Mimosa: A Language for Asynchronous Implementation of Embedded Systems Software

Nikolaus Huber[1(✉)], Susanne Graf[1,2], Philipp Rümmer[1,3], and Wang Yi[1]

[1] Uppsala University, Uppsala, Sweden
nikolaus.huber@it.uu.se
[2] Université Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France
[3] University of Regensburg, Regensburg, Germany

Abstract. This paper introduces the Mimosa language, a programming language for the design and implementation of asynchronous reactive systems, describing them as a collection of time-triggered processes which communicate through FIFO buffers. Syntactically, Mimosa builds upon the Lustre data-flow language, augmenting it with a new semantics to allow for the expression of side-effectful computations, and extending it with an asynchronous coordination layer which orchestrates the communication between processes. A formal semantics is given to both the process and coordination layer through a textual and graphical rewriting calculus, respectively, and a prototype interpreter for simulation is provided.

Keywords: Data-flow · Kahn process networks · MIMOS · Embedded Systems · Cyber-physical systems · Coordination language · Formal semantics

1 Introduction

The synchronous paradigm is an established method for designing, implementing, and verifying the software layer of embedded systems. Languages like Lustre [15] (and its commercial implementation SCADE [8]), Signal [2], and Esterel [3] have been successfully used for the implementation of various safety-critical embedded control applications. However, on complex execution platforms, such as multi- and many-core processors or distributed architectures, the stringent timing constraints required by the execution models of such languages becomes increasingly difficult to uphold. Problems further arise when trying to update synchronous systems, as newly added software components might put additional constraints on the global system tick.

The MIMOS computational model [31] offers a new way of expressing asynchronous software designs for embedded systems. It describes a program as a

Published by springer Nature Switzerland AG 2025
C. Di Giusto and A. Ravara (Eds.): COORDINATION 2025, LNCS 15731, pp. 90–113, 2025.
https://doi.org/10.1007/978-3-031-95589-1_5

network of periodically triggered processes, which communicate through FIFO buffers. The MIMOS project [32] is an ongoing effort to develop a set of tools to allow designing, implementing, and verifying embedded software on top of the MIMOS model. The current prototype simulator consists of a graphical editor, which models the communication between processes, the computation (function) of each process can be expressed either in Java or C++.

In this paper, we equip MIMOS with its own user-facing programming language, the MIMOS Application (Mimosa) language. Mimosa consists of a small kernel, which covers the implementation of the (input/output) function of each process, as well as the coordination layer. Our main contribution is the definition of the formal semantics of both layers, utilizing different formal frameworks for each. This shall serve as the foundation for future work, including the development of verification and analysis tools, or even formally verified simulators and compilers.

The paper is structured in the following way. Section 2 gives a short overview of the MIMOS computational model, and defines the subset which is currently covered by Mimosa. Section 3 gives an introduction to Mimosa by showcasing various examples, before defining the abstract syntax and formal semantics of the language in Sect. 4. We present a simulator for Mimosa in Sect. 5. Finally, we report on related work in Sect. 6 and conclude with a list of ongoing and future efforts in Sect. 7.

2 The MIMOS Model

The MIMOS model [31] builds upon the well-known framework of Kahn process networks (KPNs) [19]. In a KPN, different software components communicate exclusively through unbounded FIFO buffers, where reading from a buffer is blocking, i.e., if a component tries to read from an empty buffer it is suspended until data is available. Kahn showed, that the output of such a system (i.e., the history of values appearing in each buffer) is deterministic, independently of the scheduling order of the individual components.

In real-time systems, a bounded delay between input and output must be guaranteed, which due to the independence of any scheduling order is difficult to prove for a given KPN. MIMOS addresses this issue by assigning to each component a release pattern (i.e., an infinite series of increasing time-tags), which marks the points in time at which each respective component will try to execute. In case a component does not have all its required input at a release time point, it remains idle until its next release.

Through this assignment of release patterns, different extensions to KPNs are possible, without affecting (timed) determinism. One such extension presented in [31] is registers, which instead of buffering allows expressing a latest-value semantics.

Another possible extension is optional inputs [32]. In a KPN, a process trying to read from a buffer must always wait until data is available. With timed release patterns, it is possible to define an input as optional, where the value is subject to availability in the buffer.

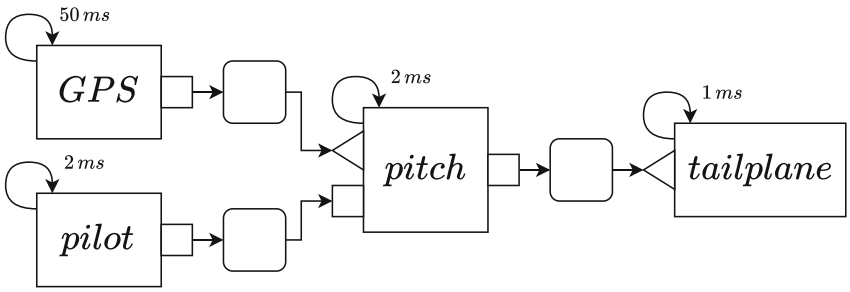


Fig. 1. Airplane pitch control system, adapted from [16].

Figure 1 shows a simplified example of a MIMOS network for the pitch control of an airplane. It consists of 4 nodes, one handling the input from the GPS subsystem, one polling the stick-control of the pilot, the actual pitch control algorithm, and the driver for the tailplane. As the GPS system is slow, it runs with a lower frequency (i.e., larger period) than the pitch control algorithm. The tailplane driver runs fastest, as it must react to minute changes in the mechanical parts of the aircraft. By utilizing optional inputs at the pitch control node for the GPS signal, as well as at the tailplane driver for the control signal, each node can run at its required frequency without constraining the rest of the system.

In the original MIMOS paper [31], a fix-point semantics has been presented, however, it neither refers to a particular programming language for the implementation of the processes, nor does it provide a textual syntax for the definition of the coordination layer. Our contribution in this paper is the formal definition of Mimosa, which equips MIMOS with a prototype programming language kernel.

As the main focus of this work is on the formal semantics of the language, we have put certain restrictions on the particular MIMOS model that we cover. For simplicity, registers are currently not modelled by our semantics, we discuss in Sect. 7 on how they can be added. We also restrict the release patterns of the processes to periodic releases, with an implicit deadline at the end of each period (i.e., if a component with period p reads its required inputs at time t, the outputs are written at $t + p$).

3 The Design of Mimosa

The design of the Mimosa language has been guided by the structure of the MIMOS computational model. Therefore, there is a clear distinction between computation and coordination, which results in different layers in the language.

The choice of syntax for the expression of computation in Mimosa is ultimately arbitrary, and future versions of the language may offer the possibility to express computation in any number of programming languages (an approach similar to Lingua Franca [21]). For the aim of this paper, which is on formal semantics, Mimosa derives its syntax from the declarative data-flow language Lustre [15], from which it retains the equation-based definition of computation.

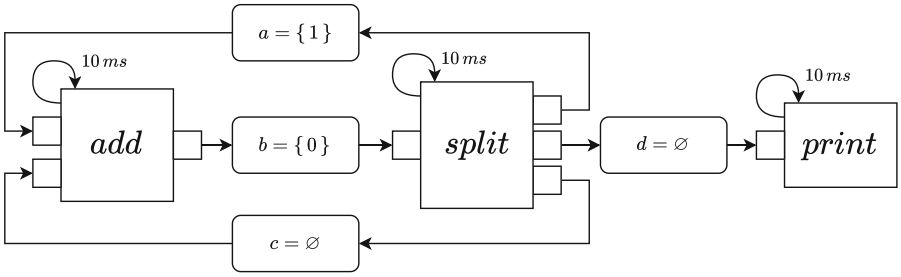


Fig. 2. Fibonacci example (graphical).

```
step print_int (_ : int) --> ()
step add (x, y) --> z { z = x + y }
step split inp --> (o1, o2, o3) { o1, o2, o3 = inp, inp, inp }

channel a : int = { 1 }
channel b : int = { 0 }
channel c : int
channel d : int

node add implements add (a, c) --> (b) every 10ms
node split implements split (b) --> (a, d, c) every 10ms
node print implements print_int (d) --> () every 10ms
```

Listing 1. Fibonacci example (Mimosa)

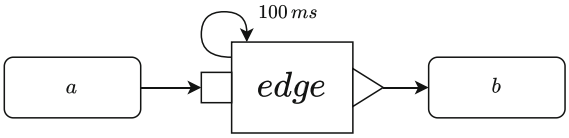


Fig. 3. Edge detector (graphical).

```
step edge_detect (in : bool) --> (out : bool?)
{
    pre_in = in -> pre in;
    out = if !pre_in && in then (Some true)
          else if pre_in && !in then (Some false)
          else None;
}

channel a : bool
channel b : bool

node edge implements edge_detect (a) --> (b?) every 100ms
```

Listing 2. Edge detector (Mimosa)

Lustre also provides the primitives to define state, which we use to express memory. This allows for a more natural definition, instead of having to declare memory cells explicitly (e.g., through state variables).

