A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs

Gwenaël Delaval
INRIA
Rennes, France
Gwenael.Delaval@inria.fr

Alain Girault
INRIA Rhône-Alpes
Grenoble, France
Alain.Girault@inria.fr

Marc Pouzet
LRi, Univ. Paris-Sud 11
Orsay, France
Marc.Pouzet@lri.fr

Abstract
We address the design of distributed systems with synchronous dataflow programming languages. As modular design entails handling both architectural and functional modularity, our first contribution is to extend an existing synchronous dataflow programming language with primitives allowing the description of a distributed architecture and the localization of some expressions onto some processors. We also present a distributed semantics to formalize the distributed execution of synchronous programs. Our second contribution is to provide a type system, in order to infer the localization of non-annotated values by means of type inference and to ensure, at compilation time, the consistency of the distribution. Our third contribution is to provide a type-directed projection operation to obtain automatically, from a centralized typed program, the local program to be executed by each computing resource. The type system as well as the automatic distribution mechanism has been fully implemented in the compiler of an existing synchronous dataflow programming language.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming—Distributed programming; D.3.2 [Programming Languages]: Languages—Concurrent, distributed, and parallel languages

General Terms Languages

Keywords Synchronous programming, distribution, type systems, functional programming

1. Motivations
Synchronous programming languages \cite{5} are frequently used in the industry for the design of real-time embedded systems. Such languages define deterministic behaviors and lie on formal semantics, making them suitable for the design and implementation of safety critical systems. They are used, for example, in critical domains such as the automotive, avionics, or nuclear industry.

Most of the systems designed with synchronous languages are centralized systems. The parallelism expressed in these languages is a functional one, whose purpose is to ease the design process by providing ideal timing and concurrency constructs to the designers. A synchronous program is then compiled into a sequential program emulating the parallel execution of the functional parallel branches. This sequential program is executed on a single computing resource. Yet, most embedded systems are composed of several computing resources (named "locations"), for reasons such as performance, dedicated actuators or sensors drivers, or adaptivity of the locations to the tasks they are assigned to (e.g., pure computing tasks vs control tasks). We call this the execution parallelism. This paper addresses the problem of mapping the functional parallelism onto the execution one, in a modular way. We focus on distributed systems implemented as networks of deterministic processes communicating with FIFOs.

Fragments of a distributed system can be designed separately, but in complex and multifunctional embedded systems, functionalities are frequently independent of the hardware architecture, implying conflicts between architectural and functional modularity. Thus, one functionality can use several locations and one location can be involved in several functionalities. As a result, programming each location separately compromises the modularity and is error-prone. This situation occurs within several industrial areas, such as automotive embedded systems, and software-defined radio \cite{15}.

Our paper is organized as follows: Section \ref{sec:overview} gives an overview of the context, and motivates our method through some examples and one application. Section \ref{sec:semantics} presents the semantics and the formalization of the spatial type system. Section \ref{sec:projection} presents the projection operation, which is our third contribution. Finally, related work and discussion about the solution will be exposed in Section \ref{sec:related_work}.

2. Overview

2.1 Distribution of Synchronous Dataflow Programs
Synchronous dataflow languages, such as Lustre, Signal, or Lucid Synchrone \cite{5}, manipulate infinite streams of values as primitive values: the notation \( \mathbb{N} \) represents the infinite stream \( 1, 1, \ldots \), while \( \text{int} \) stands for the type of infinite streams of integers. For any stream \( x \), we note \( x_i \) its \( i \)th value. In this context, functions (called nodes hereafter) are stream functions: e.g., \( \text{int} \rightarrow \text{int} \) is the type of functions from integer streams to integer streams. Combinatorial functions are implicitly lifted to apply pointwise to their arguments: e.g., if \( x = (x_i)_{i \in \mathbb{N}} \) and \( y = (y_i)_{i \in \mathbb{N}} \) are two integer streams, then \((x + y) = (x_i + y_i)_{i \in \mathbb{N}}\). Moreover, we use a unitary delay, noted \( \text{delay} \), such that \( \text{delay}(x)_{i} = x_{i-1} \) otherwise.

Such a program is classically compiled into a single function \( f \), which computes the values of outputs and updates the system’s state, from the values of inputs and the current state. This function \( f \) is then embedded inside a periodic execution loop. Our contribution is to extend this classical compilation scheme to a distributed framework: the result of the compilation of a distributed system will consist of \( n \) functions \( f_i \), one for each location \( i \), which will compute the values of outputs, communication channels, and lo-
cal state, from the values of inputs, other incoming communication channels, and the current local state.

### 2.2 Language-based Distribution

We address functional distribution, not achieved for the sake of performance but because the system is intrinsically distributed. Distribution is driven by the fact that some functions have a meaning only at some specific locations and not at others. We can think, e.g., of a function returning the value of a physical sensor and which has to be executed where the sensor is. Therefore, locations will be defined by the functionalities they provide.

Designing such distributed systems is non-trivial, because of problems such as the scheduling of communications or the type consistency of the communicated data. The usual method, using architectural languages like AADL [2], involves describing the system’s architecture by partitioning it in subsystems. Each subsystem can then be defined separately, possibly with different languages. However, in the case of tightly dependent subsystems, where conflicts between architectural and functional modularity can occur, it is less error-prone and more efficient to define the system as a whole, together with architectural annotations. Our first contribution is to provide language primitives to allow the programmer to describe the architecture, and to express where some values are located, i.e., on which location some computations are performed.

The architecture is described by the explicit declaration of the set of existing locations and the links between them. At this point, locations are symbolic: a location declaration introduces a symbolic name, which will then be used to express the fact that a stream is computed or available at this symbolic location. We define in Section 2.3 a projection operation which produces, for each symbolic name, a single non-distributed synchronous program to be executed at the physical location represented by this symbolic name.

The syntax for declaring the physical location A is `loc A`. The existence of a communication link from A to B is declared by `link A to B`. Note that we distinguish communication links from communication channels, introduced in Section 2.4. Communication links, specified by the primitive `link`, state the ability to communicate from one location to another. In contrast, actual channels used by the distributed system are inferred by the type system.

