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

Gwenaël Delaval, Alain Girault, Marc Pouzet

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Gwenaël Delaval — Alain Girault — Marc Pouzet

N° 6378

December 3, 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6378--FR+ENG

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

Gwenaël Delaval [*] , Alain Girault [†] , Marc Pouzet [‡]

Thème COM — Systèmes communicants
Projets POP ART

Rapport de recherche n° 6378 — December 3, 2007 — 46 pages

Abstract: This paper addresses the design of distributed systems with synchronous dataflow languages. As modular design entails handling both architecture and functional modularity, we propose a language-oriented solution, involving the extension of a synchronous dataflow language with primitives for program distribution. These primitives allow the programmer to describe the architecture of the system and to express where streams and expressions are located in this architecture. A distributed semantics is first proposed as formalization of the distributed execution of programs. A type and effect system is then provided 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. A type-directed projection operation allows us 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 implemented in the Lucid Synchrone compiler.

Key-words: Synchronous programming, automatic distribution, type systems

[*] Gwenael.Delaval@inrialpes.fr, http://pop-art.inrialpes.fr/~delaval
[†] Alain.Girault@inrialpes.fr, http://pop-art.inrialpes.fr/~girault
[‡] LRI, Université Paris-Sud. Marc.Pouzet@lri.fr, http://www.lri.fr/~pouzet

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Un système de types pour la répartition automatique de programmes flot de données synchrones d'ordre supérieur

Résumé : Ce rapport traite de la conception de systèmes répartis au moyen de langages flot de données synchrones. La conception modulaire de tels systèmes comprend deux préoccupations différentes, la modularité fonctionnelle et la modularité d'architecture. Ces deux préoccupations pouvant s'avérer orthogonales, nous proposons une solution intégrée dans un langage de programmation, par son extension avec des primitives de répartition. Ces primitives permettent de programmer un système réparti entier avec un seul et même langage, ainsi que la description de l'architecture et l'expression de la localisation de certains flots ou calculs sur cette architecture. Une sémantique répartie est dans un premier temps proposée comme formalisation de l'exécution répartie de programmes synchrones. Ensuite, nous proposons un système de types à effets permettant d'inférer la localisation des calculs ou des flots non annotées du programme, ainsi que d'assurer à la compilation la consistance des annotations de répartition fournies par le programmeur. Finalement, une opération de projection dirigée par les types est définie. Cette opération permet d'obtenir automatiquement, à partir d'un seul programme synchrone annoté, un programme local destiné à être exécuté par chaque ressource de calcul déclarée dans l'architecture. Ce système de type, ainsi que l'opération de projection, ont été implémentées dans le compilateur du langage Lucid Synchrone.

Mots-clés : Programmation synchrones, répartition automatique, systèmes de type

1 Introduction

Synchronous programming languages [6] 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 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. However, most embedded systems are composed of several computing resources (named "locations"). There are many 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*, or *physical* parallelism. This paper addresses the problem of mapping the functional parallelism onto the physical one, in a modular way.

Programs of a distributed system can be designed separately; but it occurs that in complex and multifunctional embedded systems, functionalities are frequently independent of the hardware architecture, implying conflicts between architecture and functional modularity. Thus, one functionality can use several locations and one location can be involved in several functionalities. As a result, programming separately each location compromises the modularity and is error-prone. This situation occurs within several industrial areas, among which automotive embedded systems, and software-defined radio [16].

Within this context, our motivation is to introduce distribution primitives in a synchronous dataflow language allowing the annotation of a synchronous program without altering its semantics. These annotations allow the programmer to state on which location some values are located or computed. We then use a type and effects system [19], named *spacial* type system hereafter, to infer modularly the localization of non-annotated values. Finally, a projection operation allows us to obtain automatically, from a complete typed program, the local program to be executed on each location.

Our paper is organized as follows: in Section 2, we expose the context and motivations of this work. The primitives added to our language will be presented in Section 2.2, the spacial type system in Section 2.3, some examples in Section 2.4,

and an application in Section 2.5. Section 3 will then present the semantics, and the formalization of the spacial type system. Section 4 shows the type-based distribution method. Finally, related work and discussion about the solution will be exposed in Section 5.

2 Motivations and Overview

2.1 Distribution of Synchronous Programs

Dataflow languages, commonly used to describe block-diagram systems, express the manipulation of infinite streams of values as primitive values. Consequently, the notation `1` represents the infinite sequence $1, 1, ...$ (in the following, we will denote by x_i the ith value of the stream x). In the same way, `int` stands for the type of infinite sequences of integers. In this context, functions (called nodes hereafter) are stream functions (e.g., $\texttt{int} \rightarrow \texttt{int}$ stands for the type of functions from integer streams to integer streams). Combinatorial functions are implicilty lifted to apply pointwise to their arguments (e.g., if $\texttt{x} = (x_i)_{i \in I\!N}$ and $\texttt{y} = (y_i)_{i \in I\!N}$ are two integer streams, $(\texttt{x} + \texttt{y}) = (x_i + y_i)_{i \in I\!N}$.

The classical compilation method involves compiling a synchronous program into one 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 [3]. We will adapt 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 local state, from the values of inputs, other incoming communication channels, and the current local state.

2.2 Language-based Distribution

We consider *functional* distribution: distribution is not achieved for the sake of performance but because the system described is intrinsically distributed. The 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, since problems such as the scheduling of communications or the type consistency of the communicated data arise. The usual method, using architecture languages such as 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 some conflicts between architectural and functional modularity can occur, it is more efficient and less error-prone to define the system as a whole, together with architectural annotations [12]. Our contribution consequently involves providing language primitives, to allow the programmer to describe the hardware 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: the declaration of a location 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 4.4 a *projection* operation, which operates w.r.t. a symbolic name, and which produces a single non-distributed synchronous program to be executed at the physical location represented by this symbolic name.

The declaration of a physical location `A` follows the syntax: `loc A`. The existence of a link from `A` to `B` is stated by: `link A to B`. Note that we distinguish *communication links* from *communication channels*, introduced in Section 3.4: communication links, specified by the primitive `link`, state the *possibility* of communications 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) will be located at A. The programmer does not need to express the localization of every value. A type system is provided, whose double function is to check the validity of the localization expressed in regard to the architecture (w.r.t. existing communication links), 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 localization inside the code. The technical expression of communications are let to the further phase of integration on actual

architecture: our method only deal with inferring 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). Such choice concern a compromise, from a programmer point of view, between no control at all (communications are possible everywhere) and absolute control (the programmer expresses every communication). We will currently assume that communications can occur everywhere in the code, and show later how this can be refined.

2.3 A Type System for the Automatic Distribution

The main difficulty of synchronous dataflow programs distribution lies in the preservation of the control flow. Usually, such programs are inlined before distribution [9]: as a result, the control flow is materialized by classic control structures such as if/then/elses. 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. Hence, to preserve the coherence of the control flow on all the locations, we must place on each location, along with functions defined on it, the functions containing the calls to these functions, and so on. 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.

Consequently, the complete control flow of the program cannot be built for analysing its consistency. Hence, type systems are provided so as to perform such an analysis. Besides the classical type system used for data consistency analysis, we provide a special type system dedicated to the distributed execution of the program. We call it a *spacial type system* and, when clear from context, we shall simply refer to it as a type system (since this paper does not address classical typing). This spacial type system describes the localization of streams, and a type-directed approach is followed to achieve code distribution.

The 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), spacial types will be inferred from the available spacial annotations in the source. The spacial type system also checks the consistency of these annotations with the given architecture. Spacial 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, spacial types will be inferred for each defined node modularly. The type system presented is a type and effects system [19].