It is important to note, that even if the syntax is shared with Lustre, the evaluation of expressions in Mimosa is quite different, as will be further explained below.

In Lustre, any variable or expression denotes a (conceptually infinite) stream of values, so that a variable x actually represents a sequence $(x_0, x_1, \ldots)$. A Lustre program transfers a set of input streams into a set of output streams. Streams are defined through equations, each of the form $x = e$, where the expression e is formed from constants (representing infinite sequences of the same value), variables (i.e., references to other streams), and stream operators (like the usual unary and binary arithmetic and logic functions extended to operate point-wise on sequences) applied to argument streams.

In addition, Lustre defines a set of *sequence operators* (**pre**, **fby**, and $\rightarrow$). If $x = (x_0, x_1, \ldots)$, then **pre** x refers to the value of x from the previous cycle, i.e., **pre** $x = (\bot, x_0, x_1, \ldots)$. The value at the first cycle is undefined ($\bot$). If $y = (y_0, y_1, \ldots)$, then $x \rightarrow y = (x_0, y_1, y_2, \ldots)$ and x **fby** $y = (x_0, y_0, y_1, \ldots)$. The initialization operator $\rightarrow$ is often used to initialize the first element of a stream under the **pre** operator (e.g., $0 \rightarrow$ **pre** x).

The condition operator is also extended to work point-wise on streams. Conceptually, **if** e_c **then** e_t **else** e_e always evaluates both e_t and e_e, and then selects according to the value of e_c. This allows Lustre to have full referential transparency, i.e., a variable inside an expression can always be substituted by its definition. In Lustre, expression evaluation is always assumed to be side effect free. Selective evaluation in Lustre is possible through the definition of multiple clocks, where the evaluation of an expression depends on the value of a boolean stream. However, experience has shown [14], that working with multiple clocks is often perceived as too difficult to comprehend by the programmer, and therefore not often used.

Mimosa retains most of the syntax of Lustre, but uses different interpretations for some of the operators. Since each component can have its own period, we shift the interpretation of streams towards sequences (which may be finite or even empty). We call the components of the network *nodes*, and each node transfers (timed) input sequences into (timed) output sequences. Analogous to Lustre, sequences are defined through equations. Most operators that combine sub-expressions do not do so point-wise in Mimosa. For example, the expression **if** e_c **then** e_t **else** e_e always evaluates e_c and the expression of the required branch respectively. This means, that **if True then** e_t **else** e_e causes the sequence produced by e_e to be empty. This removes referential transparency, but opens the possibility of expression evaluation to cause side effects. The only effect we currently model is state (i.e., memory), however, as we explain in Sect. 7 our semantics lays the foundation for dealing with other effects as well. In the remainder of this section, we introduce Mimosa by showing examples of concrete programs.

A Mimosa program is a collection of top-level definitions consisting of *steps*, *channels*, and *nodes*. A step is an elementary unit of computation, it does not have any perception of time. Steps are equivalent to function definitions in other programming languages. A step can be used inside another step, but it must be possible, during compilation, to order them in a way so that no two steps are mutually recursive. A node is an instantiation of a step as a periodically triggered process. It has a defined period, and port definitions to declare how it communicates with other nodes. A channel is a FIFO buffer, through which nodes communicate with each other. Each channel is connected to exactly one writing and one reading node. A port of a node may be marked as *optional*. If an input port to a node is marked as optional, the node can be executed even if the input channel is empty. Similarly, if an output port is marked as optional, it may happen that after a node has executed its step, there is no output written to the connected channel.

Figure 2 shows an example network adapted from [13] which calculates the Fibonacci sequence. It consists of three nodes, `add`, `split`, and `print`, each trying to execute every 10 ms. Listing 1 shows the equivalent Mimosa program. It starts with the definition of the steps, where the first is a *step prototype*. It defines only the name of the step and its signature, which will later be provided externally. For normal steps, the step signature is followed by a set of equations (same as in Lustre). The Fibonacci example defines the steps `add` and `split`, which have a trivial definition (`split` could have been equivalently defined through three separate equations). Similar to Lustre, the order of equations is not significant, as the compiler orders them automatically, and rejects a program with cyclic dependencies between equations.

After the step definitions, the channels are defined, of which two have initial elements. Finally, the three nodes are defined as instantiations of the before mentioned steps. They list which step they implement, their period, and their connections (which refer to the channels defined before).

The code shown in Listing 2 showcases the use of an optional output. The node `edge` outputs `true` whenever a rising edge in the boolean input sequence is detected, `false` when a falling edge is detected, and nothing otherwise. This can be done by wrapping the output in an option type (`out : bool?`) and declaring the output port as optional (`b?`), which means that even though the output of the step `edge` is `bool?`, the type of the channel is still `bool`. This example also illustrates the use of the **pre** and $\rightarrow$ operators to define a memory cell (in this case, to remember the value of the input from the previous execution).

Figure 4 shows an example of how this edge detector with optional output may be used in a system. Assuming `pin` polls the current level of a pin, the `edge` node then detects rising and falling edges, which it communicates to a `controller`, which can run at a lower frequency (assuming that the pin-level switches infrequently). While such a system can be implemented in the synchronous paradigm as well, the base-clock (i.e., the shortest time tick) would be constrained by the component with the smallest period. Due to the asynchronous

character of Mimosa, this issue does not arise, as individual components can have different periods without constraining the overall system.

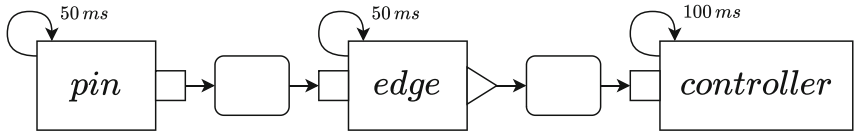


Fig. 4. Usage example for edge detector.

4 Syntax and Semantics

We now formally define the semantics of Mimosa. As the language naturally divides into two separate layers (the step and coordination layer), we give semantics to each of them individually. We start by defining the abstract syntax and semantics of steps.

4.1 Step Layer

e : Expression	::=	x	Variable
	|	c	Constant
	|	$e, \ldots, e$	Tuple
	|	**pre** e	Pre
	|	e **fby** e	Followed-by
	|	$e \rightarrow e$	Initialized-by
	|	$e\ e$	Application
	|	**if** e **then** e **else** e	Conditional
	|	**None**	None
	|	**Some** e	Some
	|	**either** e **or** e	Option match
	|	$\boldsymbol{\lambda}^{p}_{p}$**.** b	Abstraction
p : Pattern	::=	x	Variable pattern
	|	$p, \ldots, p$	Tuple pattern
b : Body	::=	$[\, p_i = e_i \,]^n$	List of n equations

Fig. 5. Abstract syntax of Mimosa expressions.

The abstract syntax of Mimosa expressions shown in Fig. 5 is actually slightly more expressive than the one defined by the concrete syntax of Mimosa, where we only allow the definition of function abstractions (i.e., steps) on the top-level.

