get}(a), \mathsf{und}(m_1), \mathsf{get}(b), \mathsf{und}(m_2) \to m_1 := a, m_2 := b & & (\textsc{Com}3) \\
\mathsf{get}(b), \mathsf{und}(m_2) \to m_2 := b & & (\textsc{Com}2)
\end{array}\right]
$$

4.5 ARx's Semantics: Encoding into Stream Builders

A builder composition can be encoded as a single stream builder under a context Γ of existing stream builders. More precisely, Γ maps names of stream builders bn to triples $(sb, \overline{x_I}, \overline{x_O})$ of a stream builder sb, a sequence of variables for input streams $\overline{x_I}$ of sb, and a sequence of variables for output streams $\overline{x_O}$. Given a context Γ, the encoding of a composition c, written $[\![c]\!]_\Gamma$, is defined in Figure 5. This encoding produces a pair with a stream builder and a sequence of output stream variables $\overline{x_O}$, built based on the stream builders from Γ, on the composition $\bowtie$, and on the substitution of variables $[x/y]$ (x substitutes y). Our implementation further applies a simplification of guarded commands in the **def** clause, by hiding output streams not in $\overline{x_O}$ and guarded commands that consume streams that are both input and output; but we do omit this process in this paper.

Program	$P ::= \overline{\mathcal{D}}\ \ s$	New Data Type	$\mathcal{D} ::= \textbf{data}\ D\langle\overline{\alpha}\rangle = Q(\overline{\mathcal{T}})\ \overline{\mid Q(\overline{\mathcal{T}})}$
Data Type	$\mathcal{T} ::= \alpha \mid D\langle\overline{\mathcal{T}}\rangle$	Stream Expression	$e ::= \cdots \mid \textbf{build}\langle\mathcal{T}\rangle(\overline{x}) \mid \textbf{match}\langle\mathcal{T}\rangle(\overline{x})$
Data Term	$t ::= Q(\overline{x})$	Builder Definition	$d ::= \textbf{def}\ bn(\overline{x : \mathcal{T}}) : \overline{\mathcal{T}} = \{s\}$

Figure 6: Syntax: extending the syntax from Figure 4 with algebraic data types, where α ranges over type variables, D over type names, and Q over data constructors.

The composition exemplified in Section 4.4, regarding the alternator, is used when calculating the encoding of "`barrier(a,b) c←fifo(b) c←a`" below, where $\Gamma = \{\texttt{fifo} \mapsto (sb_{fifo}\langle i, o\rangle, i, o), \texttt{barrier} \mapsto (sb_{barrier\langle i_1, i_2\rangle}, i_1 \cdot i_2, \emptyset)\}$.

$$[\![\texttt{barrier(a,b)}]\!]_\Gamma = ([\mathsf{get}(a), \mathsf{get}(b) \to \emptyset], \emptyset) \qquad (sb_1)$$

$$[\![\texttt{fifo(b)}]\!]_\Gamma = ([\mathsf{get}(b), \mathsf{und}(y_1) \to y_1 := b, \mathsf{get}(y_1) \to y_2 := y_1], y_2)$$

$$[\![\texttt{c←fifo(b)}]\!]_\Gamma = ([\mathsf{get}(b), \mathsf{und}(y_1) \to y_1 := b, \mathsf{get}(y_1) \to c := y_1], \emptyset) \qquad (sb_2)$$

$$[\![\texttt{c←a}]\!]_\Gamma = ([\mathsf{get}(a) \to c := a], c) \qquad (sb_3)$$

$$\left[\!\!\left[\begin{array}{l}\texttt{barrier(a,b)}\\ \texttt{c←fifo(b)}\\ \texttt{c←a}\end{array}\right]\!\!\right]_\Gamma = (sb_1 \bowtie sb_2 \bowtie sb_3, \emptyset) \ = \ (sb_{alternator}\langle a, b, c, y_1\rangle, \emptyset)$$