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 spacial 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 spacial type is a set: it is the set of locations where the stream will be available. The spacial type of a node f is written $t_i \multimap\!\langle S \rangle\!\rightarrow t_o$, where t_i and t_o are respectively the spacial 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_i 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 comprises a conditional control structure, allowing a node `f(x)` to be computed, either at `A`, or at `B`, according to the value of its input `m`. We assume that `f` is of spacial type $\forall\delta.c$ `at` $\delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow c$ `at` δ.

```
node g(m,x) = z with
  if m then
    let y = f(x) at A in z = y
  else
    let y = f(x) at B in z = y
```

The values defined in the two branches of the `if` have to be of the same type, i.e., at the same location. Furthermore, the `if`/`then`/`else` structure has to be duplicated on `A` and `B`, so as to be able to decide, at both locations, whether to apply `f` or not. Both values `m` and `x` must also be available on both locations. As the only existing link in the architecture is from `A` to `B`, the only way to resolve such constraints is to locate `m` and `x` at `A`, and the node output at `B`. The spacial type of the node `g` is therefore:

$$c \texttt{ at } A \times c \texttt{ at } A \multimap\!\langle\{A, B\}\rangle\!\rightarrow c \texttt{ at } B$$

It can be noted here that this node cannot be used within a located declaration. The program below will be rejected by our type system.

```
node g'(m,x) = y with
  y = g(m,x) at A
```

The second example is a sequence of three nodes `f1`, `f2` and `f3`, each assumed to be of spacial type $\forall\delta.c$ `at` $\delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow c$ `at` δ. `f1` and `f3` are localized by the programmer, respectively on `A` and `B`. `f2` is not explicitely 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 spacial type c `at` $A \multimap\!\langle\{A,B\}\rangle\!\rightarrow c$ `at` B. As the localization of computations has to be done modularly, a spacial type for `f2` will be given once, among the two possibilities c `at` $A \multimap\!\langle\{A\}\rangle\!\rightarrow c$ `at` A and c `at` $B \multimap\!\langle\{B\}\rangle\!\rightarrow c$ `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
```

The fourth 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 the location `A`, and then `g` to the result of the first application at the location B.

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

The spacial type of `h` is then:

$$\forall\alpha,\beta,\gamma.\Big(\begin{array}{l}(\alpha \texttt{ at } A \multimap\!\langle\{A\}\rangle\!\rightarrow \beta \texttt{ at } A)\\ \quad\times(\beta \texttt{ at } B \multimap\!\langle\{B\}\rangle\!\rightarrow \gamma \texttt{ at } B)\\ \qquad\times(\alpha \texttt{ at } A)\end{array}\Big) \multimap\!\langle\{A,B\}\rangle\!\rightarrow \gamma \texttt{ at } B$$

The fifth example is the same higher-order node, but without any annotations:

```
node h (f,g,x) = z with
       y = f(x)
  and z = g(y)
```

The spacial type of `h` is then:

$$\forall\alpha,\beta,\gamma.\forall\delta.\Big(\ (\alpha\ \texttt{at}\ \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow \beta\ \texttt{at}\ \delta) \times(\beta\ \texttt{at}\ \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow \gamma\ \texttt{at}\ \delta) \times(\alpha\ \texttt{at}\ \delta)\Big) \multimap\!\langle\{\delta\}\rangle\!\rightarrow \gamma\ \texttt{at}\ \delta$$

As each node is projected to specialized code for each symbolic location, when no annotation is provided, we must enforce each instance of the node to be executed at one single location; this location can be any symbolic location, and is also generalized.

Finally, a node, for more modularity, can be defined with local locations. These new locations are introduced as a list between [...], can then be used within the node. This higher-order node uses two location variables δ_1 and δ_2:

```
node h [δ1,δ2] (f,g,x) = z with
       y = f(x) at δ1
  and z = g(y) at δ2
```

`h` receives then the spacial type:

$$\forall\alpha,\beta,\gamma.\forall\delta_1,\delta_2 : \{\delta_1 \triangleright \delta_2\}.\Big(\ (\alpha\ \texttt{at}\ \delta_1 \multimap\!\langle\{\delta_1\}\rangle\!\rightarrow \beta\ \texttt{at}\ \delta_1) \times(\beta\ \texttt{at}\ \delta_2 \multimap\!\langle\{\delta_2\}\rangle\!\rightarrow \gamma\ \texttt{at}\ \delta_2) \times(\alpha\ \texttt{at}\ \delta_1)\Big) \multimap\!\langle\{\delta_1,\delta_2\}\rangle\!\rightarrow \gamma\ \texttt{at}\ \delta_2$$

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 $\delta = \delta' = s$, either 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 these two ways (assuming the existence of two nodes `f` and `g`, both of spacial type $\forall\delta.c\ \texttt{at}\ \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow c\ \texttt{at}\ \delta$):

```
    y1 = h (fat A,gat A,x1)
and y2 = h (fat A,gat 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, as it is the case in 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 [16]. This allows in particular the reconfiguration, possibly dynamic, 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 performances, 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 either several standards, or several functionalities [13]. In other word, there is a strong motivation for dynamicity.

Let us study the case of a multichannel reception system that supports the two mobile standards GSM and UMTS:

- The GSM standard involves a filter for 1800 MHz frequencies, a GMSK demodulator, and a CRC / convolutional error correction module.
- The UMTS standard 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.

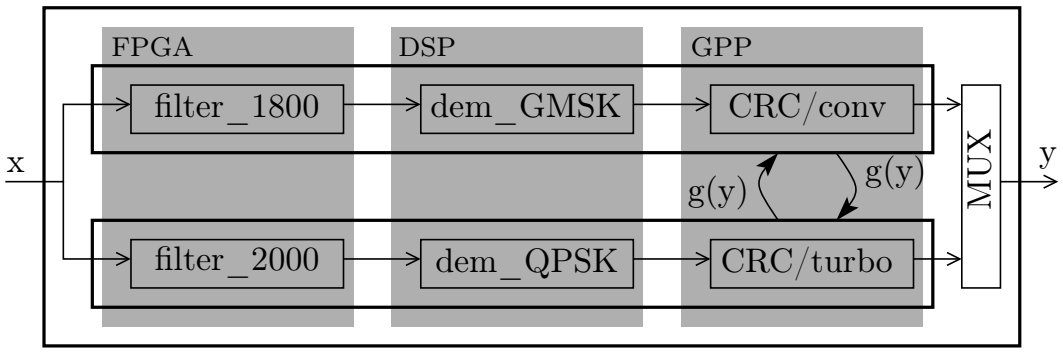

Figure 1: Functional model of a multichannel software radio.

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:

1. 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 correction of the system. This situation compromises the modularity of the system.

2. 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 claims for 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.

Figure 2 shows the implementation of this multichannel software radio, with the extended language introduced in preceding sections. This implementation strictly

```
node channel(filter,demod,crc,x) = y with
      f = filter(x) at FPGA
  and d = demod(f) at DSP
  and y = crc(d) at GPP

node multichannel_sdr(x) = y with
      c = gsm_or_umts(y) at GPP in
  and
    if (true fby c) then
      y = channel(filter_1800,gmsk,conv,x)
    else
      y = channel(filter_2000,qpsk,turbo,x)
```

Figure 2: Multichannel software radio implementation

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 spacial type $\forall\delta.c$ `at` $\delta \multimap\!\langle\{\delta\}\rangle\!\rightarrowtail c$ `at` δ. Since the conditional construct comprises declarations that have to be executed on the set of locations $\{$`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 definition of `fby` is: $(\texttt{xfbyy})_i = x_0$ if $i = 0$, y_{i-1} else). The distribution of this example will put a copy of this `if`/`then`/`else` on these three locations. Finally, the expression `true fby c` 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 spacial type of `x` is `FPGA`, and the one of `y` is `GPP`. As a result, the spacial type of the node `multichannel_sdr` is:

$$(c \texttt{ at FPGA}) \multimap\!\langle\{\texttt{FPGA}, \texttt{DSP}, \texttt{GPP}\}\rangle\!\rightarrowtail (c \texttt{ at GPP})$$

3 Formalization

This section is organized as follows: we first define a synchronous dataflow core language in Section 3.1 and give it its centralized semantics (Section 3.2) and its distributed semantics (Section 3.3). The centralized semantics is considered to be the reference semantics and we only consider programs which reacts with respect to this semantics. Programs that do not react (e.g., for typing or causality reasons) are assumed to be rejected by other means [10, 11]. The distributed semantics allows us to give a meaning to location annotations. A spacial type system is then presented in 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. These annotations are then used by a type-directed projection operation which produces a program for each location (Section 4.4).