The statement `e at A` means that every value used in the expression `e` (streams and nodes composing its subterms) will be located at A. The programmer does not need to express the localization of every value. Our second contribution is to provide a type and effects system [19] whose double function is to check the validity of the localization expressed w.r.t. the architecture, and to infer the localization of non-explicitly located values. For instance, the node `f` given below consists of two computations `g` and `h`, respectively located by the programmer on locations `A` and `B`, thanks to the `at A` and `at B` annotations.

```
node f(x) = z with
  y = g(x) at A
and z = h(y) at B;
```

Communications are abstracted, and thus not expressed by the programmer, neither technically, nor concerning their place inside the code. The technical expression of communications is left to the further phase of integration on actual architecture. Our method only deals with inferring the localization of these communications, and their coherence throughout the distributed code. We assume for now that communications can occur at any localization, and can concern any value entirely concealed within a location (i.e., not the distributed data structures, like distributed pairs). From a programmer’s point of view, this choice is a compromise between no control at all (communications are possible everywhere) and absolute control (the programmer expresses every communication).

### 2.3 A Spatial Type System for Automatic Distribution

We place ourselves in a functional framework, where for the sake of modularity, functions can neither be inlined nor analyzed dependently of their calling context. We provide a special type system dedicated to the distributed execution of the program, as an analysis provided to the programmer to help ensure the consistency of the distribution specification. We call it a spatial type system and, when clear from context, we shall simply refer to it as a type system. This spatial type system describes the localization of streams, and a type-directed approach is followed to achieve code distribution. This also allows us to preserve higher-order features, hence allowing the expression of dynamic reconfiguration of nodes by application of other nodes as inputs.

The other motivation for using a type system is to achieve type inference: in order not to force the programmer to specify everything (i.e., the localization of each stream), spatial types will be inferred from the available spatial annotations in the source. The spatial type system also checks the consistency of these annotations with the given architecture. Spatial consistency means, e.g., that applying a node located on a location to a stream located elsewhere is not correct. As we are in a functional context, spatial types will be inferred for each defined node modularly.

A typed program is then automatically distributed by the compiler, by extracting, for each declared location, one program strictly composed of computations to be performed on this location, as well as added communications from and to other locations in the form of added inputs and outputs.

The spatial type of a stream is the location where this stream is located. In the case of a stream whose values are communicated via a channel from one location to another, its spatial type is a set: it is the set of locations where the stream will be available. The spatial type of a node `f` is written `t_f : (S) → t_o`, where `t_f` and `t_o` are respectively the spatial types of `f`’s inputs and outputs, and `S` is the set of locations involved in the computation of `f`. This set of locations can be larger than the union of `t_f` and `t_o`’s sets of locations, since the computation of `f` can involve intermediary locations.

#### 2.4 Examples

All the examples below assume the architecture declaration:

```
loc A; loc B; link A to B;
```

The first example is a sequence of three nodes `f_1`, `f_2` and `f_3`, each assumed to be of spatial type `∀ δ . c : δ → c` at `δ`. `f_1` and `f_3` are localized by the programmer, respectively on `A` and `B`. `f_2` is not explicitly localized.

```
node g(x) = y3 with
  y1 = f1(x) at A
and y2 = f2(y1)
and y3 = f3(y2) at B
```

This node will be given the spatial type `c : A → (c : B)` at `B`. As the localization of computations has to be done modularly, a spatial type for `f_2` will be given once, among the two possibilities `c : A → (c : B)` at `A` and `c : B → (c : B)` at `B`. In contrast, since there is no communication link from `B` to `A`, the following node will be rejected by the type system:

```
node g'(x) = y3 with
  y1 = f1(x) at B
and y2 = f2(y1)
and y3 = f3(y2) at A
```

Furthermore, it can be noted here that node `g` cannot be used within a located declaration. The following node will be rejected by our type system:

```
node g'(m,x) = y with
  y = g(m,x) at A
```
The second example involves a higher-order node: the node $h$ takes as input two nodes $f$ and $g$, and an input $x$, and applies $f$ to $x$ at one location, and then $g$ to the result of the first application at another location. This example shows also how a node can be defined with local locations for more modularity. These new locations are introduced as a list between $\ldots$, can then be used within the node. This higher-order node uses two location variables $\delta_1$ and $\delta_2$:

\[
\text{node } h \{ \delta_1, \delta_2 \} (f,g,x) = z \text{ with } \\
y = f(x) \text{ at } \delta_1 \\
\text{and } z = g(y) \text{ at } \delta_2
\]

$h$ receives then the spatial type:

\[
\forall \alpha, \beta, \gamma \forall \delta_1, \delta_2 : \{ \delta_1 \triangleright \delta_2 \}.
\]

\[
\begin{align*}
( & \alpha \text{ at } \delta_1 \triangleright \{ \delta_1 \} \rightarrow \beta \text{ at } \delta_1 ) \\
\times ( & \beta \text{ at } \delta_2 \triangleright \{ \delta_2 \} \rightarrow \gamma \text{ at } \delta_2 ) \\
\times ( & \alpha \text{ at } \delta_1 ) \triangleright \{ \delta_1, \delta_2 \} \rightarrow \gamma \text{ at } \delta_2
\end{align*}
\]

The set of constraints $(\{ \delta_1 \triangleright \delta_2 \})$ is inferred from the links required by the node. These constraints are resolved, with the actual architecture, when this node is instantiated. A constraint $\delta \triangleright \delta'$ is resolved, either by stating $\delta = \delta'$, or with two locations $s$ and $s'$ such that there exists a communication link from $s$ to $s'$ in the local architecture. Thus, the node $h$ can be instantiated in these two ways (assuming the existence of two nodes $f$ and $g$, both of spatial type $\forall \alpha, \beta, \gamma \forall \delta_1, \delta_2 : \{ \delta_1 \triangleright \delta_2 \}$):

\[
y_1 = h (f \text{ at } A, g \text{ at } A_x1) \\
y_2 = h (f \text{ at } A, g \text{ at } B_x2)
\]

We can observe than an arrow type appearing on the left of another arrow type cannot comprise more than one location. This is caused by the form taken by the distribution: since the projection operation on one node is performed on locations, and not sets of locations, we cannot handle effect variables, unlike other type and effect systems.

2.5 Application

As a concrete example, we consider the definition of a reception channel of a software radio. A software radio, or software-defined radio, is a radio in which components usually defined as hardware, e.g., demodulation or filter components, are defined as software [15]. This allows in particular the reconfiguration, possibly dynamically, of these components.