5 Extension I: Algebraic Data Types

This section extends our language of stream builders with constructs for algebraic data types, and allowing types to influence the semantics. Its grammar, presented in Figure 6, extends the grammar from Figure 4 with declarations of ADTs, with **build** and **match** primitive stream builders, and with type annotations for builder definitions. For simplicity, we also use the following notations: we omit $\overline{X}$, $\langle\overline{X}\rangle$, and $(\overline{X})$ when $\overline{X}$ is empty; we write **build**, **match**, and $bn(x)$ instead of **build**$\langle\alpha\rangle$, **match**$\langle\alpha\rangle$, and $bn(x : \alpha)$, respectively, when α is a type variable not used anywhere else; and we omit the output type $\overline{\mathcal{T}}$ in builder definitions to denote a sequence of fresh type variables, whose dimension is determined during type-checking (when unifying types).

A program starts by a list of definitions of algebraic data types, such as the ones below, which we will now assume to be part of a prelude library.

$$\textbf{data}\ \mathsf{Unit} = \mathsf{U} \qquad \textbf{data}\ \mathsf{Nat} = \mathsf{Zero} \mid \mathsf{Succ}(\mathsf{Nat})$$
$$\textbf{data}\ \mathsf{Bool} = \mathsf{True} \mid \mathsf{False} \qquad \textbf{data}\ \mathsf{List}\langle\alpha\rangle = \mathsf{Nil} \mid \mathsf{Cons}(\alpha, \mathsf{List}\langle\alpha\rangle)$$

These algebraic data types (ADTs) are interpreted as the smallest fix-point of the underlying functor, i.e., they describe finite terms using the constructors for data types. All constructors Q have at least one argument; when absent, it denotes an argument with a unit value $Q(\mathsf{Unit})$ (in declarations) and $Q(\mathsf{U})$

$$[\![\mathsf{match}\langle D\langle\overline{\mathcal{T}}\rangle\rangle]\!] = \left(\begin{bmatrix}\mathsf{get}(in), isQ_1(in) \;\rightarrow\; out_{1,1} := getQ_{1,1}(in), \ldots, out_{1,k_1} := getQ_{1,k_1}(in)\\ \cdots\\ \mathsf{get}(in), isQ_n(in) \;\rightarrow\; out_{n,1} := getQ_{n,1}(in), \ldots, out_{n,k_n} := getQ_{n,k_n}(in)\end{bmatrix}, \; out_{1,1}\cdots out_{1,k_1}\cdots out_{n,1}\cdots out_{n,k_n}\right)$$

$$[\![\mathsf{build}\langle D\langle\overline{\mathcal{T}}\rangle\rangle]\!] = \left(\begin{bmatrix}\mathsf{get}(in_{1,1}), \ldots, \mathsf{get}(in_{1,k_1}) \;\rightarrow\; out := Q_1(in_{1,1}, \ldots, in_{1,k_1})\\ \cdots\\ \mathsf{get}(in_{n,1}), \ldots, \mathsf{get}(in_{n,k_n}) \;\rightarrow\; out := Q_n(in_{n,1}, \ldots, in_{n,k_n})\end{bmatrix}, out\right)$$

Figure 7: Semantics of match and build, considering that D is defined as **data** $D\langle\overline{\mathcal{T}}\rangle = Q_1(g_{1,1}, \ldots, g_{1,k_1}) \mid \ldots \mid Q_n(g_{n,1}, \ldots, g_{n,k_n})$.