3.1 The Core Language Syntax

A program is made of an architecture description ($\mathcal{A}$), 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 (A `link to` A) which state the existence of a communication link from one location to another. A location s is either a location variable δ, 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` $f[\delta_1, \ldots, \delta_n](x) = e$ `with` D).

Definitions D are either single equations ($x = e$), definitions naming the result of an application ($x = x(e)$), 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), a binary combinatory operation ($\mathrm{op}(e, e)$, where op can be $(+)$, $(-)$,...), an initialized delay (e `fby` e), an access function (`fst` e and `snd` e), or an expression annotated with an explicit location s (e `at` s).

$$
\begin{array}{lcl}
P & ::= & \mathcal{A}\,;d\,;D \\
\mathcal{A} & ::= & \mathcal{A}\,;\mathcal{A} \mid \texttt{loc}\ A \mid A\ \texttt{link to}\ A \\
s & ::= & \delta \mid A \\
d & ::= & \texttt{node}\ f[\delta_1, \ldots, \delta_n](x) = e\ \texttt{with}\ D \mid d\,;d \\
D & ::= & x = e \mid x = x(e) \mid D\ \texttt{and}\ D \\
 & & \mid\ \texttt{if}\ e\ \texttt{then}\ D\ \texttt{else}\ D \\
e & ::= & i \mid x \mid (e, e) \mid \mathrm{op}(e, e) \mid e\ \texttt{fby}\ e \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid e\ \texttt{at}\ s
\end{array}
$$

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. A sequence of location parameters $\vec{\delta}$ is associated to functions.

$$v ::= i \mid (v, v) \mid \delta\lambda x.e \texttt{ with } D$$
$$R ::= [v_1/x_1, \ldots, v_n/x_n]$$

An environment R associates values to names and assumes that names are pairwise distinct (for all $i \neq j$, $x_i \neq x_j$).

Given a sequence d of node definitions $\texttt{node } f_i[\vec{\delta_i}](x_i) = e_i \texttt{ with } D_i$, an initial global environment R_d is defined, holding λ-values of each f_i. This initial environment will be given as input of the main program.

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

The synchronous centralized semantics is defined by mean of two reaction predicates. $R \vdash e_1 \xrightarrow{v} e_2$ 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 $S_{in} \vdash P : S_{out}$, meaning that under a sequence of input environments $S_{in} = R_1.R_2\ldots$, the program $P = \mathcal{A};d;D$ produces a sequence of output environments $S_{out} = R'_1.R'_2\ldots$ such that (denoting $\mathrm{hd}(R.S) = R$ and $\mathrm{tl}(R.S) = S$):

$$\frac{R_d, \mathrm{hd}(S_{in}), \mathrm{hd}(S_{out}) \vdash D \xrightarrow{\mathrm{hd}(S_{out})} D' \qquad \mathrm{tl}(S_{in}) \vdash \mathcal{A};d;D' : \mathrm{tl}(S_{out})}{S_{in} \vdash \mathcal{A};d;D : S_{out}}$$

The rules for the reaction predicates are given in Figure 3. 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 `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 OP). Pair construction and destruction follow classical rules (rules PAIR, FST and SND). Note that locations are not taken into account here (rule AT): annotations added by the programmer does not alter the centralized semantics of the program (i.e., its functionality). An

equation $x = e$ emits the reaction environment defining x (rule DEF). A sequential function application is replaced by its body and argument definition (rule APP). The rule AND states that parallel equations are mutually recursive. The rule LET stands for sequential definitions. A conditional statement executes its first branch if its condition is true (rule IF-1) and its second branch otherwise (rule IF-2)

3.3 The Distributed Synchronous Semantics

The distributed semantics also operates on a program $P = \mathcal{A}; d; D$, 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 *spacialized execution*: the values $\hat{v}$ emitted by expressions are now *distributed values*, i.e., they are annotated with location informations stating how these values are distributed on the architecture.

$$\begin{array}{lll} \hat{v} & ::= & dv \texttt{ at } s \\ dv & ::= & i \mid (\hat{v}, \hat{v}) \mid \Lambda\vec{\delta}.\lambda x.e \texttt{ with } D \\ \hat{R} & ::= & [\hat{v}_1/x_1, \ldots, \hat{v}_n/x_n] \\ s & ::= & A \mid \top \\ G & ::= & \langle \mathcal{S}, \mathcal{L} \rangle \end{array}$$

A location s stands either for a single location A or a special value $\top$ abstracting a set of locations. The following definition of $\sqcup$ holds:

$$s \sqcup s = s \qquad\qquad \top \sqcup s = s \sqcup \top = \top$$

From this definition of distributed values, we can observe that only a subset of such values are meaningful: for distributed pairs, the values $(dv \texttt{ at } A, dv' \texttt{ at } A) \texttt{ at } A$ (i.e., a centralized pair) and $(dv \texttt{ at } A, dv' \texttt{ at } B) \texttt{ at } \top$ (a distributed pair) are meaningful, whereas no meaning can be associated to the distributed value $(dv \texttt{ at } A, dv' \texttt{ at } B) \texttt{ at } A$. The meaningful values are defined as *well-formed values*: a distributed value $\hat{v}$ is *well-formed* iff:

- either $\hat{v} = i \texttt{ at } A$ or $\hat{v} = \lambda x.e \texttt{ with } D \texttt{ at } A$;
- or, $\hat{v} = (dv_1 \texttt{ at } s_1, dv_2 \texttt{ at } s_2) \texttt{ at } s$, where $(dv_i \texttt{ at } s_i)_{i=1,2}$ are well-formed, and $s = s_1 \sqcup s_2$.

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

$$\begin{array}{lll} \mathrm{loc}(i \texttt{ at } s) & = & \{s\} \\ \mathrm{loc}(\lambda x.e \texttt{ with } D \texttt{ at } s) & = & \{s\} \\ \mathrm{loc}((\hat{v}_1, \hat{v}_2) \texttt{ at } s) & = & \mathrm{loc}(\hat{v}_1) \cup \mathrm{loc}(\hat{v}_2) \end{array}$$

$$\text{(Imm)}\quad R \vdash i \xrightarrow{i} i \qquad \text{(Inst)}\quad R, [v/x] \vdash x \xrightarrow{v} x$$

$$\text{(Fby)}\quad \frac{R \vdash e_1 \xrightarrow{v_1} e_1' \qquad R \vdash e_2 \xrightarrow{v_2} e_2'}{R \vdash e_1 \texttt{ fby } e_2 \xrightarrow{v_1} v_2 \texttt{ fby } e_2'}$$

$$\text{(Op)}\quad \frac{R \vdash e_1 \xrightarrow{i_1} e_1' \qquad R \vdash e_2 \xrightarrow{i_2} e_2' \qquad i = \mathrm{op}(i_1, i_2)}{R \vdash \mathrm{op}(e_1, e_2) \xrightarrow{i} \mathrm{op}(e_1', e_2')}$$