Consider a reception channel composed of three main components: a pass-band filter allowing the selection of the carrier wave, a demodulator component, and a component allowing the analysis of the received signal, e.g., an error-correction function. For the sake of performance, these components are usually implemented on different architectural elements: the pass-band filter on a FPGA, the demodulator on a digital signal processor (DSP), and the error correction on a general-purpose processor (GPP).

Each component of this reception channel could easily be defined separately. But in the case of software-defined radio, the system must support several functionalities [10]; each of these functionalities must be engineered separately, and then integrated together. Then, there is a conflict between distribution and functional modularity issues.

Let us study the case of a multichannel reception system that supports the two mobile standards GSM and UMTS: the former involves a filter for 1800 MHz frequencies, a GMSK demodulator, and a CRC / convolutional error correction module, while the latter involves a 2 GHz filter, a QPSK demodulator, and a CRC / convolutional / turbo codes error correction module.

Figure 1 shows an implementation of this reception channel on a system composed of three hardware components: a FPGA dedicated to the execution of the two pass-band filters, a DSP for the demodulation functions, and a GPP for error-correction modules and for the control of the whole system, i.e., in this case, the switch between the two channels. This system has one input $x$, the radio signal from the antenna, $y$ denotes the output signal of the system, i.e., the decoded and corrected information received by the channel. From this value, a function gsm_or_umts (noted $g$ on Figure 1) for the sake of brevity), local at GPP, computes what channel will be used at next instant. In this figure, each location is graphically represented by a gray box.

In a classical context, designing this multichannel software radio would be performed by separately programming each of the three hardware components, which raises two problems. Firstly, there is no guarantee that the components interact as specified: i.e., the 1800 filter with the GMSK demodulator, and so on. This requires the MUX function to be duplicated on the three computing resources, so as to guarantee the correctness of the system. This situation compromises the modularity of the system. Secondly, each of the two channels corresponds to an independent software entity. Programming independently each hardware component leads to the separate design, at least from some point of the design flow, of closely related software components (e.g., filter and demodulator of the same channel).

For the sake of modularity, this system would be better designed by considering the channels independently, and not the hardware components. This situation suggests adding primitives allowing to express the localization of streams directly in the language. Such primitives should allow the programming of software components independently of the architecture, handled as a separate concern. Thus, consistency analysis such as data typing could be performed on the global program: communication channels could be typed and the data consistency of the whole system could be checked. This way, inconsistencies due to serialisation could be detected at compilation time.

The code below implements this multichannel software radio, with our extended language. The architecture consists of three locations, FPGA, DSP, and GPP, completely connected:

\[
\text{node channel(filter,demod,crc,x) = y with } \\
f = \text{filter}(x) \text{ at FPGA} \\
d = \text{demod}(f) \text{ at DSP} \\
y = \text{crc}(d) \text{ at GPP}
\]

\[
\text{node multichannel\_sdr(x) = y with } \\
c = \text{gsm\_or\_umts(y) at GPP in } \\
\text{if (true fby c) then } \\
y = \text{channel(filter\_1800,GMSK,conv,x) } \\
\text{else } \\
y = \text{channel(filter\_2000,QPSK,turbo,x)}
\]

This implementation strictly follows the architecture of the system described in Figure 1. It shows the declaration of three symbolic locations (FPGA, DSP, and GPP). We assume that all filter, demodulation, and correction functions are local ones, i.e., they are of
spatial type $\forall \delta. c$ at $\delta \rightarrow \{\delta\} \rightarrow c$ at $\delta$. Since the conditional construct comprises declarations that have to be executed on the set of locations $\{\text{FPGA, DSP, GPP}\}$, $c$ is thus inferred to be communicated to these locations. The conditional if/then/else is evaluated with the value of $c$ at the previous instant. The distribution of this example will put a copy of this if/then/else on these three locations. Finally, the expression $\text{true}_s$ will be computed at GPP, since the result of this expression has to be communicated to the three locations where the conditional construct will be duplicated.

By the same reasoning, we can infer that the spatial type of $x$ is FPGA, and the one of $y$ is GPP. As a result, the spatial type of the node multichannel_sdr is:

$$(c \text{ at FPGA}) \rightarrow ((\text{FPGA, DSP, GPP})) \rightarrow (c \text{ at GPP})$$

### 3. Formalization

We first define a synchronous dataflow core language (Section 3.1), and give its centralized semantics (Section 3.2) and its distributed semantics (Section 3.3). The centralized semantics is considered to be a reference semantics and we only consider programs that react w.r.t. this semantics. Programs that do not react (e.g., for typing or causality reasons) are assumed to be rejected by other means [9]. The distributed semantics allows us to give a meaning to location annotations. A spatial type system is then presented (Section 3.4). It is used to both reject programs which cannot be distributed and to annotate every expression from the source code with explicit locations.

#### 3.1 The Core Language Syntax

$$P ::= A; d; D$$

$$A ::= A; A \mid \text{loc } A \mid \text{link } A \to A$$

$$s ::= \delta \mid A$$

$$d ::= \text{node } f[\delta_1, \ldots, \delta_n](p) = e \text{ with } D \mid d; d$$

$$D ::= p = e \mid p = x(e) \mid D \mid \text{if } e \text{ then } D \text{ else } D$$

$$p ::= p, p, x$$

$$e ::= i \mid x \mid (e, e) \mid \text{op}(e, e) \mid e \text{ fby } e \mid e \text{ at } s$$

A program is made of an architecture description $(A)$, a sequence of node definitions $(d)$ and a main set of equations $(D)$. An architecture description is a set of declarations of locations (loc $A$) or links (link $A \to A$) which state the existence of a communication link from one location to another. A location $s$ is either a location variable $\delta$, or a location constant $A$. A node definition is composed of an expression and a set of equations. A set of local locations $\{\delta_1, \ldots, \delta_n\}$ can be associated as location parameters of a node definition (node $f[\delta_1, \ldots, \delta_n](p) = e$ with $D$). Definitions $D$ define patterns of variables $p$, and are either single equations ($p = e$), definitions naming the result of an application ($p = x(e)$), parallel declarations ($D$ and $D$), or alternative declarations (if $e$ then $D$ else $D$). An expression $e$ may be an immediate value ($i$), a variable ($x$), a pair construction ($e, e$; pair destruction ($e, e$)), and are either single expressions ($e$), or expression annotated with an explicit location $s$ ($e$ at $s$).