(in terms). Each definition of an ADT **data** $D\langle\overline{\mathcal{T}}\rangle = Q_1(\overline{g}) \mid \ldots \mid Q_n(\overline{g})$, e.g., **data** $\mathsf{List}\langle\alpha\rangle = \mathsf{Nil} \mid \mathsf{Cons}(\alpha, \mathsf{List}\langle\alpha\rangle)$, introduces:
- *Term constructors* Q_i to build new terms, e.g. Nil and Cons(True, Nil);
- *Term inspectors* $isQ_i(x)$ that check if x was built with Q_i, e.g. isNil and isCons return True only if their argument has shape Nil or Cons, respectively;
- *Term projections* $getQ_{i,j}$ that given a term built with Q_i return the j-th argument, e.g. $get\mathsf{Cons}_2(\mathsf{Cons}(\mathsf{True}, \mathsf{Cons}(\mathsf{False},\mathsf{Nil}) = \mathsf{Cons}(\mathsf{False}, \mathsf{Nil}))$;

Given these new constructs the new semantic encodings is presented in Figure 7. For example, $[\![\mathsf{match}\langle\mathsf{List}\langle\alpha\rangle\rangle]\!]$ yields the builder below, and $[\![\mathsf{match}\langle\alpha\rangle]\!]$ is undefined unless α is refined to an ADT.

$$\begin{bmatrix}\mathsf{get}(in), is\mathsf{Nil}(in) \;\rightarrow\; out_{1,1} := get\mathsf{Nil}_1(in);\\ \mathsf{get}(in), is\mathsf{Cons}(in) \;\rightarrow\; out_{2,1} := get\mathsf{Cons}_1(in), out_{2,2} := get\mathsf{Cons}_2(in)\end{bmatrix}$$

6 Extension II: Reactive Semantics

In reactive languages, produced data is typically kept in memory, possibly triggering consumers when it is initially produced. In this section we provide a finer control about who can trigger the computation, and a notion of memory that is read without being consumed, to accommodate both perspectives.

In the semantics of stream builders we add a notion of *active* variables, whereas a guarded command can only be selected if one of its variables is active, and adapt the operational semantics accordingly. We also introduce a new element to the guards: $\mathsf{ask}(v)$, that represents a non-destructive read.

Syntax: asking for data The extension for our language consists of updating the assignment grammar:

$$\text{Assignment} \qquad a ::= \overline{x} \leftarrow e \mid \overline{x} \rightsquigarrow e$$

whose squiggly arrow is interpreted as a creator of reactive variable: the values from e are buffered before being used by $\overline{x}$, and this values can be read (non-destructively) when needed using the new guard ask. This is formalised below.

$$
\begin{aligned}
[\![\overline{x} \rightsquigarrow e]\!]_\Gamma &= [\![(\overline{y} \leftarrow e)\ \ (\overline{x \rightsquigarrow y})]\!]_\Gamma \qquad \text{for some fresh } \overline{y} \\
[\![x \rightsquigarrow y]\!]_\Gamma &= \left(\left[\mathsf{ask}(m) \to x := m\ ,\ \mathsf{get}(y) \to m := y\right]\ ,\ \emptyset\right)
\end{aligned}
$$

Observe that "$\mathsf{get}(m) \to x := m, m := m$" is very similar to "$\mathsf{ask}(m) \to x := m$". The former consumes the variable m and defines it with its old value, and the latter reads m without consuming it. This subtle difference has an impact in our updated semantics, defined below, by marking assigned variables as *"active"*. In the first case m becomes active, allowing guarded commands that use m to be fired in a follow up step. In the second case m will become inactive, and guarded commands using m with no other active variables will not be allowed to fire.

Semantics: active/passive variables The reactive semantics for a stream builder $sb = \overline{init} \wedge \overline{gc}$ is given by the rules below. The state is extended with two sets of so-called *active* input and output variables, with initial state $\langle \overline{init}, \emptyset, \emptyset \rangle$. A system can evolve in two ways: (1) by evolving the program as before, consuming and producing data over variables, or (2) by an update to the context that becomes ready to write to (push) or read from (pull) a stream. Below we write "out(u)" to return the assigned variables in u (c.f. Section 4.4), "in(g)" to return the variables of g within get and ask constructs, and $\langle \sigma \rangle \xrightarrow{x}_{g,u} \langle \sigma' \rangle$ to denote the step from state σ to σ' by x when selecting the guarded command $g \to u$.