The abstract syntax also allows nested (anonymous) functions, which allows us to treat functions as regular expressions. We expand on this choice in Sect. 7.

A Mimosa expression is either a variable, a constant, a tuple construction, a definition of memory (using the *memory operators* **pre**, **fby**, and $\rightarrow$), an application of a function to an argument, a conditional, the construction or destruction of an optional expression, or a function abstraction.

We assume a suitable set of constants and a corresponding set of basic steps for arithmetic and boolean operations to be defined in a standard library. Mimosa uses a Hindley-Milner-style type system [10,25], which we omit for brevity.

Patterns either refer to a single variable, or to a tuple of sub-patterns. In practice, we also allow _ to mean a pattern matching any value.

The abstraction $\boldsymbol{\lambda}_{p_{in}}^{p_{out}}\boldsymbol{.}\ [p_i = e_i]^n$ defines a function that requires a value compatible with pattern p_{in}, and returns a value according to pattern p_{out}. It is defined through a set of n equations, where each one is given by a left-hand pattern p_i and a right-hand expression e_i. It is assumed that equations are ordered according to dependency, are causal (i.e., any cyclic dependency is broken by a **pre**), and properly initialized (see Appendix A).

An expression can be evaluated under an environment Γ which maps names to values. We define two operations, *projection* and *update*, on environments:

Projection. The operation $\Gamma \Downarrow_p$ returns the bindings of the variable names given by pattern p. For example:

$$\begin{aligned}(x \mapsto 1, y \mapsto 2) \Downarrow_x &= 1\\ (x \mapsto 1, y \mapsto 2) \Downarrow_{(x,y)} &= (1, 2)\end{aligned}$$

Update. The operation $\Gamma \Uparrow_p^x$ returns a new environment where the value x is mapped to the name(s) given by pattern p. For example:

$$\begin{aligned}(x \mapsto 1, y \mapsto 2) \Uparrow_z^3 &= (x \mapsto 1, y \mapsto 2, z \mapsto 3)\\ (x \mapsto 1, y \mapsto 2) \Uparrow_{(x,y)}^{(3,4)} &= (x \mapsto 3, y \mapsto 4)\end{aligned}$$

As illustrated by the example above, updating an environment is destructive, i.e., previous bindings are lost.

The semantics of Lustre is usually defined denotationally as the unique least-fixed-point of the given set of stream equations. This works well for a language with full referential transparency. In Mimosa, however, the selective evaluation of certain sub-expressions makes it harder to find a denotational interpretation. We therefore express the semantics of expression evaluation in Mimosa operationally.

The evaluation relation $\Gamma \vdash e \Rightarrow v, e'$ expresses, that under an environment Γ, the expression e evaluates to value v (which is either a constant or a tuple of values), and an updated expression e', which shall be evaluated the next time the current expression is run. The full set of evaluation rules is presented in Fig. 6.

The rules for variable, constant, and tuple evaluation are trivial. The evaluation of **pre** e returns an undefined value $\perp$, where the next expression is the

$$\frac{}{\Gamma \vdash x \Rightarrow \Gamma \Downarrow_x, x}\ (\textsc{Var}) \qquad \frac{}{\Gamma \vdash c \Rightarrow c, c}\ (\textsc{Const})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1, e_1' \quad \cdots \quad \Gamma \vdash e_n \Rightarrow v_n, e_n'}{\Gamma \vdash (e_1\ ,\ \ldots\ ,\ e_n) \Rightarrow (v_1\ ,\ \ldots\ ,\ v_n), (e_1'\ ,\ \ldots\ ,\ e_n')}\ (\textsc{Tuple})$$

$$\frac{\Gamma \vdash e \Rightarrow v, e'}{\Gamma \vdash \mathbf{pre}\ e \Rightarrow \bot, (v\ \rightarrow\ \mathbf{pre}\ e')}\ (\textsc{Pre})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1, \_}{\Gamma \vdash (e_1\ \mathbf{fby}\ e_2) \Rightarrow v_1, e_2}\ (\textsc{Fby}) \qquad \frac{\Gamma \vdash e_1 \Rightarrow v_1, \_ \quad \Gamma \vdash e_2 \Rightarrow \_, e_2'}{\Gamma \vdash (e_1\ \rightarrow\ e_2) \Rightarrow v_1, e_2'}\ (\textsc{Init})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \mathbf{True}, e_1' \quad \Gamma \vdash e_2 \Rightarrow v, e_2'}{\Gamma \vdash (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \Rightarrow v, (\mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3)}\ (\textsc{IfTrue})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \mathbf{False}, e_1' \quad \Gamma \vdash e_3 \Rightarrow v, e_3'}{\Gamma \vdash (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \Rightarrow v, (\mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3')}\ (\textsc{IfFalse})$$

$$\frac{}{\Gamma \vdash \mathbf{None} \Rightarrow \mathbf{None}, \mathbf{None}}\ (\textsc{None}) \qquad \frac{\Gamma \vdash e \Rightarrow v, e'}{\Gamma \vdash \mathbf{Some}\ e \Rightarrow \mathbf{Some}\ v, \mathbf{Some}\ e'}\ (\textsc{Some})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \mathbf{Some}\ v, e_1'}{\Gamma \vdash (\mathbf{either}\ e_1\ \mathbf{or}\ e_2) \Rightarrow v, (\mathbf{either}\ e_1'\ \mathbf{or}\ e_2)}\ (\textsc{EitherSome})$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \mathbf{None}\ , e_1' \quad \Gamma \vdash e_2 \Rightarrow v, e_2'}{\Gamma \vdash (\mathbf{either}\ e_1\ \mathbf{or}\ e_2) \Rightarrow v, (\mathbf{either}\ e_1'\ \mathbf{or}\ e_2')}\ (\textsc{EitherNone})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow \boldsymbol{\lambda}_{p_{in}}^{p_{out}}.\ [p_i = e_i]^n, \_ \\ \Gamma \vdash e_2 \Rightarrow v_2, e_2' \quad \Gamma \Uparrow_{p_{in}}^{v_2} \vdash [p_i = e_i]^n \hookrightarrow [p_i = e_i']^n, \Gamma'\end{array}}{\Gamma \vdash e_1\ e_2 \Rightarrow \Gamma' \Downarrow_{p_{out}}, (\boldsymbol{\lambda}_{p_{in}}^{p_{out}}.\ [p_i = e_i']^n)\ e_2'}\ (\textsc{App})$$

$$\frac{\Gamma' \vdash e_1 \Rightarrow v_1, e_1' \quad \ldots \quad \Gamma' \vdash e_n \Rightarrow v_n, e_n'}{\Gamma \vdash [p_i = e_i]^n \hookrightarrow [p_i = e_i']^n, \Gamma' = \Gamma \Uparrow_{p_1}^{v_1} \ldots \Uparrow_{p_n}^{v_n}}\ (\textsc{Eqs})$$

Fig. 6. Structural semantics for expression evaluation.

current value of e initializing the updated **pre** e'. This ensures that the current value of e is always returned at the next evaluation cycle.

Both e_1 **fby** e_2 and $e_1 \rightarrow e_2$ return the value of e_1, however, only in the second case is e_2 also evaluated, and its value discarded (this allows, analogously to Lustre, to remove the $\bot$ values of **pre** expressions).

The rules for conditional execution are again trivial. It is important to note, that unlike in Lustre, only one branch is evaluated, while the other is kept as is in the next expression.

The expression **either** e_1 **or** e_2 evaluates the first sub-expression e_1, and in case it is of the form **Some** v returns value v, otherwise it evaluates e_2. Similar to conditional execution, e_2 is only evaluated if needed.

To evaluate an application $e_1\ e_2$, the expression e_1 is mapped to a function abstraction $\boldsymbol{\lambda}_{p_{in}}^{p_{out}}\!\!\cdot\ [p_i = e_i]^n$, and the argument e_2 to a value v. The list of equations $[p_i = e_i]^n$ is then evaluated in an environment in which v is bound according to p_{in}, which results in an updated list $[p_i = e_i']^n$, and a new environment Γ'. The value of the overall application is then the projection of p_{out} onto Γ', and the next expression is again an application, where the function term is updated with the new equation list, and the argument expression is e_2'.