#### 3.2 The Centralized Synchronous Semantics

The purpose of the centralized semantics is to serve as a reference semantics. This semantics does not take into account distribution primitives. We first introduce auxiliary definitions. A value is either an immediate constant ($i$), a pair or a function.

$$v ::= i \mid (v, v) \mid \lambda x. e \text{ with } D$$

$$R ::= [v_1/x_1, \ldots, v_n/x_n] \text{ s.t. } \forall i \neq j, x_i \neq x_j$$

$$S ::= R_1, R_2, \ldots$$

A reaction environment $R$ associates values to names and assumes that names are pairwise distinct. $S$ denotes a sequence of reaction environments.

Given a sequence $d$ of node definitions node $f[\delta_1](x_i) = e_i$ with $D$, an initial global environment $R_d$ is defined, holding $\lambda$-values of each $f_i$. This initial environment will be given as input of the main program.

$$R_d = [\lambda x_1.e_1 \text{ with } D_1/f_1, \ldots, \lambda x_n.e_n \text{ with } D_n/f_n]$$

The synchronized centralized semantics is defined by means of two reaction predicates. $R \vdash e_1 \xrightarrow{\beta} e_2$ states that in the reaction environment $R$, the expression $e_1$ emits the value $v$ and rewrites into the new expression $e_2$. The predicate $R \vdash D_1 \xrightarrow{R'} D_2$ states that in the reaction environment $R$, the declaration $D_1$ defines the reaction environment $R'$ and rewrites into $D_2$. The centralized execution of a program $P$ is denoted $S_1 \vdash P : S_0$, meaning that under a sequence of input environments $S_1 = S_i \ldots$, the program $P = A;d;D$ produces a sequence of output environments $S_0 = S_1;S_2;\ldots$ such that:

$$R_{d_1},R_{d_2},R_0 \vdash D_0 \xrightarrow{R_0} D' \quad S_1;A;d;D' : S_0$$

The rules for the reaction predicates are given in Figure 2

![Figure 2. Centralized synchronous semantics.](image-url)

An immediate value emits itself and rewrites to itself (rule IMM). A variable emits its current value as it is present in the reaction environment (rule INST). An initialized delay $e_1 \text{ fby } e_2$ emits the first value of $e_1$, then the previous value of $e_2$ (rule FBY). An
operation is performed pointwisely on immediate values (rule $\text{Op}$). Pair construction follow classical rules (rule $\text{Pair}$). Locations are not taken into account here (rule $\text{At}$): annotations added by the programmer do not alter the centralized semantics of the program (i.e., its functionality). An equation $p = e$ emits the reaction environment defining the variables contained in $p$ (rule $\text{Def}$). A sequential function application is replaced by its body and argument definition (rule $\text{App}$). Parallel equations are mutually recursive (rule $\text{And}$). A conditional statement executes its first branch if its condition is true (rule $\text{If-1}$) and its second branch otherwise (rule $\text{If-2}$).

### 3.3 The Distributed Synchronous Semantics

The distributed semantics also operates on a program $\mathcal{P} = A_1; d_1; D_1$, but takes into account the architecture description and the explicit locations. However, it remains a synchronous semantics in the sense that the desynchronization due to the insertion of communications is not accounted for. It defines a 

spatialized execution:

the values $\tilde{v}$ emitted by expressions are now 

distributed values, i.e., they are annotated with location information stating how these values are distributed on the architecture:

$\tilde{v} := lv \text{ at } A \mid (\tilde{v}, \tilde{v}) \mid A \delta L A x \text{ with } D$

$lv := i \mid (lv, lv)$

$R := [v_1, x_1, \ldots, v_n/x_n] \text{ s.t. } \forall i \neq j, x_i \neq x_j$

$S := R_1, R_2, \ldots$

$G := \{S, L\}$

A distributed value $\tilde{v}$ is either a local value $lv$, localized on the site $A (lv \text{ at } A)$, a distributed pair $(\tilde{v}, \tilde{v})$, or a node. A sequence of location parameters $\bar{s}$ is associated to nodes. A local value is either an immediate value $i$, or a local pair $(lv, lv)$. $R$ denotes a distributed reaction environment, and $S$ a sequence of distributed environments. $G$ denotes an architectural graphs composed of a set of locations $S$, and a set of communication links $L \subseteq S \times S$.

Several values can represent the same distributed value:

$\langle lv_1, lv_2 \rangle \text{ at } A = \langle lv_1 \text{ at } A, lv_2 \text{ at } A \rangle$

$\tilde{v}_1 = \tilde{v}_1', \tilde{v}_2 = \tilde{v}_2'$

$(\tilde{v}_1, \tilde{v}_2) = (\tilde{v}_1', \tilde{v}_2')$

These equalities mean that a local pair $(1, 2)$, localized on the site $A$, can be equivalently denoted by the distributed values $(1, 2) \text{ at } A$ or $(1 \text{ at } A, 2 \text{ at } A)$. A distributed pair can be, for example, the pair $(1 \text{ at } A, 2 \text{ at } D)$: the first compound is located on $A$, and the second on $D$.

The operator $\text{loc}(\cdot)$ gathers the set of locations from a distributed value:

$\text{loc}(i \text{ at } s) = \{s\}$

$\text{loc}((\tilde{v}_1, \tilde{v}_2) \text{ at } s) = \text{loc}(\tilde{v}_1) \cup \text{loc}(\tilde{v}_2)$

The operator $\mid \cdot \mid$ erases annotations from a distributed value to get a centralized value, and extends straightforwardly to reaction environments:

$\mid i \text{ at } s \mid = i$

$\mid (\tilde{v}_1, \tilde{v}_2) \text{ at } s \mid = (\tilde{v}_1)\mid, (\tilde{v}_2)\mid$

The distributed semantics is defined by means of two predicates refined from their centralized versions. $\hat{R} \stackrel{\ell}{\rightarrow} e_1 \overrightarrow{e_2}$ states that in the distributed reaction environment $\hat{R}$, the expression $e_1$ emits the distributed value $\tilde{v}$ and rewrites into $e_2$. $\ell$ represents the set of locations involved in the computation of $\tilde{v}$. The predicate for declarations is defined as well.