$$\text{(Pair)}\quad \frac{R \vdash e_1 \xrightarrow{v_1} e_1' \qquad R \vdash e_2 \xrightarrow{v_2} e_2'}{R \vdash (e_1, e_2) \xrightarrow{(v_1, v_2)} (e_1', e_2')} \qquad \text{(Fst)}\quad \frac{R \vdash e \xrightarrow{(v_1, v_2)} e'}{R \vdash \texttt{fst } e \xrightarrow{v_1} \texttt{fst } e'} \qquad \text{(Snd)}\quad \frac{R \vdash e \xrightarrow{(v_1, v_2)} e'}{R \vdash \texttt{snd } e \xrightarrow{v_2} \texttt{snd } e'}$$

$$\text{(At)}\quad \frac{R \vdash e \xrightarrow{v'} e'}{R \vdash e \texttt{ at } s \xrightarrow{v} e' \texttt{ at } s} \qquad \text{(Def)}\quad \frac{R \vdash e \xrightarrow{v} e'}{R \vdash x = e \xrightarrow{[v/x]} x = e'}$$

$$\text{(App)}\quad \frac{R(f) = \lambda y.e \texttt{ with } D \qquad R \vdash x = e \texttt{ and } y = e' \texttt{ and } D \xrightarrow{R'} D'}{R \vdash x = f(e') \xrightarrow{R'} D'}$$

$$\text{(And)}\quad \frac{R, R_2 \vdash D_1 \xrightarrow{R_1} D_1' \qquad R, R_1 \vdash D_2 \xrightarrow{R_2} D_2'}{R \vdash D_1 \texttt{ and } D_2 \xrightarrow{R_1, R_2} D_1' \texttt{ and } D_2'}$$

$$\text{(If-1)}\quad \frac{R \vdash e \xrightarrow{\texttt{true}} e' \qquad R \vdash D_1 \xrightarrow{R'} D_1'}{R \vdash \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{R'} \texttt{if } e' \texttt{ then } D_1' \texttt{ else } D_2}$$

$$\text{(If-2)}\quad \frac{R \vdash e \xrightarrow{\texttt{false}} e' \qquad R \vdash D_2 \xrightarrow{R'} D_2'}{R \vdash \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{R'} \texttt{if } e' \texttt{ then } D_1 \texttt{ else } D_2'}$$

Figure 3: Centralized synchronous semantics

The $|\cdot|$ operator erases annotations from a distributed value to get a centralised value:

$$\begin{aligned} |i \texttt{ at } s| &= i \\ |\lambda x.e \texttt{ with } D \texttt{ at } s| &= \lambda x.e \texttt{ with } D \\ |(\hat{v}_1, \hat{v}_2) \texttt{ at } s| &= (|\hat{v}_1|, |\hat{v}_2|) \end{aligned}$$

This operator extends straightforwardly to reaction environments.

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

We note $\mathcal{S}$ the set of declared constant locations, and $\mathcal{L} \subseteq \mathcal{S} \times \mathcal{S}$ the set of declared communication links. The relation $\mathcal{L}$ defines the *possibility* of communications, 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 $\mathcal{A}$ defines an architecture graph G: the notation $G \vdash \mathcal{A} : G'$ means that given the architecture graph G, $\mathcal{A}$ defines the new architecture graph G'. The rules Arch, Def-Loc and Def-Link define this predicate:

$$\text{(Arch)}\quad \frac{\langle \mathcal{S}, \mathcal{L} \rangle \vdash \mathcal{A}_1 : \langle \mathcal{S}_1, \mathcal{L}_1 \rangle \qquad \langle \mathcal{S}_1, \mathcal{L}_1 \rangle \vdash \mathcal{A}_2 : \langle \mathcal{S}_2, \mathcal{L}_2 \rangle}{\langle \mathcal{S}, \mathcal{L} \rangle \vdash \mathcal{A}_1 ; \mathcal{A}_2 : \langle \mathcal{S}_2, \mathcal{L}_2 \rangle}$$

$$\text{(Def-Loc)}\quad \langle \mathcal{S}, \mathcal{L} \rangle \vdash \texttt{loc } A : \langle \mathcal{S} \cup \{A\}, \mathcal{L} \rangle$$

$$\text{(Def-Link)}\quad \frac{A_1, A_2 \in \mathcal{S}}{\langle \mathcal{S}, \mathcal{L} \rangle \vdash \texttt{link } A_1 \texttt{ to } A_2 : \langle \mathcal{S}, \mathcal{L} \cup \{A_1 \mapsto A_2\} \rangle}$$

For clarity reasons, we assume that the graph G defined by an architecture is global for subsequent semantic rules. The annotated execution of a program P is denoted $\hat{S}_{in} \Vdash P : \hat{S}_{out}$, meaning that under a sequence of input environments $\hat{S}_{in} = \hat{R}_1.\hat{R}_2 \ldots$, the program $P = \mathcal{A}; d; D$ produces a sequence of output environments $\hat{S}_{out} = \hat{R}'_1.\hat{R}'_2 \ldots$ such that:

$$\frac{\langle\emptyset,\emptyset\rangle \vdash \mathcal{A} : G \qquad \hat{R}_d, \mathrm{hd}(\hat{S}_{in}), \mathrm{hd}(\hat{S}_{out}) \overset{\ell}{\Vdash} D \xrightarrow{\mathrm{hd}(\hat{S}_{out})} D' \qquad \mathrm{tl}(\hat{S}_{in}) \Vdash \mathcal{A};d;D' : \mathrm{tl}(\hat{S}_{out})}{\hat{S}_{in} \Vdash \mathcal{A};d;D : \hat{S}_{out}}$$

Where, as for the centralized semantics, $\hat{R}_d$ defined from the sequence of node definitions $d = \texttt{node}\ f_i[\vec{\delta_i}](x_i) = e_i\ \texttt{with}\ D_i$ as:

$$\hat{R}_d = [\Lambda\vec{\delta_1}.\lambda x_1.e_1\ \texttt{with}\ D_1/f_1, \ldots, \Lambda\vec{\delta_n}.\lambda x_n.e_n\ \texttt{with}\ D_n/f_n]$$

The rules for the predicates $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{\hat{v}} e_2$ and $\hat{R} \overset{\ell}{\Vdash} D_1 \xrightarrow{\hat{R}'} D_2$ are given in Figures **??** and 5. An immediate value can be emitted anywhere (rule IMM). Rule 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 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 OP). An annotated expression must involve at most the location stated for its computation (rule AT). An application involves choosing a set of constant locations, and replacing location parameters by these locations in the expression and the declaration (rule AT). 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 in a well-formed environment, a program reacts only by emitting well-formed values.

Lemma 1. *For all $D, D', \hat{R}, \hat{R}'$ such that $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$, if $\hat{R}$ is well-formed, then $\hat{R}'$ is well-formed.*

Proof. By induction on the structure of the expression or declaration. The proof lies on the definitions of well-formed values, and the $\sqcup$ operation:

- if a value $(\hat{v}_1, \hat{v}_2)\ \texttt{at}\ s$ is well-formed, then $\hat{v}_1$ and $\hat{v}_2$ are well-formed (case $e = \texttt{fst}\ e'$ or $e = \texttt{snd}\ e'$) ;
- if two values $dv_1\ \texttt{at}\ s_1$ and $dv_2\ \texttt{at}\ s_2$ are well-formed, then the value:

  $$(dv_1\ \texttt{at}\ s_1, dv_2\ \texttt{at}\ s_2)\ \texttt{at}\ s_1 \sqcup s_2$$

  is well-formed (case $e = (e_1, e_2)$) ;

$$\text{(Imm)}\quad \hat{R} \overset{\ell}{\Vdash} i \xrightarrow{i \texttt{ at } s} i \qquad \text{(Inst)}\quad \hat{R}, [\hat{v}/x] \overset{\mathrm{loc}(\hat{v})}{\Vdash} x \xrightarrow{\hat{v}} x$$

$$\text{(Comm)}\quad \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{dv \texttt{ at } s} e' \qquad (s, s') \in \mathcal{L}}{\hat{R} \overset{\ell \cup \{s'\}}{\Vdash} e \xrightarrow{dv[s'/s] \texttt{ at } s'} e'}$$

$$\text{(Fby)}\quad \frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{\hat{v}_1} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{\hat{v}_2} e_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} e_1 \texttt{ fby } e_2 \xrightarrow{\hat{v}_1} |\hat{v}_2| \texttt{ fby } e_2'}$$

$$\text{(Op)}\quad \frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{i_1 \texttt{ at } A} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{i_2 \texttt{ at } A} e_2' \qquad i = \mathrm{op}(i_1, i_2)}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} \mathrm{op}(e_1, e_2) \xrightarrow{i \texttt{ at } A} \mathrm{op}(e_1', e_2')}$$

$$\text{(Pair)}\quad \frac{\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{dv_1 \texttt{ at } s_1} e_1' \qquad \hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{dv_2 \texttt{ at } s_2} e_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} (e_1, e_2) \xrightarrow{(dv_1 \texttt{ at } s_1, dv_2 \texttt{ at } s_2) \texttt{ at } s_1 \sqcup s_2} (e_1', e_2')}$$

$$\text{(Fst)}\quad \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{(\hat{v}_1, \hat{v}_2) \texttt{ at } s} e'}{\hat{R} \overset{\ell}{\Vdash} \texttt{fst } e \xrightarrow{\hat{v}_1} \texttt{fst } e'} \qquad \text{(Snd)}\quad \frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{(\hat{v}_1, \hat{v}_2) \texttt{ at } s} e'}{\hat{R} \overset{\ell}{\Vdash} \texttt{snd } e \xrightarrow{\hat{v}_2} \texttt{snd } e'}$$

$$\text{(At)}\quad \frac{\hat{R} \overset{\{s\}}{\Vdash} e \xrightarrow{\hat{v}} e'}{\hat{R} \overset{\{s\}}{\Vdash} e \texttt{ at } s \xrightarrow{\hat{v}} e' \texttt{ at } s}$$

Figure 4: Distributed synchronous semantics (expressions)

(DEF)
$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'}{\hat{R} \overset{\ell}{\Vdash} x = e \xrightarrow{[\hat{v}/x]} x = e'}$$

(APP)
$$\frac{\hat{R}(f) = \Lambda\delta_1, \ldots, \delta_n.\lambda y.e \texttt{ with } D \texttt{ at } s \qquad \{s_1, \ldots, s_n\} \subseteq \mathcal{S} \qquad \hat{R} \overset{\ell}{\Vdash} x = e[\vec{s}/\vec{\delta}] \texttt{ and } y = e' \texttt{ and } D[\vec{s}/\vec{\delta}] \xrightarrow{\hat{R}'} D'}{\hat{R} \overset{\ell}{\Vdash} x = f(e') \xrightarrow{\hat{R}'} D'}$$

(AND)
$$\frac{\hat{R}, \hat{R}_2 \overset{\ell_1}{\Vdash} D_1 \xrightarrow{\hat{R}_1} D_1' \qquad \hat{R}, \hat{R}_1 \overset{\ell_2}{\Vdash} D_2 \xrightarrow{\hat{R}_2} D_2'}{\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} D_1 \texttt{ and } D_2 \xrightarrow{\hat{R}_1, \hat{R}_2} D_1' \texttt{ and } D_2'}$$

(IF-1)
$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\texttt{true at } s} e' \qquad \hat{R} \overset{\ell'}{\Vdash} D_1 \xrightarrow{\hat{R}'} D_1' \qquad \forall s' \in \ell', s \triangleright s'}{\hat{R} \overset{\ell \cup \ell'}{\Vdash} \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{\hat{R}'} \texttt{if } e' \texttt{ then } D_1' \texttt{ else } D_2}$$

(IF-2)
$$\frac{\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\texttt{false at } s} e' \qquad \hat{R} \overset{\ell'}{\Vdash} D_2 \xrightarrow{\hat{R}'} D_2' \qquad \forall s' \in \ell', s \triangleright s'}{\hat{R} \overset{\ell \cup \ell'}{\Vdash} \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 \xrightarrow{\hat{R}'} \texttt{if } e' \texttt{ then } D_1 \texttt{ else } D_2'}$$

Figure 5: Distributed synchronous semantics (equations)

- rule COMM applies only on values of the form dv `at` s, where $s \neq \top$. For any $s' \neq \top$, if dv `at` s is well-formed, then $dv[s'/s]$ `at` s' is well-formed.

□

Lemma 2 states that if a program reacts with the distributed semantics, then it reacts with the centralized one and produces the same values. The proof is left in appendix.

Lemma 2. *For all* $D, D', \hat{R}, \hat{R}'$, *if* $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$, *then there exists* R, R' *such that* $R = |\hat{R}|$, $R' = |\hat{R}'|$ *and* $R \vdash D \xrightarrow{R'} D'$.

3.4 Spacial Types

For the sake of clarity, we first present a simplified version of the type system. For the projection, we will need a refinement of this first version to take communication channels into account (see Section 4.3).

The syntax of spacial type expressions is:

$$\begin{array}{rcl} \sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \forall \delta_1, \ldots, \delta_n : C.t \\ t & ::= & c \mid \alpha \mid t \texttt{ at } s \mid t \multimap\!\langle \ell \rangle\!\rightarrow t \mid t \times t \\ \ell & ::= & \{s_1, \ldots, s_n\} \\ s & ::= & \delta \mid A \\ H & ::= & H \texttt{ at } A \mid [x_1 : \sigma_1, \ldots, x_n : \sigma_n] \\ C & ::= & \{s_1 \triangleright s'_1, \ldots, s_n \triangleright s'_n\} \end{array}$$

We note H the spacial typing environments. H `at` A denotes a located environment, i.e., a typing environment from which every spacial type will be forced to represent a value entirely located on A.

We distinguish spacial type schemes (σ), which can be quantified, from simple spacial types (t). A set of constraints C can be associated to quantification of location variables ($\forall \delta_1, \ldots, \delta_n : C.t$). We note $\forall \delta_1, \ldots, \delta_n.t$ the scheme $\forall \delta_1, \ldots, \delta_n : \emptyset.t$. A simple spacial type can be either a stream type (c), a type variable (α), a located type (t `at` s), a node type ($s \multimap\!\langle \ell \rangle\!\rightarrow s$), or a pair type ($s \times s$). ℓ denotes sets of locations.

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 constr($\mathcal{L}$).

$$\text{constr}(\mathcal{L}) = \{s \triangleright s' | (s, s') \in \mathcal{L}\}$$

A value of spacial type t $\texttt{at}$ s is a value located on s. A value of spacial type $t_1 \multimap\langle \ell \rangle\rightarrow t_2$ is a node whose input is of spacial type t_1, whose output is of spacial type t_2, and whose computation involves the set of locations ℓ.