To illustrate the principle of equation evaluation, we can simplify the Eqs rule to the evaluation of a single equation. At first try, the rule may be written as follows:

$$\frac{\Gamma \vdash e \Rightarrow v, e'}{\Gamma \vdash (p = e) \hookrightarrow (p = e'), \Gamma \Uparrow_p^v}\ (\text{Eqs'})$$

To evaluate an equation, one has to evaluate the expression e on the right-hand side, which results in a value v and an updated expression e'. The result of the evaluation of the whole equation is then an updated equation $p = e'$, and a new environment in which the value v is bound according to the pattern p on the left-hand side of the given equation. However, this rule falls short when trying to evaluate an equation that refers to a previous value of itself. We can illustrate that on a simple example:

$$x = 0 \ \rightarrow\ \textbf{pre}\ x$$

This defines a perfectly valid sequence of integers (i.e., the constant sequence of consecutive zeros), however, if we evaluate it according to the above given rule

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash 0 \Rightarrow 0, 0} \qquad \dfrac{\Gamma \vdash x \Rightarrow \boxed{?}, x}{\Gamma \vdash (\textbf{pre}\ x) \Rightarrow \bot, (\boxed{?} \rightarrow \textbf{pre}\ x)}\ (\star)}{\Gamma \vdash (0 \rightarrow \textbf{pre}\ x) \Rightarrow 0, (\boxed{?} \rightarrow \textbf{pre}\ x)}}{\Gamma \vdash (x = 0 \rightarrow \textbf{pre}\ x) \hookrightarrow (x = \boxed{?} \rightarrow \textbf{pre}\ x), \Gamma \Uparrow_x^0}$$

we see that there is an issue when trying to evaluate the **pre** at $(\star)$. In the example above, $\boxed{?}$ represents a hole in the expression, for which we would need the value of x. But this value is not yet bound in Γ, as it is being defined by the equation currently under evaluation. Therefore, we need to evaluate the right-hand side expression of an equation under an environment that already has the final value of the evaluation bound according to the pattern on the left-hand side:

$$\frac{\Gamma' \vdash e \Rightarrow v, e'}{\Gamma \vdash (p = e) \hookrightarrow (p = e'), \Gamma' = \Gamma \Uparrow_p^v}\ (\text{Eqs''})$$

With this rule, we get the correct result which we formulate in the lemma below:

$$\dfrac{\dfrac{}{\Gamma' \vdash 0 \Rightarrow 0, 0} \quad \dfrac{\Gamma' \vdash x \Rightarrow 0, x}{\Gamma' \vdash \mathbf{pre}\ x \Rightarrow \bot, (0 \rightarrow \mathbf{pre}\ x)}}{\dfrac{\Gamma' \vdash 0 \rightarrow \mathbf{pre}\ x \Rightarrow 0, (0 \rightarrow \mathbf{pre}\ x)}{\Gamma \vdash (x = 0 \rightarrow \mathbf{pre}\ x) \hookrightarrow (x = 0 \rightarrow \mathbf{pre}\ x), \Gamma' = \Gamma \Uparrow_x^0}}$$

Lemma 1. *Under the assumption that* $\Gamma \vdash e \Rightarrow v$, e' *defines* v *and* e' *uniquely, the evaluation defined by* Eqs" *is also unique, i.e., if* $\Gamma \vdash (p = e) \hookrightarrow (p = e'_1), \Gamma'_1$ *and* $\Gamma \vdash (p = e) \hookrightarrow (p = e'_2), \Gamma'_2$ *then* $e'_1 = e'_2$ *and* $\Gamma'_1 = \Gamma'_2$. *This generalizes to the rule for multiple equations (*Eqs*).*

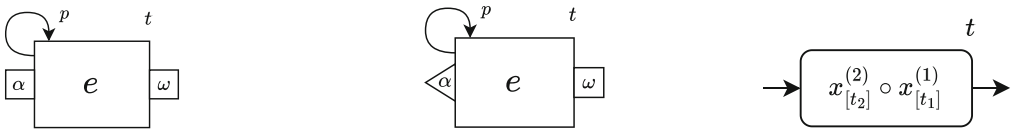
Proof. The rule Eqs" can only lead to diverging results if there are multiple valid candidates Γ' which bind the current value of the sequence being defined. Since the value of e cannot depend causally on any variable bound in p (i.e., any reference on the right-hand side of the equation can only refer to variables in the left-hand side under a **pre** operator), the value is uniquely defined ($v_1 = v_2$), and therefore the updated environment must be as well. The general Eqs rule follows by induction on the list of equations. □

Theorem 1. *The step evaluation relation* $\Gamma \vdash e \Rightarrow v, e'$ *is deterministic, i.e., if* $\Gamma \vdash e \Rightarrow v_1, e'_1$ *and* $\Gamma \vdash e \Rightarrow v_2, e'_2$ *then* $v_1 = v_2$ *and* $e'_1 = e'_2$.

Proof. By structural induction on the syntax of expressions. Most cases are trivial, the case for the App rule follows from Lemma 1. □

4.2 Coordination Layer

With the semantics for expressions in place, we can now define the semantics of the coordination layer. While it is possible to give the semantics of the coordination layer as a textual rewriting calculus as well, it is easier to define and understand as a graph-rewriting system.



(a) Node (b) Node with optional input (c) Channel

Fig. 7. Examples of graph symbols for the coordination layer semantics.

Figure 7 shows examples of the used graph symbols. The first example shows a node with period p, which tries to execute expression e at its next *activation time* t. It communicates with other nodes through inputs taken from a channel

at input α, and puts its result in a channel at output ω. Figure 7b shows the same node with an optional input α instead.

Figure 7c depicts a channel with two elements inside. It has a time-tag t, which we refer to as its *validity time*. Each element inside the channel also has a time tag assigned to it. The validity time of a channel is intended to be the *earliest possible time-tag* of the next value written into it. We write the elements inside a channel as a sequence $\rho = x^{(n)}_{[t_n]} \circ x^{(n-1)}_{[t_{n-1}]} \circ \cdots \circ x^{(1)}_{[t_1]}$, with the following invariants:

- $\forall i.\, t_i \leq t$, i.e., the time-tags of all elements in the channel must be smaller or equal to the validity time of the channel.
- Elements inside the channel are always ordered according to their time-tags, i.e., the oldest item (the one with the smallest time-tag) is the right-most item in the sequence.

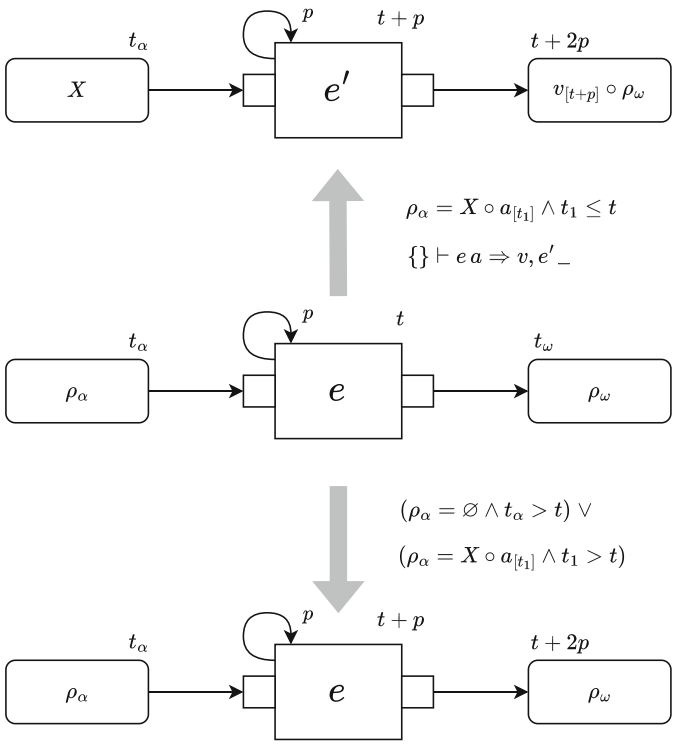


Fig. 8. Rewriting rule for FIFO input.