We denote by $\mathcal{S}$ the set of declared constant locations, and by $\mathcal{L} \subseteq \mathcal{S} \times \mathcal{S}$ the set of declared communication links. The relation $\mathcal{C}$ defines the ability to communicate, and not the actual existence of communication channels, which will be inferred by the refined version of the type system. $G$ denotes an architecture graph, composed of a set of locations $\mathcal{S}$, and a set of links $\mathcal{L}$ between these locations. An architecture description $A$ defines an architecture graph $G$; the notation $G \vdash A : G'$ means that given the architecture graph $G$, $A$ defines the new architecture graph $G'$. The rules $\text{Arch}$, $\text{Def-loc}$ and $\text{Def-link}$ define this predicate:

$$
\text{Arch} \quad \langle S, L \rangle \vdash A : \langle S_1, L_1 \rangle \quad \langle S_1, L_1 \rangle \vdash A : \langle S_2, L_2 \rangle \\
\quad \langle S_2, L_2 \rangle \vdash A : \langle S, L \rangle
$$

$$
\text{Def-loc} \quad \langle S, L \rangle \vdash \text{loc} A : \langle S, L \cup \{A\}, L \rangle
$$

$$
\text{Def-link} \quad A_1, A_2 \in S \quad \langle S, L \rangle \vdash \text{link} A_1 \text{ to } A_2 : \langle S, L \cup \{A_1 \rightarrow A_2\} \rangle
$$

For clarity, we assume that the architecture graph $G$ is global for subsequent semantic rules. The annotated execution of a program $\mathcal{P} = S_1 \vdash P : S_0$, meaning that under a sequence of input environments $S_1 = R_1, R_2, \ldots$, the program $\mathcal{P} = A_1; d_1; D_1$ produces a sequence of output environments $S_0 = R'_1, R'_2, \ldots$.

$$
\langle \emptyset, \emptyset \rangle \vdash A : G \quad \hat{R}_d, \hat{R}_i, \hat{R}_o \stackrel{\ell}{\rightarrow} D \quad \hat{R}_d \vdash \hat{R}'_d \quad \hat{R}_i \vdash \hat{R}'_i \quad \hat{R}_o \vdash \hat{R}'_o
$$

where $\hat{R}_d$ is defined from the sequence of node definitions $d = n_1 l_1(x_1) = e_1$ with $D_d$ as:

$$
\hat{R}_d = [A_1, \lambda x_1.e_1 \text{ with } D_1/f_1, \ldots, A_n, \lambda x_n.e_n \text{ with } D_n/f_n]
$$

The rules for the predicates $\hat{R} \vdash e_1 \overrightarrow{e_2}$ and $\hat{R} \vdash \hat{R}_d \overrightarrow{\hat{R}'_d}$ are given in Figure 1. An immediate value can be emitted anywhere (rule $\text{Imm}$). Rule $\text{Inst}$ defines the instantiation. A distributed value can be communicated from location $s$ to location $s'$ if there exists a communication link from $s$ to $s'$ (rule $\text{Comm}$). A binary operation can be performed only on immediate values located on the same location $A$; the result is located on $A$ as well (rule $\text{Op}$). An annotated expression must have at least the location stated for its computation (rule $\text{At}$). An application involves choosing a set of constant locations, and replacing location parameters by these locations in the expression and the declaration (rule $\text{App}$). The other rules state that the computation of a statement involves the union of the locations involved for the computation of its compounds.

Lemma 1 states that if a program reacts with the distributed semantics, then it reacts with the centralized one and produces the same values.

**Lemma 1.** For all $D, D', R, R'$, if $R \vdash D \overrightarrow{D'}$, then there exists $R, R'$ such that $R = [\hat{R}], R' = [\hat{R}']$ and $R \vdash \overrightarrow{D \overrightarrow{D'}}$.

### 3.4 Spatial Types

For the sake of clarity, we first present a simplified version of the type system. For projection, we refine this first version to take communication channels into account (Section 1).

The syntax of spatial type expressions is:

$$
\begin{align*}
\sigma &::= \forall x_1, \ldots, x_n, \forall \delta_1, \ldots, \delta_n : C.t \\
t &::= t \cdot (t) \cdot t \mid t \times t \mid tc \text{ at } s \\
t c &::= c \mid \alpha \mid tc \rightarrow tc \mid tc \times tc \\
\ell &::= \{s_1, \ldots, s_n\} \\
s &::= \emptyset A \\
\mathcal{H} &::= \mathcal{H} \at A \mid x_1 : \sigma_1, \ldots, x_n : \sigma_n \\
\mathcal{C} &::= \{s_1 \triangleright s_1, \ldots, s_n \triangleright s_n\}
\end{align*}
$$

$\mathcal{H}$ is the spatial typing environments. $\mathcal{H} \at A$ denotes a located environment, i.e., a typing environment from which every spatial type will be forced to represent a value entirely located on $A$. 
We distinguish spatial type schemes ($\sigma$), which can be quantified, from simple spatial types ($t$). A set of constraints $C$ can be associated to quantification of location variables ($\forall \theta_1, \ldots, \theta_n : C.t$). We note $\forall \theta_1, \ldots, \theta_n t$ the scheme $\forall \theta_1, \ldots, \theta_n : \emptyset t$. A spatial type can be either a node type ($t \leftarrow t \rightarrow t$), a pair type ($t \times t$), or a located type ($t \text{ at } s$). A located type can be either a stream type ($\mathbb{C}$, such as boolean, integer, etc.), a type variable ($\alpha$), a local function ($t \rightarrow t$), or a local pair type ($t \times t$). $\ell$ denotes sets of locations. A location is either a location variable $\delta$, or a location $A$. $C$ is a set of constraints between locations. A constraint $s_1 \triangleright s_2$ means that either $s_1 = s_2$, or there exists a communication link from $s_1$ to $s_2$. Conversely, a declaration of communications links $\mathcal{L}$ leads to the set of constraints $\text{constr}(\mathcal{L}) = \{ s \triangleright s' \mid (s, s') \in \mathcal{L} \}$.