$$
\frac{\begin{array}{l}(g \to u) \in \overline{gc} \quad \langle \sigma_m \rangle \xrightarrow{\sigma_i, \sigma_o}_{g,u} \langle \sigma'_m \rangle \\ \left(\mathsf{in}(g) \cap A_i \neq \emptyset\right) \vee \left(\mathsf{out}(u) \cap A_o \neq \emptyset\right) \\ A'_i = (A_i - \mathsf{in}(g)) \cup (\mathsf{out}(u) \cap sb.M) \\ A'_o = A_o - \mathsf{out}(u)\end{array}}{\langle \sigma_m, A_i, A_o \rangle \xrightarrow{\sigma_i, \sigma_o} \langle \sigma'_m, A'_i, A'_o \rangle}
$$

$$
\frac{\langle \sigma_m \rangle \nrightarrow \langle \sigma'_m \rangle \quad x \in sb.I}{\langle \sigma_m, A_i, A_o \rangle \xrightarrow{\mathsf{push}(x)} \langle \sigma'_m, A_i \cup \{x\}, A_o \rangle}
$$

$$
\frac{\langle \sigma_m \rangle \nrightarrow \langle \sigma'_m \rangle \quad x \in sb.O}{\langle \sigma_m, A_i, A_o \rangle \xrightarrow{\mathsf{pull}(x)} \langle \sigma'_m, A_i, A_o \cup \{x\} \rangle}
$$

The previous semantic rules must be accommodated to take the ask constructor into account. This is done by redefining the *guard satisfaction* definition in Section 4.3 to incorporate a new rule, presented below, and *vars* in Section 4.4 to include also the ask variables.

$$
\sigma_m, \sigma_i \models \mathsf{ask}(\overline{v}) \qquad\qquad \overline{v} \subseteq \mathsf{dom}(\sigma_m)
$$

Example: ADT and reactivity We illustrate the encoding and semantics of reactive stream builders using the GUI manager example (Figure 1 and Figure 3). The equality below depicts the adapted system following the ARx syntax (left) and its semantics (right).

$$
\left[\!\!\left[\begin{array}{l}\texttt{last} \rightsquigarrow \texttt{sel} \\ \texttt{t,f} \leftarrow \texttt{match(last)} \\ \texttt{barrier(t,mouse)} \\ \texttt{barrier(f,time)} \\ \texttt{display} \leftarrow \texttt{mouse} \\ \texttt{display} \leftarrow \texttt{time} \\ \texttt{display}\end{array}\right]\!\!\right] = \left[\begin{array}{lll}\mathsf{get}(\texttt{sel}) & \to & m := \texttt{sel} \\ \mathsf{get}(\texttt{mouse}), & & \texttt{last} := m, \\ \cancel{\mathsf{get}(\texttt{last}), \mathsf{get}(\texttt{t}),} & \to & \texttt{t} := getTrue_1(\texttt{last}), \\ \mathsf{ask}(m), isTrue(m) & & \texttt{display} := \texttt{mouse} \\ \mathsf{get}(\texttt{time}), & & \texttt{last} := m, \\ \cancel{\mathsf{get}(\texttt{last}), \mathsf{get}(\texttt{f}),} & \to & \texttt{f} := getFalse_1(\texttt{last}), \\ \mathsf{ask}(m), isFalse(m) & & \texttt{display} := \texttt{time}\end{array}\right]
$$

This encoding also returns the sequence of output streams, which in this case is `display`. The stream builder is further simplified by our toolset by removing intermediate stream variables `last`, `t`, and `f` from the updates, as depicted in the screenshot in Figure 3.