In the current version of Mimosa, a node can only write either none or exactly one element into each output channel. In Sect. 7 we discuss how to eliminate this restriction, in which case multiple elements in a channel may have the same time-tag. This is not a problem for the semantics, as the insertion order is preserved by the order of elements in the sequence representing a channel. By slight abuse of notation, we use the infix operator $\circ$ both for appending elements on either end of the sequence, and for forming sequences out of individual elements themselves.

To ease comprehension, we only present the rewrite rules for nodes with one input and one output port. The general rules for multiple input/output are shown in Appendix B. Figure 8 illustrates the rewriting rules for a node connected to two channels through its input and output port. The upper rule expresses, that if at time t (i.e., the time at which the node tries to execute next) the input channel has a right-most (i.e., oldest) element $a_{[t_1]}$ for which $t_1 \leq t$ (i.e., the time-tag of the element a is not in the future of the node), then the expression $e\,a$ can be evaluated. This leads to a value v, which is put in the output channel with time-tag $t + p$ (i.e., the end of the current period), and the validity time of the output channel can be updated to $t + 2p$, which is the earliest time at which a new element may be put into the channel. After the rewrite step, the input channel has its right-most element removed, the expression of the node is updated, the next activation time set to $t + p$, and the expression of the node rewritten to e' (we can ignore the rewritten argument).

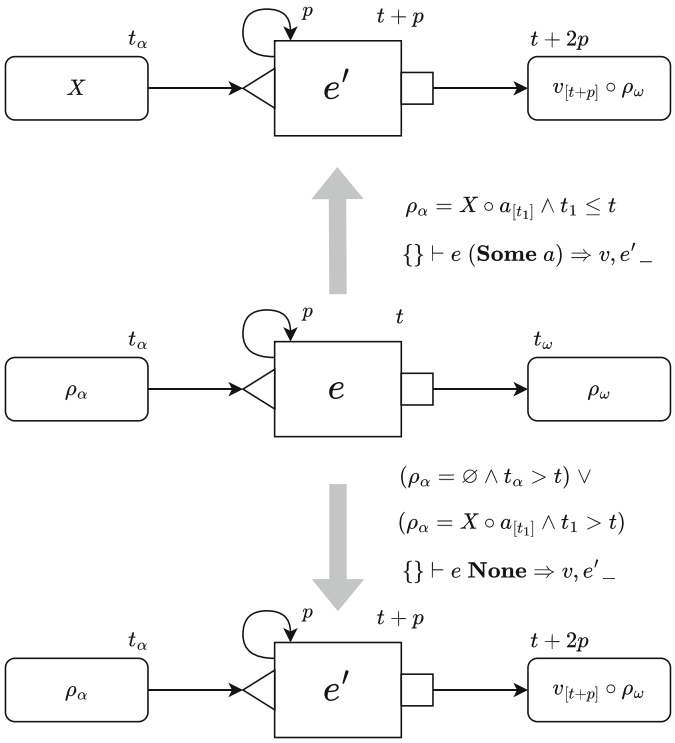


Fig. 9. Rewriting rule for optional input.

The lower rule expresses an idle step. If the input channel is empty, but its validity time t_α is larger than the activation time t of the node, or if the right-most element has a time-tag larger than t (i.e., from the perspective of the node it is in the future), then the next activation time of the node can be set to $t + p$, and the validity time of the output buffer can be updated to $t + 2p$.

Figure 9 shows the rewriting rules for a node with optional input, which means its activation is never blocked. In the upper case, there is a valid input in the input channel, i.e., it has a time-tag smaller or equal to the current node

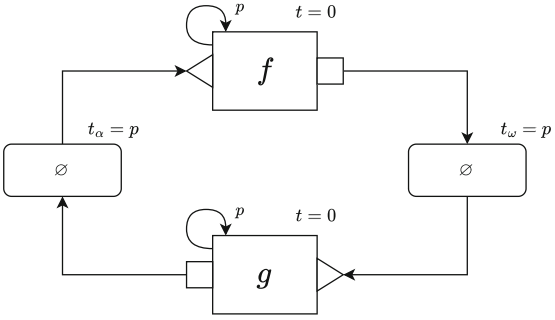


Fig. 10. Example of a correct initial configuration.

activation time. Therefore, the expression e (**Some** a) is evaluated. In the lower case, no valid input value is present, so either the input buffer is empty but the channel is valid until the current node activation time, or the time-tag of the oldest element is in the future. In this case, the expression e **None** is evaluated. The rest is analogous to the rules presented before.

To guarantee the invariant that the validity time is the earliest time point at which a value can be written to a channel (i.e., every $a_{[t']}$ added to a channel with validity time t guarantees $t' \geq t$), we initialize each channel with the first possible writing time of its respective writing node. An example of a correct initialization of a network with initially empty buffers is shown in Fig. 10.

Theorem 2. *The rewriting semantics described above is confluent, i.e., every possible sequence of rewritings leads to the same (timed) history of values appearing in each channel.*

Proof. The rewriting rules are local (two possible rewriting candidates can only overlap on a common channel). Assume that there are two different nodes A and B, where A outputs into a channel σ which is the input to B, and that t_B is the next activation time for B. Let t_σ be the validity time of the common channel, which is always the earliest time point at which A can write, an invariant maintained by the rules. If $\sigma = \varnothing$, but $t_\sigma > t_B$, then executing B first causes an idle step. The same happens if node A is executed first, as the output of A can only appear at or after t_σ (it cannot be earlier than the already established validity time of the channel). The case where σ is not empty is trivial, as executing A first only puts another element in the channel, which has no impact on the execution of B. In all other cases (e.g. $t_\sigma < t_B$) no rule can be applied to B, and A must execute first. □

The rules explained above generalize naturally to the case of multiple inputs and outputs, as well as to optional ports (they only change the writing behaviour, never the activation conditions for a node). The coordination layer semantics makes no assumption about the order of rewriting[1]. This may be exploited when

[1] note that, strictly speaking, we do not even need a notion of "global time", at least as long as there are no real-time dependent side effects. This fact may be exploited for functional verification.

verifying different scheduling strategies for execution on real hardware, which will in most cases have to sequentially schedule node executions on limited hardware resources.

5 Simulation

As a first means to simulate programs written in Mimosa, we have implemented the language through a deep-embedding in the programming language OCaml. We have implemented an expression evaluation function which is a direct translation of the semantic evaluation rules given in Fig. 6. This not only adds confidence in the correctness of the implementation, it also allowed us to find (often intricate) errors in the rules early on.

The steps introduced by step prototypes may cause side effects (such as the `print_int` in Listing 1). Therefore, care must be taken during simulation regarding their (temporal) ordering. Since the coordination semantics covers all possible scheduling orders, we have opted to implement it as a discrete-event simulator which constructs the global timeline of node executions assuming perfect timings (i.e., no jitter and no clock drift). In case the execution of multiple nodes at the same time point leads to multiple observable side effects (such as printing), the order of these effects is undefined.

We provide a prototype compiler, which can translate a Mimosa program into the mentioned OCaml embedding. This always creates an OCaml module (similar to a package in other programming languages) with a specific signature:

```
module type Simulation = sig
    type t
    val exec_ms : t -> unit
    val exec : t -> int -> unit
    val init : unit -> t
end
```

A particular simulation run is a value of type `Simulation.t`, which is created by a call to `Simulation.init ()`. After creation, the timeline of a simulation run can be advanced through `Simulation.exec_ms` and `Simulation.exec` by one or multiple milliseconds, respectively.

If the Mimosa program defines prototypes (i.e., external steps), the output of the compiler is instead a functor, i.e., a function taking a module defining all the external steps through corresponding OCaml functions, and returning a module of the above given `Simulation` type. This allows using the same code for both interactive simulation and automated tests by instantiating the functor with different argument modules. Examples are shown in the documentation of the compiler [18].