A value of spatial type $t \text{ at } s$ is a value located on $s$. A value of spatial type $t \leftarrow t \rightarrow t$ is a node whose input is of spatial type $t_1$, whose output is of spatial type $t_2$, and whose computation involves the set of locations $\ell$. The following equalities stand:

\[(t_1 \times t_2) \text{ at } s = (t_1 \text{ at } s) \times (t_2 \text{ at } s)\]
\[(t_1 \leftarrow t_2) \text{ at } s = (t_1 \text{ at } s) \leftarrow (t_2 \text{ at } s)\]
\[t_1 \leftarrow t_2 = t_1' \leftarrow t_2'\]
\[(t_1 \times (t_2 \leftarrow t_3)) = (t_1 \times t_2) \leftarrow t_3\]

The instantiation mechanism ensures the localization of a type instantiated from a located environment:

\[(t[\gamma] \text{ at } s, C) \leq (t \text{ at } s, C) \leq H(t)\]

We note respectively FLV($t$) and FTV($t$) the set of free location variables and free type variables of the type $t$. FLV and FTV are straightforwardly extended to typing environments.

A set of constraints $C$ is compatible with a set of communication links $\mathcal{L}$, noted $\mathcal{L} \models C$, iff $s \triangleright s' \in C \land s \not\equiv s' \Rightarrow (s, s') \in \mathcal{L}$.

Before presenting our spatial type system, we introduce the following notations:

- For a program $P$, the notation $\vdash P : t$ means that the program $P$ is of spatial type $t$.
- For declarations (resp. expressions), the notation $H[G \vdash D : H'/\ell]$ (resp. $H[G \vdash e : t/\ell]$) means that, in the spatial type environment $H$ and the architecture graph $G$, the declaration $D$ (resp. the expression $e$) defines a new environment $H'$ (resp. of spatial type $t$), and its computation involves the set of locations $\ell$.

The function locations($\cdot$) gives the set of locations involved in the spatial type given as argument. It is defined as:

- locations($t_1 \times t_2$) = locations($t_1$) $\cup$ locations($t_2$)
- locations($t_1 \leftarrow t_2$) = $\ell$
- locations($t \text{ at } s$) = $\{s\}$

The top-level declaration of a program is typed from the initial environment $H_0$, defined as:

$\ell \vdash H_0 = \{ \ldots \}$

Our spatial type system is formally defined by the inference rules shown in Figure 3. Typing a program involves building an architecture graph from the architecture description, and then using it to type the nodes and the main declarations (rule R PROG).

An immediate value can be used on any location (rule IMM). Type schemes can be instantiated (rule INST). Typing a pair involves stating that this pair has to be evaluated on the union of the sets of locations on which each member of the pair has to be evaluated (rule PAIR). Typing a located expression ($e \text{ at } A$) involves building a located typing environment (rule AT). Communications are expressed as subtyping (rule COMM).

The spatial type of a node consists of the spatial types of its inputs, the computed expression, and the set of locations involved in this computation (rule NODE). The type of a node is generalized w.r.t. the set of locations and links introduced by this architecture.

Typing an expression $x \in e$ involves building a singleton typing environment (rule DEF). Rule APP states that an application must be evaluated on the union of the set of locations where the node $f$ and its argument $e$ must be evaluated, and the set of locations $\ell_f$ involved in the computation of the node $f$. Parallel declarations involve, for their computations, the union of the sets of and of locations involved in the computation of their compounds (rule AND).

Finally, typing an if/then/else declaration involves locating the condition expression on a location $s$, and adding constraints that every location involved in declarations $D_1$ and $D_2$ must be accessible from $s$ (rule IF).
4. Distribution

4.1 Principle

Once programs have been typed, every expression is annotated with a location that specifies where it has to be computed. Communications are inserted when a value is produced at a location and used at another. From this typed program, the compiler produces several new programs — one for every location s — erasing the code that is not necessary at this location. The run-time we have chosen is a classical one for globally asynchronous locally synchronous (GALS) systems: communications are done through FIFOs.

We show below the result of the projection of the node f of Section 2.2 on A and B, noted respectively f_A and f_B. The distribution of this node will involve adding a communication between these two computations. This communication will take the form of an additional output on f_A (named here c_y, holding the value y computed on A and used on B), together with an additional input on f_B. Original inputs and outputs are not suppressed: \( \llcorner \) denotes an irrelevant value which will not be used on the current location. It is used here to replace the output z, whose computation is suppressed at A.

\[
\text{node } f_A(x) = (\llcorner, c_y) \text{ with } c_y = g(x)
\]

\[
\text{node } f_B(x, c_y) = z \text{ with } z = h(c_y)
\]

The semantically equivalent distributed system is then obtained by connecting the input and output c_y, holding the communicated value y. The program below shows the distributed execution, using a FIFO materialized by send/receive primitives, of the result of the projection of the program \( y = f(x) \).

\[
\begin{align*}
(y_A, c_y) &= f_A(x); \\
(y_B) &= f_B(\llcorner, c_y)
\end{align*}
\]

4.2 Example

The result of the projection of the two nodes of the Section 2.2 on the location DSP is given on Figure 5. The projection of the channel node shows that the node applications of filter and crc have been removed, and that a new input c1 (holding the value of f) and a new output c2 (holding the value of d) have been added. This implies the addition, on the projection of the multichannel_sdr node, of two new inputs (c2 and c3) and two new outputs (c4 and c5), one for each channel instance. The new input c1 of the projected multichannel_sdr node holds the value c.

\[
\begin{align*}
\text{node } \text{channel(filter, demod, crc, x, c1)} &= (\llcorner, c_2) \text{ with } \\
&\quad d = \text{demod}(c1) \\
&\quad \text{and } c2 = d
\end{align*}
\]

\[
\begin{align*}
\text{node } \text{multichannel_sdr(x, s, c1, c2, c3)} &= (\llcorner, c_4, c_5) \text{ if } c1 \text{ then } \\
&\quad (y, c_4) = \text{channel}(\llcorner, gmsk, \llcorner, \llcorner, c_2) \\
&\quad \text{else } \\
&\quad (y, c_5) = \text{channel}(\llcorner, qpsk, \llcorner, \llcorner, c_3)
\end{align*}
\]
node definition to node instances. \( T \) denotes sequences of channels. The concatenation of two sequences of channels, noted \( T_1, T_2 \), is defined iff channel names in \( T_1 \) and \( T_2 \) are disjoint. We denote by \( e \) the empty sequence.