The following equalities stand for spacial types:

$$\begin{aligned}(t_1 \times t_2)\ \texttt{at}\ s &= (t_1\ \texttt{at}\ s) \times (t_2\ \texttt{at}\ s)\\ (t_1 \multimap\langle \{s\} \rangle\rightarrow t_2)\ \texttt{at}\ s &= (t_1\ \texttt{at}\ s) \multimap\langle \{s\} \rangle\rightarrow (t_2\ \texttt{at}\ s)\\ t\ \texttt{at}\ s\ \texttt{at}\ s &= t\ \texttt{at}\ s\end{aligned}$$

$$\frac{t_1 = t_1' \quad t_2 = t_2'}{(t_1 \times t_2) = (t_1' \times t_2')} \qquad \frac{t_1 = t_1' \quad t_2 = t_2'}{t_1 \multimap\langle \ell \rangle\rightarrow t_2 = t_1' \multimap\langle \ell \rangle\rightarrow t_2'}$$

A spacial type t is *well-formed* iff

$$\forall t', s_1, s_2, t = t'\ \texttt{at}\ s_1\ \texttt{at}\ s_2 \Rightarrow s_1 = s_2.$$

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

$$(t[t_1/\alpha_1, \ldots, t_n/\alpha_n, s/\delta], C[s_1/\delta_1, \ldots, s_m/\delta_m]) \leq \forall \alpha_1 \ldots \alpha_n \forall \delta_1 \ldots \delta_m : C.t$$

$$(t\ \texttt{at}\ s, C) \leq (H\ \texttt{at}\ s)(x) \Leftrightarrow (t\ \texttt{at}\ s, C) \leq H(x)$$

A type t can be generalized w.r.t. a typing environment H and local locations $\mathcal{S} = \{\delta_1, \ldots, \delta_p\}$ to a type scheme $\forall \alpha_1, \ldots, \alpha_n . \forall \delta_1, \ldots, \delta_p : C.t$, provided that the set of location variable generalized $\{\delta_1, \ldots, \delta_p\}$ is not free in H, and matches with the local architecture. Generalization w.r.t. an empty local architecture allows the introduction of at most one location variable, with no constraint. This means that a node, without any location constraint given by the programmer, will receive a type of the form $\forall \alpha_1, \ldots, \alpha_n . \forall \delta . (t\ \texttt{at}\ \delta)$, meaning that this node is available and computable at any unique location δ.

We note respectively $\mathrm{FLV}(t)$ and $\mathrm{FTV}(t)$ 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 $s \triangleright s' \in C \wedge s \neq s' \Rightarrow (s, s') \in \mathcal{L}$.

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

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

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

$$\begin{array}{ll} \text{locations}(t_1 \times t_2) & = \text{locations}(t_1) \cup \text{locations}(t_2) \\ \text{locations}(t_1 \multimap\!\langle \ell \rangle\!\rightarrow t_2) & = \ell \\ \text{locations}(t \texttt{ at } s) & = \{s\} \end{array}$$

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

$$H_0 = \left[\begin{array}{l} \cdot \texttt{ fby } \cdot : \forall\alpha.\forall\delta.\alpha \texttt{ at } \delta \times \alpha \texttt{ at } \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow \alpha \texttt{ at } \delta, \\ \texttt{fst } \cdot : \forall\alpha,\beta.\forall\delta.\alpha \texttt{ at } \delta \times \beta \texttt{ at } \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow \alpha \texttt{ at } \delta, \\ \texttt{snd } \cdot : \forall\alpha,\beta.\forall\delta.\alpha \texttt{ at } \delta \times \beta \texttt{ at } \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow \beta \texttt{ at } \delta, \\ (+) : \forall\delta.c \texttt{ at } \delta \times c \texttt{ at } \delta \multimap\!\langle\{\delta\}\rangle\!\rightarrow c \texttt{ at } \delta, \\ \ldots \end{array} \right]$$

Our spacial type system is formally defined by the axioms and inference rules shown in Figure 6.

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 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 `at` A) involves building a located typing environment (rule At). Communications are expressed as subtyping (rule Comm).

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

Typing an equation $x = e$ involves building a singleton typing environment (rule Def). Rule App states that an application must be evaluated on the union of the

$$\text{(Prog)}\quad \frac{\langle\emptyset,\emptyset\rangle \vdash \mathcal{A} : G \qquad H_0|G \vdash d : H/\ell \qquad H, H_1|G \vdash D : H_1/\ell'}{\vdash \mathcal{A}; d; D : H_1}$$

$$\text{(Imm)}\quad H|G \vdash i : c \texttt{ at } s/\{s\} \qquad\qquad \text{(Inst)}\quad \frac{(t, C) \leq (H(x)) \qquad \mathcal{L} \models C}{H|\langle\mathcal{S}, \mathcal{L}\rangle \vdash x : t/\operatorname{locations}(t)}$$

$$\text{(Pair)}\quad \frac{H|G \vdash e_1 : t_1/\ell_1 \qquad H|G \vdash e_2 : t_2/\ell_2}{H|G \vdash (e_1, e_2) : t_1 \times t_2/\ell_1 \cup \ell_2}$$

$$\text{(At)}\quad \frac{H \texttt{ at } s|\langle\mathcal{S}, \mathcal{L}\rangle \vdash e : t/\ell \qquad s \in \mathcal{S}}{H|\langle\mathcal{S}, \mathcal{L}\rangle \vdash e \texttt{ at } s : t/\ell} \qquad \text{(Comm)}\quad \frac{H|\langle\mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } s/\ell \qquad \mathcal{L} \models s \triangleright s'}{H|\langle\mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } s'/\ell \cup \{s'\}}$$

$$\text{(Node)}\quad \frac{\begin{array}{c} H, x : t, H_1|\langle\mathcal{S}', \mathcal{L}'\rangle \vdash D : H_1/\ell_1 \qquad H, x : t, H_1|\langle\mathcal{S}', \mathcal{L}'\rangle \vdash e : t/\ell_2 \qquad \mathcal{S}' = \mathcal{S} \cup \{\delta_1, \ldots, \delta_n\} \\ \mathcal{L}' \subseteq \mathcal{L} \cup (\{\delta_1, \ldots, \delta_n\} \times \mathcal{S}) \cup (\mathcal{S} \times \{\delta_1, \ldots, \delta_n\}) \qquad \{\alpha_1, \ldots, \alpha_m\} = \operatorname{FTV}(t) - \operatorname{FTV}(H) \\ C = \operatorname{constr}(\mathcal{L}' \setminus \mathcal{L}) \qquad \sigma = \forall\alpha_1, \ldots, \alpha_m.\forall\delta_1, \ldots, \delta_n : C.t \multimap\langle\ell_1 \cup \ell_2\rangle\rightarrow t_1 \end{array}}{H|G \vdash \texttt{node } f[\delta_1, \ldots, \delta_n](x) = e \texttt{ with } D : [\sigma/f]/\ell_1 \cup \ell_2}$$

$$\text{(Def)}\quad \frac{H|G \vdash e : t/\ell}{H|G \vdash x = e : [t/x]/\ell} \qquad \text{(App)}\quad \frac{H|G \vdash f : t_1 \multimap\langle\ell_1\rangle\rightarrow t_2/\ell_2 \qquad H|G \vdash e : t_1/\ell_3}{H|G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3}$$

$$\text{(And)}\quad \frac{H|G \vdash D_1 : H_1/\ell_1 \qquad H|G \vdash D_2 : H_2/\ell_2}{H|G \vdash D_1 \texttt{ and } D_2 : H_1, H_2/\ell_1 \cup \ell_2} \qquad \text{(Let)}\quad \frac{H|G \vdash D_1 : H_1/\ell_1 \qquad H, H_1|G \vdash D_2 : H_2/\ell_2}{H|G \vdash \texttt{let } D_1 \texttt{ in } D_2 : H_2/\ell_1 \cup \ell_2}$$

$$\text{(If)}\quad \frac{\begin{array}{c} H|G \vdash e : c \texttt{ at } s/\ell \\ H|G \vdash D_1 : H'/\ell_1 \qquad H|G \vdash D_2 : H'/\ell_2 \qquad \mathcal{L} \models \{s \triangleright s' | s' \in \ell_1 \cup \ell_2\} \end{array}}{H|G \vdash \texttt{if } e \texttt{ then } D_1 \texttt{ else } D_2 : H'/\ell \cup \ell_1 \cup \ell_2}$$

Figure 6: Spacial type system

set of locations where the node f and its argument e must be evaluated, and the set of locations ℓ_1 involved in the computation of the node f. Parallel and sequential declarations involve, for their computations, the union of the sets and of locations involved in the computation of their compounds (rules AND and LET). 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).