6 Related Work

Lingua Franca (LF) [21] is a polyglot framework for the design of hard real-time systems. As it is based upon the Reactor model [22], computation is triggered by the occurrence of events, which are time-stamped data items. As a polyglot language, LF is meant to augment other programming languages with a coordination layer based on discrete event semantics. This allows for easy integration of libraries and other software components already written in particular mainstream languages. However, it makes formal reasoning about a particular program behaviour difficult, as one would need formal semantics for each language used in the implementation. Marin et al. [24] have formalized the semantics of LF as a rewriting logic. Together with a simple imperative language for reactions, they show an embedding in Maude [7], which offers the opportunity for model checking analysis. Deantoni et al. [11] provide advanced tooling for LF by incorporating the coordination semantics into GEMOC [5], an Eclipse-based language and modelling workbench. By including an abstract execution model of LF into GEMOC, an interactive debugger is created, and execution traces can be injected into the workbench to analyse and validate traces obtained from compiled LF programs written in different languages. A similar approach may be interesting for future work on Mimosa as well.

Giotto [16] is a programming language for real-time systems, where a program describes a set of periodic tasks that communicate through ports, which always keep the last value written to them. Giotto has a particular focus on execution modes, where the system behaviour is governed by some global mode. Giotto's ports offer a similar communication primitive as MIMOS registers. In later work [20], a microkernel has been introduced as a compilation target, which separates the execution of code interacting with the environment [17] and the scheduling of tasks in a system. A similar approach may be interesting for the implementation of Mimosa as well.

The synchronous programming paradigm introduces into software the same synchronous abstractions as those used for digital circuit design, where a global system tick abstracts over the timing characteristics of the underlying electronics. By shifting the timing domain from physical real-time to logical ticks, temporal reasoning about the behaviour of a program becomes easier, and correctness can be verified independently of the execution platform, as long as the execution platform can guarantee the synchronous hypothesis. This becomes difficult on heterogenous or distributed hardware, where jitter and clock-drift may happen. The PALS (Physically Asynchronous Logically Synchronous) [30] protocol has been proposed as an implementation strategy, where the programmer can use the synchronous abstraction during implementation, and a middleware runtime [1] takes care of the execution on distributed hardware.

Various extensions to multi-rate settings for synchronous languages have been proposed. In some of them [12,28], special rate transition operators are introduced to describe the communication policy between two nodes with different execution rates. This is syntactically quite heavy, and we believe that the asynchronous communication channels with buffering in Mimosa are easier to grasp

for a programmer. In addition, optional inputs and outputs allow modelling sporadic events, which is difficult with fixed communication policies. Bourke et al. [4] introduced an extension to the synchronous-reactive model, which allows specifying multi-rate systems. It offers features for load-balancing, resource-limiting, and even specifying end-to-end delay constraints, which they compile to a single periodic task. MIMOS, and by extension Mimosa, avoids many of the problems stemming from multiple execution rates by shifting to an asynchronous model of computation in the first place.

The Timed-C compiler [27] is a source to source compiler, which extends the C programming language with a set of timing primitives. It defines tasks which communicate through channels, similar to the ones proposed for MIMOS, and compiles to plain C on top of a real-time operating system. As a general purpose language, C is very expressive, however, it is also known for its intricacies, which makes reasoning about the behaviour of C programs difficult. We advocate, that a language for embedded systems rarely needs this degree of expressiveness, and that restricting the language to a small, well-defined kernel simplifies the verification of systems programmed in it.

7 Conclusion and Future Work

In this work, we have introduced Mimosa, a new programming language for embedded systems software on top of the MIMOS computational model. The focus of this paper is on the definition of a formal semantics, and therefore, the presented language kernel has been kept minimal. To facilitate the implementation of real systems in Mimosa, the language is intended to be extended in multiple ways:

The current implementation of Mimosa is a prototype simulator which allows for defining test cases and experimenting with the language. We are working on a compiler that translates Mimosa expressions into equivalent C code and translates the coordination layer into a set of tasks for a real-time operating system. This will allow us to run Mimosa programs on real embedded hardware. For simulation purposes, we can assume the channels to have infinite capacity, for running Mimosa programs on real hardware requires to know the bounds of the channels beforehand. Part of the MIMOS project [32] is therefore the development of algorithms for this kind of buffer-size estimation.

This paper presents a minimal language kernel. For a fully-fledged programming language, Mimosa needs to be extended with a module system, a standard library, and potentially a more expressive type system (including enumeration and record types). The concrete syntax of Mimosa currently only allows for function abstraction at the top-level, even if the abstract syntax already includes the potential for nested functions. Extending the syntax to include (anonymous) function definitions would allow us to treat functions as first-class values, which can even be transmitted through channels.

To keep the graphical rewriting rules simpler, the current coordination layer semantics does not include registers defined in the original MIMOS paper [31].

Registers always keep only the latest value written to them, and are therefore particularly useful as a communication link between sensors and controllers, as the sensor can then run at a higher (or lower) frequency. Registers never block the activation of a node, the only addition to the rewriting rules concerns how values are read from them. Registers can be modelled analogously to channels, where reading does not remove a value from the channel, but only looks up the value item with the largest time-tag smaller or equal than the current activation time of the node.

The optional ports presented in this paper are actually a special case of the more general *up-to* ports [32], which allow reading or writing multiple data items from or to a channel. The current coordination layer semantics can be extended to cover these types of reading and writing strategies as well, which eases the communication between nodes with different periods. The expression language would then need to be extended with operators on lists of values (such as the typical *map* and *fold* operators known from functional programming languages). This is similar to the extension of Lustre with array iterators [26].

In Mimosa, certain expressions, such as conditionals, only selectively evaluate their sub-expressions, which in turn removes the referential transparency of variable names. We have chosen this design to provide a similar mechanism to Lustre's multi-clocks, without the added mental complexity. It also lays the foundations for facilitating expression evaluation with side effects. The only effect we currently model is memory, however, recent programming languages [6,23,29] have shown the advantage of being able to express side effects of a program in terms of types. Being able to track the effects an expression may exhibit during evaluation offers new opportunities for optimization (e.g., application of functions to constants during compile time). It also allows for speculative execution, where a node without observable side effects can execute even before its release time in case it has all its required inputs. This can ultimately lead to better utilization of processors, while keeping the functional output of the system the same. We therefore intend to extend a future version of Mimosa with effect types.

Acknowledgments. This work was partially funded by ERC through project CUSTOMER and by the Knut and Alice Wallenberg Foundation through project UPDATE.

A Initialization Analysis

Initialization analysis is part of the frontend of the Mimosa compiler [18]. It is responsible for proving that the undefined values at the start of sequences, which are introduced by **pre** operators, do not affect the output of a step.

The analysis performed by the Mimosa compiler is similar to the one in Lustre [9], however, the selective evaluation of certain sub-expressions leads to different requirements for proper initialization.

For example, in Lustre the following holds:

$$\textbf{if } a \textbf{ then } (0 \rightarrow b) \textbf{ else } (0 \rightarrow c) \equiv 0 \rightarrow (\textbf{if } a \textbf{ then } b \textbf{ else } c)$$

This does not hold in Mimosa, as the two branches are selectively evaluated depending on the value of the condition expression. In general, both branches of a conditional expression need to be properly initialized.