The following transitions are valid runs of this program.

$$
\begin{aligned}
\langle \emptyset, \emptyset, \emptyset \rangle &\xrightarrow{\text{pull}(\texttt{display})} \langle \emptyset, \emptyset, \{\texttt{display}\} \rangle \\
&\xrightarrow{\text{push}(\texttt{sel})} \langle \emptyset, \{\texttt{sel}\}, \{\texttt{display}\} \rangle \\
&\xrightarrow{\texttt{sel} \mapsto \text{True}} \langle \{m \mapsto \text{True}\}, \{m\}, \{\texttt{display}\} \rangle \\
&\xrightarrow{\texttt{mouse} \mapsto (2,3); \texttt{display} \mapsto (2,3)} \langle \{m \mapsto \text{True}\}, \emptyset, \emptyset \rangle \\
&\xrightarrow{\text{pull}(\texttt{display})} \langle \{m \mapsto \text{True}\}, \emptyset, \{\texttt{display}\} \rangle \\
&\xrightarrow{\texttt{mouse} \mapsto (5,8); \texttt{display} \mapsto (5,8)} \langle \{m \mapsto \text{True}\}, \emptyset, \emptyset \rangle
\end{aligned}
$$

7 Conclusions

We proposed *ARx*, a lightweight programming language to specify component-based architecture for reactive systems, blending principles from reactive programming and synchronous coordination languages. *ARx* supports algebraic data types and is equipped with a type checking engine (not introduced here) to check if streams are well-composed based on the streamed data.

Programs are encoded into *stream builders*, which provide a formal and compositional semantics to build programs out of simpler ones. A stream builder specifies the initial state of a program and a set of guarded commands which describe the steps (commands) that the program can perform provided some conditions (guards)—over the internal state and the inputs received from the environment—are satisfied.

We built an online tool to specify, type check, and analyse the semantics of ARx programs, and visualize both the architectural view of the program and its operational reactive semantics.

Future work plans include the *verification* of properties, the addition of new *semantic extensions*, and the development of *code generators*. These properties could be specified using hierarchical dynamic logic and verified with model checkers such as mCRL2, following [17], or could address the possibility of infinite loops caused by priorities of push and pulls from the environment. The semantic extensions could target, e.g., notions of variability, probability, time, and quality of service.

Acknowledgments This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDB/04234/2020); by the Norte Portugal Regional Operational Programme (NORTE 2020) under the Portugal 2020 Partnership Agreement, through the European Regional Development Fund (ERDF) and also by national funds through the FCT, within project NORTE-01-0145-FEDER-028550 (REASSURE); and by the

Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by national funds through the FCT, within projects POCI-01-0145-FEDER-029946 (DaVinci) and POCI-01-0145-FEDER-029119 (PReFECT).

References

1. The reactive manifesto v2.0. https://www.reactivemanifesto.org, 2014.
2. Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
3. Christel Baier. Probabilistic models for Reo connector circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, 2005.
4. Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
5. Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013.
6. Gérard Berry. The foundations of Esterel. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. The MIT Press, 2000.
7. Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Full abstraction for signal flow graphs. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, POPL '15, pages 515–526, New York, NY, USA, 2015. ACM.
8. Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of coordination via shared dataspaces. *Sci. Comput. Program.*, 46(1-2):71–98, 2003.
9. Régis Cridlig and Eric Goubault. Semantics and analysis of linda-based languages. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *WSA*, volume 724 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 1993.
10. Kasper Dokter and Farhad Arbab. Rule-based form for stream constraints. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, pages 142–161, Cham, 2018. Springer International Publishing.
11. Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed rescala: an update algorithm for distributed reactive programming. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 361–376. ACM, 2014.
12. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.
13. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
14. Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. *Arrows, Robots, and Functional Reactive Programming*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
15. Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. page 18, 2010.

16. Sun Meng and Farhad Arbab. On resource-sensitive timed component connectors. In *FMOODS*, pages 301–316, 2007.
17. José Proença and Alexandre Madeira. Taming hierarchical connectors. In *Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, 2019*, Lecture Notes in Computer Science (to appear), 2019.
18. José Proença and Dave Clarke. Interactive interaction constraints. In Rocco De Nicola and Christine Julien, editors, *COORDINATION*, volume 7890 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2013.
19. Yoshitaka Sakurai and Takuo Watanabe. Towards a statically scheduled parallel execution of an FRP language for embedded systems. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2019, pages 11–20, New York, NY, USA, 2019. Association for Computing Machinery.