A set of locations ℓ is related to an abstract location, as defined in Section 3.3, by:

$$\overline{\ell} = \begin{cases} s & \text{if } \ell = \{s\} \\ \top & \text{else.} \end{cases}$$

We denote by $\hat{v} : t$ the fact that the distributed value $\hat{v}$ has spacial type t:

$$\frac{s \neq \top}{dv \texttt{ at } s : c \texttt{ at } s} \qquad \frac{\hat{v}_1 : t_1 \qquad \hat{v}_2 : t_2}{(\hat{v}_1, \hat{v}_2) : t_1 \times t_2}$$

We denote by $\hat{R} : H$ the type compatibility between $\hat{R}$ and H:

$$\hat{R} : H \Leftrightarrow \forall x \in \mathrm{dom}(\hat{R}), x \in \mathrm{dom}(H) \\ \wedge\, \exists (t, C) \text{ s.t. } (t, C) \leq H(x) \wedge \hat{R}(x) : t$$

Theorem 1 states that if a program reacts with the centralized semantics, and is accepted by the spacial type system, then there exists a spacialized execution such that the distributed values of this execution are equal to the centralized ones. The spacial types are preserved by this spacial execution. The proof is in appendix.

Theorem 1 (Correction)**.** *For all D, D', H, H', R, R', G, if $H|G \vdash D : H'/\ell$ and $R \vdash D \xrightarrow{R'} D'$, then there exists $\hat{R}, \hat{R}'$ such that $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$, $\hat{R} : H$, $\hat{R}' : H'$, $|\hat{R}| = R$ and $|\hat{R}'| = R'$.*

3.5 Implementation

The spacial type system presented in the previous section lies on a subtyping mechanism. It corresponds to the case when communications can occur *anywhere* in the code. This situation raises two problems. The first one is that the implementation of type systems with subtyping mechanism is costly: usual algorithms involve the systematic application of the subtyping rule. The second one is that this choice leads

to a situation where the programmer has no control on 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 show here how to refine our type system in order to address these two problems.

We restrain communicated values to be variables introduced by equations ($x = e$). Thus, within the program given in Section 2.2, 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 at instantiation points.

The refined type system, noted $H|G \vdash_i e : t/\ell$, involves the suppression of the rule Comm and the modification of the rule Def as follows:

$$\text{(Def)}\quad \frac{H|G \vdash_i e : t \texttt{ at } s/\ell}{H|G \vdash_i x = e : [\forall \delta : \{s \triangleright \delta\}.t \texttt{ at } \delta/x]/\ell}$$

Typing an equation $x = e$ involves locating the result of e on a location s, and building a typing environment allowing x to be on any location accessible from s ($x : \forall \delta : \{s \triangleright \delta\}.t \texttt{ at } \delta$) (rule Def). The other rules remain the same.

We then show the correction of the refinement, i.e., that every program accepted by the refined type system is accepted by the original type system.

Theorem 2 (Refined type system correction)**.** *For all H, H', D, G, ℓ, if $H|G \vdash_i D : H'/\ell$, then $H|G \vdash D : H'/\ell$.*

Proof. By induction on the structure of D and e. Whenever the rule Inst of the refined type system can be applied, we can apply the rule Inst, then the rule Comm, of the original type system. □

This refined type system has been implemented in the compiler of Lucid Synchrone.

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 s. 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 `f` 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 (named here `c_y`) on `f_A`, together with an additional input on `f_B`. Here, `c_y` holds the value `y`, computed on `A` and used on `B`. Original inputs and outputs are not suppressed: `()` 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`.

```
node f_A(x) = ((),c_y) with
   c_y = g(x)

node f_B(x,c_y) = z 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)`.

```
(y_A,c_y) = f_A(x);          | receive(c_y);
send(c_y)                    | y_B = f_B((),c_y)
```

4.2 Example

The result of the projection of the two nodes of the Section 2.5 on the location `DSP` is given on Figure 7. 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`.

```
node channel(filter,demod,crc,x,c1) = ((),c2) with
      d = demod(c1)
  and c2 = d