In Lustre, **pre** statements may also be nested to refer to values of a stream from multiple cycles before:

$$0 \rightarrow 0 \rightarrow \mathbf{pre}\ \mathbf{pre}\ x$$

This expression leads to an undefined value at the second cycle in Mimosa:

$$\dfrac{\dfrac{}{\Gamma \vdash 0 \Rightarrow 0, \_} \qquad \dfrac{\dfrac{\dots}{\Gamma \vdash \mathbf{pre}\ x \Rightarrow \bot, \dots}}{\Gamma \vdash (\mathbf{pre}\ \mathbf{pre}\ x) \Rightarrow \_, (\bot \rightarrow \dots)}}{\dfrac{\dfrac{}{\Gamma \vdash 0 \Rightarrow 0, \_} \qquad \Gamma \vdash (0 \rightarrow \mathbf{pre}\ \mathbf{pre}\ x) \Rightarrow \_, (\bot \rightarrow \dots)}{\Gamma \vdash (0 \rightarrow 0 \rightarrow \mathbf{pre}\ \mathbf{pre}\ x) \Rightarrow 0, (\bot \rightarrow \dots)}}$$

Nested **pre** statements are nevertheless allowed in Mimosa, $\rightarrow$ and **pre** need to be used alternately, which leads to the expected behaviour:

$$\dfrac{\dfrac{}{\Gamma \vdash 0 \Rightarrow 0, \_} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash x \Rightarrow \Gamma \Downarrow_x, x}}{\Gamma \vdash \mathbf{pre}\ x \Rightarrow \bot, (\Gamma \Downarrow_x \rightarrow \mathbf{pre}\ x)}}{\Gamma \vdash (0 \rightarrow \mathbf{pre}\ x) \Rightarrow 0, (\Gamma \Downarrow_x \rightarrow \mathbf{pre}\ x)}}{\dfrac{\dfrac{}{\Gamma \vdash 0 \Rightarrow 0, \_} \qquad \Gamma \vdash \mathbf{pre}\ (0 \rightarrow \mathbf{pre}\ x) \Rightarrow \bot, (0 \rightarrow \Gamma \Downarrow_x \rightarrow \mathbf{pre}\ x)}{\Gamma \vdash (0 \rightarrow \mathbf{pre}\ (0 \rightarrow \mathbf{pre}\ x)) \Rightarrow 0, (0 \rightarrow \Gamma \Downarrow_x \rightarrow \mathbf{pre}\ x)}}$$

Given this requirement, each sequence in Mimosa can only have one of two possible initialization types: initialized or uninitialized. In order to further simplify the reasoning about programs, we also require that step inputs and outputs are always initialized.

B Node-Level Rewriting Rules

(See Figs. 11 and 12).

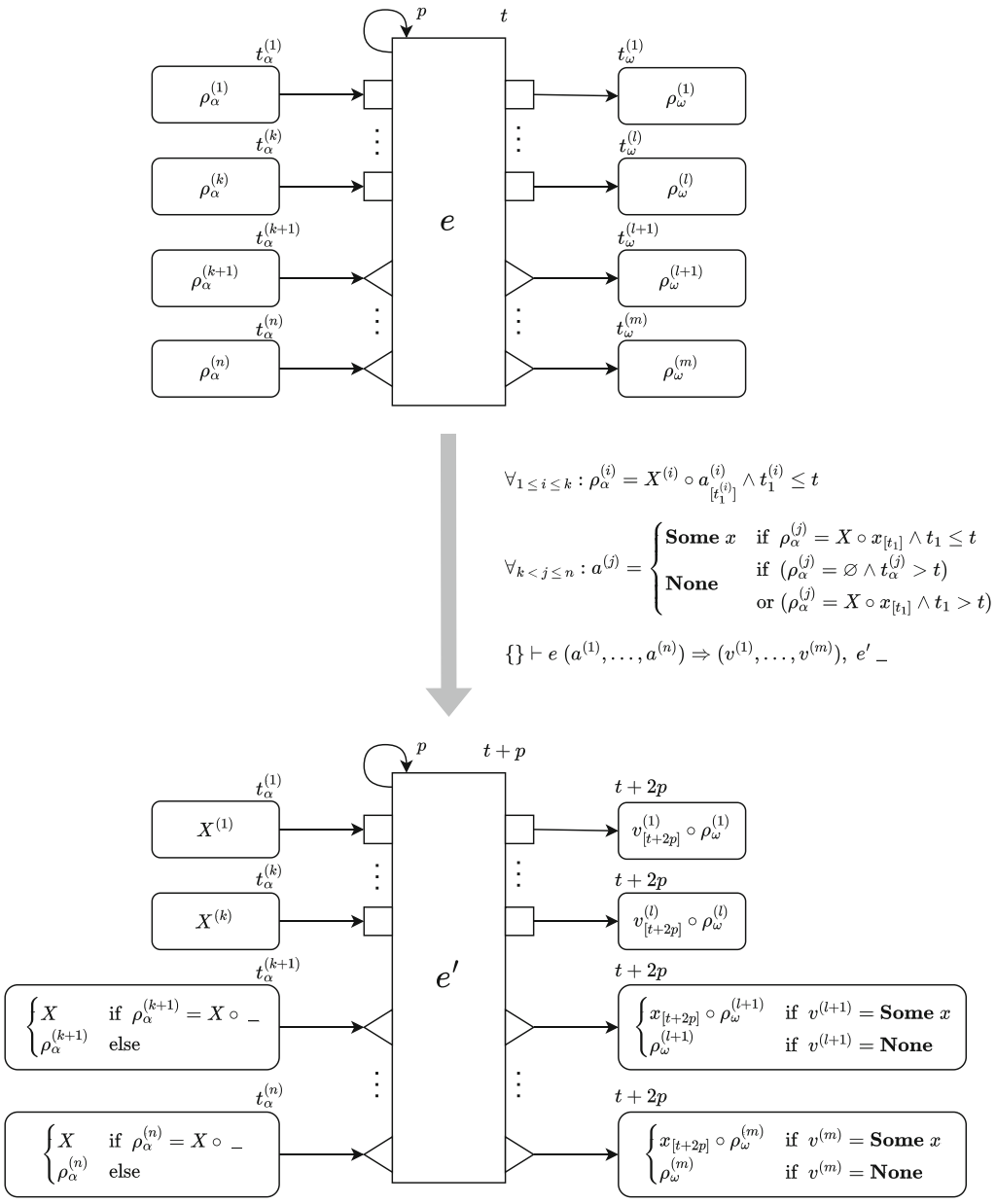


Fig. 11. General rewriting rule for node execution.