We note \( \text{dom}(T) \) the set of channel names of \( T \). \( T', T'' \cong T \) means that the sequences \( T \) and \( T'' \) are equal, modulo channel renaming. This renaming allows the multiple instantiations of node comprising communications.

\( T' \cong T \iff T'[c_1/c_1, \ldots, c_n/c_n] \) where \( \{c_1, \ldots, c_n\} = \text{dom}(T) \).

The projection of a declaration \( \mathcal{D} \) on a location \( A \) is noted \( \mathcal{H} | \mathcal{D} | e : t/\ell/T \quad \Rightarrow \quad \mathcal{D}' \), and results in a new declaration \( \mathcal{D}' \), containing only the computations to be performed on \( A \). The projection of an expression \( e \), of spatial type \( t \), on a location \( A \), is noted \( \mathcal{H} | e : t/\ell/T \quad \Rightarrow \quad e'/\mathcal{D} \), and results in a new expression \( e' \), as well as a declaration \( \mathcal{D} \), containing channels outputs to be defined.

A channel named \( c \) in an environment channel will be introduced as the variable \( c \) as input or output of the target program, \( c \) assumed to be of different name space than other variables of the program.

We denote by \( e \) the empty declaration, and by \( \bot \) a value which will never be used (i.e., a void). For any declaration \( \mathcal{D}, \mathcal{D}' \) and \( e = e \) and \( \mathcal{D} = \mathcal{D}' \). For any expression \( e \), we have \( (e \bot) = (e \bot) = \bot \).

Also, we note \( T \uparrow A \) (resp. \( T \downarrow A \)) the set of channels with origin (resp. destination) \( A \):

\[
\begin{align*}
    &\begin{cases}
        0 \uparrow A = \emptyset \\
        \left(\left[ A_1 \overset{c_1}{\leftrightarrow} A_2, T\right]\right) \uparrow A = \left[ A_1 \overset{c_1}{\leftrightarrow} A_2, (T \uparrow A) \right] \quad \text{if} \ A_1 = A \\
        \left(\left[ A_1 \overset{c_1}{\leftrightarrow} A_2, T\right]\right) \downarrow A = \left[ A_1 \overset{c_1}{\leftrightarrow} A_2, (T \downarrow A) \right] \quad \text{else},
    
        0 \downarrow A = \emptyset \\
        \left(\left[ A_1 \overset{c_1}{\leftrightarrow} A_2, T\right]\right) \uparrow A = \left[ A_1 \overset{c_1}{\leftrightarrow} A_2, (T \uparrow A) \right] \quad \text{if} \ A_2 = A \\
    \end{cases}
\end{align*}
\]

The projection rules are given in Figures 6 and 7. Channels are used at communication points. If an expression \( e \) is sent from \( A \) to \( A' \) through the channel \( A \leftrightarrow A' \), then:

- for the projection on \( A \), the communication involves sending a value: the resulting expression is void, and we add the definition of the channel \( c \) as the result of the projection of \( e \) on \( A \) (rule \( \text{COMM-P-FROM} \));
- for the projection on \( A' \), the communication involves receiving a value: the resulting expression is the channel holding this value (rule \( \text{COMM-P-To} \)).

Finally, if \( A \) does not appear in the set of locations involved in its computation, then the expression can be suppressed on \( A \) (rule \( \text{SUPPR-P} \)). Projections of a pair consist in the projection of its compounds (rule \( \text{PAIR-P} \)).

The projection of a located declaration and parallel declarations involves the projection of its compound (rules \( \text{AT-P} \) and \( \text{AND-P} \)). The projections of applications and node definitions involve adding to the inputs and outputs of the node, the channels used by this node (rules \( \text{APP-P} \) and \( \text{NODE-P} \)). Nodes with local architecture are assumed to be inlined. The relevance of the name order appears here, as the order of the added inputs and outputs must be consistent with every instances of these nodes, and for every projection.

Projection of a conditional is divided in two rules: one for the projection on a location where the conditional expression is computed: this first rule shows the definition of every channel needed to send this value to other locations where the conditional will be evaluated (rule \( \text{IF-P-FROM} \)); and one for the projection on a location where the conditional expression has to be received: this expression is then replaced by the name of the channel holding this value (rule \( \text{IF-P-To} \)).

\[
\begin{align*}
    &\begin{cases}
        \mathcal{D} \quad \Rightarrow \quad \mathcal{D}' \quad \text{if} \ A \neq A' \\
        \mathcal{D} \quad \Rightarrow \quad \mathcal{D}' \quad \text{if} \ A \neq A' \quad \text{and} \ A \iff A' v \quad \Rightarrow \quad \mathcal{D}' v \quad \text{if} \ A \neq A' v \quad \Rightarrow \quad \mathcal{D}' v
    \end{cases}
\end{align*}
\]

We can then relate two reaction environments \( R \) and \( R' \) w.r.t. a typing environment \( \mathcal{H} \):

\[
R \equiv_H R' \quad \text{iff} \quad \forall x \in \text{dom}(R), \forall A \in \mathcal{S}, R(A) = \text{dom} R(\mathcal{H}(x)) \quad R'(x_A)
\]

Figure 6. Rules for the projection operation (I).

The global meaning of a distributed program is then defined by the parallelization of its projected declarations.

We note \( \mathcal{S} = \{ A_1, \ldots, A_n \} \) the set of defined constant locations where the source declarations are projected. The global meaning of a declaration \( \mathcal{D} \) projected on the locations \( \mathcal{S} \), is defined by:

\[
\begin{align*}
    &\mathcal{T} \quad \Rightarrow \quad \mathcal{T}' \\
    &\mathcal{T}_1 \quad \Rightarrow \quad \mathcal{T}_1' \quad \text{and} \quad \mathcal{T}_2 \quad \Rightarrow \quad \mathcal{T}_2' \\
    &\mathcal{D}_1 \quad \Rightarrow \quad \mathcal{D}_1' \quad \text{and} \quad \mathcal{D}_2 \quad \Rightarrow \quad \mathcal{D}_2'
\end{align*}
\]

In order to relate a target program with its source, we define a relation on values, denoted \( \leq \), such that \( v' \leq v \) means that the value \( v' \), emitted from an expression of type \( t \), represents the value \( v \) at the location \( A \). We have:

\[
\begin{align*}
    &\begin{cases}
        v' = v \\
        v' \leq v \quad \text{at} \ A v \quad \Rightarrow \quad A \neq A' \quad v' \leq v \quad \text{at} \ A' v \quad \Rightarrow \quad v' \leq v \quad \text{at} \ A' v
    \end{cases}
\end{align*}
\]
5. Discussion

5.1 Implementation

The spatial type system presented in Section 5.4 relies on a subtyping mechanism. It corresponds to the case where communications can occur anywhere in the code. This situation raises two problems. Firstly, the implementation of type systems with subtyping mechanism is costly: usual algorithms rely on the systematic application of the subtyping rule. Secondly, this choice leads to a situation where the programmer has no control over where the communications can occur. These problems, though orthogonal, can be addressed together: giving some control to the programmer means restricting the points where subtyping can be applied. We refine our type system in order to address these two problems.