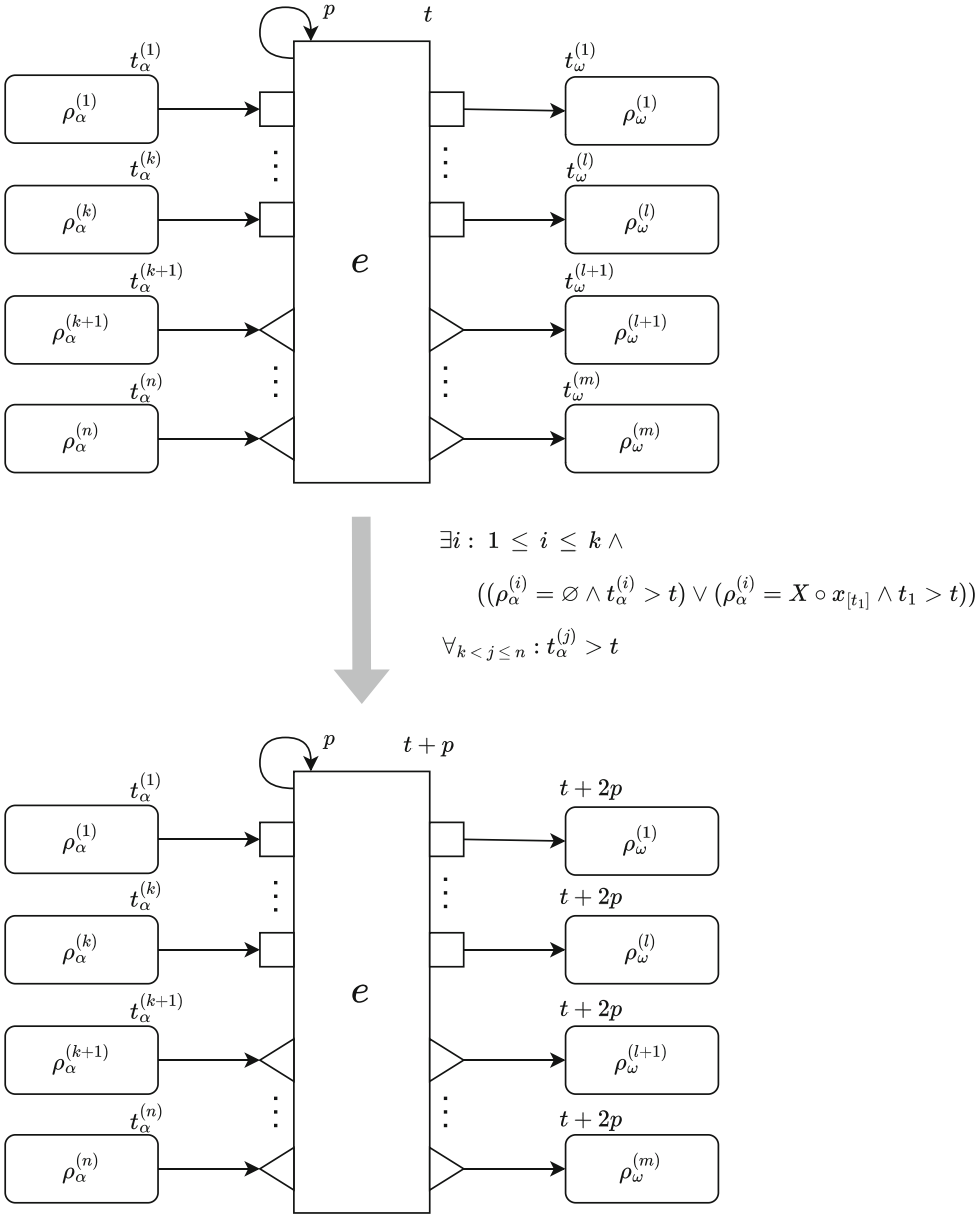


Fig. 12. General rewriting rule for idle step.

References

1. Al-Nayeem, A., Kim, C., Kang, W., Wu, P.L., Sha, L.: Middleware design for physically-asynchronous logically-synchronous (PALS) systems. In: 2013 Proceedings of the International Conference on Embedded Software (EMSOFT), pp. 1–10 (2013). https://doi.org/10.1109/EMSOFT.2013.6658583
2. Benveniste, A., Bournai, P., Gautier, T., Le Borgne, M., Le Guernic, P., Marchand, H.: The signal declarative synchronous language: controller synthesis and systems/architecture design. In: Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228), vol. 4, pp. 3284–3289 (2001). https://doi.org/10.1109/CDC.2001.980328

3. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992). https://doi.org/10.1016/0167-6423(92)90005-V
4. Bourke, T., Bregeon, V., Pouzet, M.: Scheduling and compiling rate-synchronous programs with end-to-end latency constraints. In: Papadopoulos, A.V. (ed.) 35th Euromicro Conference on Real-Time Systems (ECRTS 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 262, pp. 1:1–1:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2023). https://doi.org/10.4230/LIPIcs.ECRTS.2023.1
5. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the GEMOC studio (tool demo). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, pp. 84–89. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2997364.2997384
6. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effects as capabilities: effect handlers and lightweight effect polymorphism. Proc. ACM Program. Lang. **4**(OOPSLA) (2020). https://doi.org/10.1145/3428194
7. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
8. Colaço, J.L.: An overview of Scade, a synchronous language for safety-critical software (keynote). In: Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS 2020, p. 1. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3427763.3432350
9. Colaço, J.L., Pouzet, M.: Type-based initialization analysis of a synchronous dataflow language. Int. J. Softw. Tools Technol. Transf. **6**(3), 245–255 (2004). https://doi.org/10.1007/s10009-004-0160-y
10. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1982, pp. 207–212. Association for Computing Machinery, New York (1982). https://doi.org/10.1145/582153.582176
11. Deantoni, J., Cambeiro, J., Bateni, S., Lin, S., Lohstroh, M.: Debugging and verification tools for LINGUA FRANCA in GEMOC studio. In: 2021 Forum on specification & Design Languages (FDL), pp. 01–08 (2021). https://doi.org/10.1109/FDL53530.2021.9568383
12. Forget, J., Boniol, F., Lesens, D., Pagetti, C.: A real-time architecture design language for multi-rate embedded control systems. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC 2010, pp. 527–534. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1774088.1774196
13. Geilen, M., Basten, T.: Kahn process networks and a reactive extension. In: Bhattacharyya, S., Deprettere, E., Leupers, R., Takala, J. (eds.) Handbook of Signal Processing Systems, pp. 967–1006. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-6345-1_34
14. Halbwachs, N.: A synchronous language at work: the story of Lustre. In: Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE 2005, pp. 3–11 (2005). https://doi.org/10.1109/MEMCOD.2005.1487884
15. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language Lustre. Proc. IEEE **79**(9), 1305–1320 (1991). https://doi.org/10.1109/5.97300

16. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 166–184. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45449-7_12
17. Henzinger, T.A., Kirsch, C.M.: The embedded machine: predictable, portable real-time code. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI 2002, pp. 315–326. Association for Computing Machinery, New York (2002). https://doi.org/10.1145/512529.512567
18. Huber, N.: The mimosa simulator - software artifact (2025). https://doi.org/10.5281/zenodo.14963241
19. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, 5–10 August 1974, pp. 471–475. North-Holland (1974)
20. Kirsch, C.M., Sanvido, M.A.A., Henzinger, T.A.: A programmable microkernel for real-time systems. In: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, VEE 2005, pp. 35–45. Association for Computing Machinery, New York (2005). https://doi.org/10.1145/1064979.1064986
21. Lohstroh, M., Menard, C., Schulz-Rosengarten, A., Weber, M., Castrillon, J., Lee, E.A.: A language for deterministic coordination across multiple timelines. In: 2020 Forum for Specification and Design Languages (FDL), pp. 1–8 (2020). https://doi.org/10.1109/FDL50818.2020.9232939
22. Lohstroh, M., et al..: Reactors: a deterministic model for composable reactive systems. In: Chamberlain, R., Edin Grimheden, M., Taha, W. (eds.) Cyber Physical Systems. Model-Based Design, pp. 59–85. Springer, Cham (2020)
23. Madsen, M.: The principles of the Flix programming language. In: Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, pp. 112–127. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3563835.3567661
24. Marin, M., Ölveczky, P.C., Reja, M., Rukhaia, M., Bae, K.: Semantics and formal analysis of lingua franca cps specifications in rewriting logic. In: Lee, E.A., Mousavi, M.R., Talcott, C. (eds.) Rebeca for Actor Analysis in Action. LNCS, vol. 15560, pp. 70–101. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-85134-6_4
25. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978). https://doi.org/10.1016/0022-0000(78)90014-4
26. Morel, L.: Efficient compilation of array iterators for Lustre. Electron. Notes Theor. Comput. Sci. **65**(5), 19–26 (2002). https://doi.org/10.1016/S1571-0661(05)80437-2. SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002)
27. Natarajan, S., Broman, D.: Timed C: an extension to the C programming language for real-time systems. In: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 227–239 (2018). https://doi.org/10.1109/RTAS.2018.00031
28. Pagetti, C., Forget, J., Boniol, F., Cordovilla, M., Lesens, D.: Multi-task implementation of multi-periodic synchronous programs. Discrete Event Dyn. Syst. **21**(3), 307–338 (2011). https://doi.org/10.1007/s10626-011-0107-x

29. Reinking, A., Xie, N., de Moura, L., Leijen, D.: Perceus: garbage free reference counting with reuse. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, pp. 96–111. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3453483.3454032
30. Sha, L., Al-Nayeem, A., Sun, M., Meseguer, J., Olveczky, P.C.: PALS: physically asynchronous logically synchronous systems. Technical report, University of Illinois at Urbana-Champaign (2009)
31. Yi, W., Mohaqeqi, M., Graf, S.: MIMOS: a deterministic model for the design and update of real-time systems. In: ter Beek, M.H., Sirjani, M. (eds.) COORDINATION 2022. IFIP Advances in Information and Communication Technology, vol. 13271, pp. 17–34. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-08143-9_2
32. Yi, W., et al.: MIMOS in a nutshell, in preparation