We restrain communicated values to be variables introduced by equations \((x = e)\). Thus, in the program of Section 5.4 only \(y\) and \(z\) can be communicated from one location to another. Then, we can use a generalization mechanism to infer communication constraints, instead of inferring them by subtyping. This corresponds to restricting the subtyping mechanism to instantiation points.

This refined type system, as well as the projection operation presented in Section 5.4 have been implemented in the Lucid Synchrone compiler [1]. The independence of the method w.r.t. other analysis allowed this implementation to be quite modular, implying very few modifications within the rest of the compiler. The implementation consists to generate distribution constraints, which are resolved modularly for each node. We implemented a naive resolution algorithm, which can easily be replaced by any existing distribution algorithm, taking into account efficiency issues (for example, the AAA method of SynDex [18]). This implementation has been tested on the software-defined radio example, which is in its actual size composed of 150 lines of code, spread out into 25 Lucid Synchrone nodes. We have tested our automatic distribution algorithm on a benchmark suite composed of programs of few hundred lines from the current Lucid Synchrone release. This point is a good indication of the flexibility of the extension proposed, allowing the distribution of programs not primarily designed with distribution in mind. The scalability of this method is a direct outcome of the use of a type system aiming at modular distribution; it has been also applied on an ad hoc example composed of 10 Lucid Synchrone nodes, comprising each 60 equations.

5.2 Related Work

Various solutions have emerged in order to use synchronous languages for the design of distributed systems. Some of them operate on a compiled model of the program, by “coloring” atomic instructions with localization information, inferred from input and output locations [8]. The whole program is first compiled into an intermediate sequential format, on which the distribution is then applied. This format consists of a sequence of atomic instructions, representing one computation step of new values carried on each stream. Then, the distribution involves placing each instruction onto one or several locations, taking into account the consistency of the control flow on each location. Another approach is to directly annotate the source program with locations, so as to define the localization of each variable of the program: the distribution is then performed with regard to these annotations. This approach has been applied to Signal [3] as well as Lustre [7]. In both cases, the soundness of the distribution algorithm has been proved [6, 14], meaning that the combined behavior of the distributed fragments has the same functional and temporal semantics as the initial centralized program.

The originality of our method resides in the fact that we use a spatial type system to check the consistency of the distribution specifications inserted by the programmer, and that we perform modular distribution, allowing the expression of higher-order features, applied for instance to dynamic reconfiguration of nodes by application of other nodes as inputs. Since a Kahn semantics [11] can be given to our language, the semantic equivalence between the source program and the synchronous product of the fragments resulting from the projection is sufficient to describe an asynchronous distribution. While the language presented has higher-order features, this method can easily be applied to other language with comparable semantics such as Lustre. In contrast, more general frameworks such as Signal cannot be addressed here for this reason.

Several approaches have been considered to solve the problem of data consistency of distributed programs. A translation operation is presented in [16], as well as an effect type system, in order to automatically obtain a multi-tier application from an annotated source program. Our proposition differs by the fact that our type system is not only a specification, but also consists in what the authors called “location analysis”, thus allowing us to perform this analysis in a modular way, and on a program comprising higher-order features. This last approach, as well as our projection operation, can be compared with slicing methods [20], as they consists in extracting specific parts of a global program. Type systems have been used to ensure memory consistency [19], or for pointer analysis within a distributed architecture [12]. The ACUTE language [17] is an extension of OCAML with typed marshalling. Communication channels between two ACUTE programs can also be considered as typed, as the type of marshalled and unmarshalled data are dynamically verified, at execution time. The consistency considered is between separately-built programs, whereas our approach is to consider the programming of a distributed system as one global program, allowing global static verifications. Our approach can be compared with automatic partitioning: the J-Orchestra system [13] allows the user to assign network sites to classes of Java programs. Then, this automatic partitioning system transform the initial program into a distributed one, taking into account distant or direct references. This last approach differs with our by the fact that we integrate the distribution constructs within our language, which allows relying distribution on semantic basis. Finally, OZ and its distributed extension [4] proposes a way to separate the functionality of a distributed application and its distribution structure, by allowing the programmer to give a different distributed semantics to every different object of a program. These two languages aims at loosely coupled distributed systems without architecture constraints, whereas our approach concerns strongly coupled architecture, and we aim to ensure the consistency of the distribution w.r.t. one architecture, given the communication constraints.

5.3 Conclusion and Future Work

We have proposed a spatial type system to solve the problem of automatic distribution of dataflow programs. It is based on a core dataflow language, which we have extended with distribution primitives to allow the programmer to specify, on one hand his/her target distributed architecture, and on the other hand where some nodes and/or variables are to be located. The underlying philosophy is
the functional distribution, meaning that some functionalities of the program must be computed at some precise location because they require some specific sensors, actuators, and/or computing resources that are available only at this location. In this context, we use type inference to decide at which location each node must be computed, and at which nodes the program communication primitives must be inserted. The compilation of a correctly spatially typed program produces one program for each computing location specified by the programmer. We use abstract communication channels to exchange a value between two locations and to synchronize them. Compiling each program and linking with a dedicated library implementing those communication channels then gives one binary code for each location. The refined version of the type system, as well as the operation projection, has been implemented in a synchronous dataflow compiler [1].

Future work mainly involves allowing the description of more complex architectures, with hierarchical locations or communication masking (e.g., MPSoCs): our proposal of architecture constraints is not sufficient to catch the complexity of actual architecture of distributed embedded systems. Yet, our current constraints show the interest of a type system for checking the consistency of a distributed program w.r.t. such constraints.

References