node multichannel_sdr(x,s,c1,c2,c3) = (y,c4,c5)
  if c1 then
    (y,c4) = channel(filter_1800,gmsk,conv,x,c2)
  else
    (y,c5) = channel(filter_2000,qpsk,turbo,x,c3)
```

Figure 7: Result of the projection on `DSP`

4.3 Channel Inference

We define now a refined type system, allowing the inference of communication channels. These channels will be used for the projection, and have been omitted in the first type system for the sake of clarity.

A channel is a location pair associated with a name, noted $A_1 \overset{n}{\mapsto} A_2$: n is the name of the channel, A_1 its source location, and A_2 its destination location. The set of channel names is ordered, so as to keep consistence of inputs and outputs added, from the node definition to node instances. T denotes sets of channels. The union of sets of channels, noted $T_1 \uplus T_2$, is defined iff channel names in T_1 and T_2 are disjoint.

A channel can be renamed: we note by $s \overset{r(n)}{\mapsto} s'$ the channel $s \overset{n}{\mapsto} s'$ renamed by the function r. Given two channel sets T and T', we note $T' \cong T$ the fact that T' is equal to T modulo a monotone renaming of its channels (i.e., the order between these channels is the same in T and T'):

$$T' \cong T \Leftrightarrow \exists r \text{ s.t. } r \text{ monotone and } T' = \{s \overset{r(n)}{\mapsto} s' | s \overset{n}{\mapsto} s' \in T\}$$

The Figures 8 and 9 shows our refined type system: $H|G \vdash e : t/\ell/T$ means that, in the typing environment H, the expression e has spacial type t, and its computation involves the set of locations ℓ and the set of communication channels T. Channels are added for each communication (rule Comm-C), and for application of nodes which need internal channels (rule App-C). For applications, we have to rename channels, with a renaming preserving the name order. Thus, we can instanciate nodes several times, adding each time new channel names as inputs and outputs

when the projection will be performed. The other rules only perform the disjoint unions (w.r.t. channel names) of sets of channels. The rules defining the program types and the architecture remain the same and are not repeated.

$$\text{(Imm-C)}\quad H|G \vdash i : c \texttt{ at } s/\{s\}/\emptyset \qquad\qquad \text{(Inst-C)}\quad \frac{(t, C) \leq (H(x)) \qquad \mathcal{L} \models C}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash x : t/\text{locations}(t)/\emptyset}$$

$$\text{(Pair-C)}\quad \frac{H|G \vdash e_1 : t_1/\ell_1/T_1 \qquad H|G \vdash e_2 : t_2/\ell_2/T_2}{H|G \vdash (e_1, e_2) : t_1 \times t_2/\ell_1 \cup \ell_2/T_1 \uplus T_2}$$

$$\text{(At-C)}\quad \frac{H \texttt{ at } A|G \vdash e : t/\ell/T}{H|G \vdash e \texttt{ at } A : t/\ell/T} \qquad\qquad \text{(Comm-C)}\quad \frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } s/\ell/T \qquad \mathcal{L} \models s \triangleright s'}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } s'/\ell \cup \{s'\}/T \uplus \{s \overset{n}{\mapsto} s'\}}$$

Figure 8: Refined type system, with communication channels (expressions)

4.4 Projection

We will now define a type-directed operation of *projection of an expression on a location A*. This operation is defined separately, as it has to be performed on an already annotated program: links between values of each projected program are defined by the channels inferred by the type system.

The projection of a declaration D on a location A is noted $H|G \vdash D : H'/\ell/T \overset{A}{\Longrightarrow} D'$, and results in a new declaration D', containing only the computations to be performed on A. The projection of an expression e, of spacial type t, on a location A, is noted $H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} e'/D$, and results in a new expression e', as well as a declaration D, containing channels outputs to be defined. A channel named n in an environment channel will be introduced as the variable c_n as input or output of the target program, c assumed to be of different name space than other variables of the source program.

We denote by ϵ the empty declaration, and by () a value which will never be used (i.e., void). For any declaration D, D `and` $\epsilon = \epsilon$ `and` $D = D$. For any expression e, we have $((\,)\ e) = (e\ (\,)) = (\,)$.

(Node-C)

$$\frac{\begin{array}{c}H, x : t, H_1 | \langle \mathcal{S}', \mathcal{L}' \rangle \vdash D : H_1/\ell_1/T_1 \\ H, x : t, H_1 | \langle \mathcal{S}', \mathcal{L}' \rangle \vdash e : t/\ell_2/T_2 \qquad \mathcal{S}' = \mathcal{S} \cup \{\delta_1, \ldots, \delta_n\} \\ \mathcal{L}' \subseteq \mathcal{L} \cup (\{\delta_1, \ldots, \delta_n\} \times \mathcal{S}) \cup (\mathcal{S} \times \{\delta_1, \ldots, \delta_n\}) \qquad \{\alpha_1, \ldots, \alpha_m\} = \mathrm{FTV}(t) - \mathrm{FTV}(H) \\ C = \mathrm{constr}(\mathcal{L}' \setminus \mathcal{L}) \qquad \sigma = \forall \alpha_1, \ldots, \alpha_m . \forall \delta_1, \ldots, \delta_n : C . t \multimap \langle \ell_1 \cup \ell_2 / T_1 \uplus T_2 \rangle \rightarrowtail t_1\end{array}}{H | G \vdash \texttt{node}\ f[\delta_1, \ldots, \delta_n](x) = e\ \texttt{with}\ D : [\sigma/f]/\ell_1 \cup \ell_2/\emptyset}$$

(Def-C)

$$\frac{H | G \vdash e : t/\ell/T}{H | G \vdash x = e : [t/x]/\ell/T}$$

(App-C)

$$\frac{H | G \vdash f : t_1 \multimap \langle \ell_1/T_1 \rangle \rightarrowtail t_2/\ell_2/T_2 \qquad H | G \vdash e : t_1/\ell_3/T_3 \qquad T_1' \cong T_1}{H | G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T_1' \uplus T_2 \uplus T_3}$$

(And-C)

$$\frac{H | G \vdash D_1 : H_1/\ell_1/T_1 \qquad H | G \vdash D_2 : H_2/\ell_2/T_2}{H | G \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1 \uplus T_2}$$

(Let-C)

$$\frac{H | G \vdash D_1 : H_1/\ell_1/T_1 \qquad H, H_1 | G \vdash D_2 : H_2/\ell_2/T_2}{H | G \vdash \texttt{let}\ D_1\ \texttt{in}\ D_2 : H_2/\ell_1 \cup \ell_2/T_1 \uplus T_2}$$

(If-C)

$$\frac{\begin{array}{c}H | G \vdash e : c\ \texttt{at}\ s/\ell/T \\ H | G \vdash D_1 : H'/\ell_1/T_1 \qquad H | G \vdash D_2 : H'/\ell_2/T_2 \qquad C = \{s \triangleright s' | s' \in \ell_1 \cup \ell_2\} \qquad \mathcal{L} \models C\end{array}}{H | G \vdash \texttt{if}\ e\ \texttt{then}\ D_1\ \texttt{else}\ D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T \uplus T_1 \uplus T_2 \uplus \mathrm{channels}(C)}$$

Figure 9: Refined type system, with communication channels (equations)

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

$$T \uparrow A = \{A \overset{n}{\mapsto} A' \in T\}$$

$$T \downarrow A = \{A' \overset{n}{\mapsto} A \in T\}$$

The projection rules are given in Figures 10 and 11.

Channels are used at communication points. If an expression e is send from A to A' through the channel $A \overset{n}{\mapsto} A'$, 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_n as the result of the projection of e on A (rule Comm-P-From);
- for the projection on A', the communication involves receiving a value: the resulting expression is the channel holding this value (rule 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 Suppr-P). Projections of a pair consist in the projection of its compounds (rule Pair-P).

The projection of a located declaration and parallel declarations involves the projection of its compound (rules At-P and 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 App-P and 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 If-P-From);
- 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 If-P-To).

Lemma 3 states that a program accepted by the spacial type system can always be projected on every location of the architecture.

Lemma 3. *For all* H, H', D, ℓ, T*, if* $H|G \vdash D : H'/\ell/T$ *then* $\forall A \in \mathcal{S}, \exists D'$ *s.t.* $H|G \vdash D : H'/\ell/T \stackrel{A}{\Longrightarrow} D'$.

Proof. Since the projection operation is type-directed, a projection rule can always be applied from the equivalent one of the type system. The property trivially holds. □

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

(Imm-P)
$$H|G \vdash i : c \texttt{ at } s/\{s\}/\emptyset \overset{A}{\Longrightarrow} i/\epsilon$$

(Inst-P)
$$\frac{(t, C) \leq (H(x)) \qquad \mathcal{L} \models C}{H|G \vdash x : t/\ell/\emptyset \overset{A}{\Longrightarrow} x_A/\epsilon}$$

(Suppr-P)
$$\frac{A \notin \ell}{H|G \vdash e : t/\ell/T \overset{A}{\Longrightarrow} ()/\epsilon}$$

(Comm-P-From)
$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } A/\ell/T \overset{A}{\Longrightarrow} e'/D \qquad \mathcal{L} \models A \triangleright s'}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } s'/\ell \cup \{s'\}/T \uplus \{A \overset{n}{\mapsto} s'\} \overset{A}{\Longrightarrow} ()/D \texttt{ and } c_n = e'}$$

(Comm-P-To)
$$\frac{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } s/\ell/T \overset{A'}{\Longrightarrow} e'/D \qquad \mathcal{L} \models s \triangleright A'}{H|\langle \mathcal{S}, \mathcal{L}\rangle \vdash e : t \texttt{ at } A'/\ell \cup \{A'\}/T \uplus \{s \overset{n}{\mapsto} A'\} \overset{A'}{\Longrightarrow} c_n/D}$$

(Pair-P)
$$\frac{H|G \vdash e_1 : t_1/\ell_1/T_1 \overset{A}{\Longrightarrow} e_1'/D_1 \qquad H|G \vdash e_2 : t_2/\ell_2/T_2 \overset{A}{\Longrightarrow} e_2'/D_2}{H|G \vdash e_1, e_2 : t_1 \times t_2/\ell_1 \cup \ell_2/T_1 \uplus T_2 \overset{A}{\Longrightarrow} e_1', e_2'/D_1 \texttt{ and } D_2}$$

Figure 10: Rules for the projection operation (expressions)

(Node-P)

$$\frac{\begin{array}{c}H, x : t, H_1 | G \vdash D : H_1/\ell_1/T_1 \stackrel{A}{\Longrightarrow} D' \\ H, x_t, H_1 | G \vdash e : t/\ell_2/T_2 \stackrel{A}{\Longrightarrow} e'/D_e \qquad T_1 \uplus T_2 \uparrow A = \{A \stackrel{n_1}{\mapsto} A_1, \ldots, A \stackrel{n_p}{\mapsto} A_p\} \\ T_1 \uplus T_2 \downarrow A = \{A \stackrel{m_1}{\mapsto} A_1, \ldots, A \stackrel{m_q}{\mapsto} A_q\} \qquad n_i < n_{i+1} \qquad m_i < m_{i+1}\end{array}}{\begin{array}{c}H | G \vdash \texttt{node}\ f(x) = e\ \texttt{with}\ D : [\mathrm{gen}_H(t \multimap\langle \ell_1/T_1 \uplus T_2\rangle\rightarrow t_1)/f]/\ell_1 \cup \ell_2/\emptyset \\ \stackrel{A}{\Longrightarrow} \texttt{node}\ f_A(x, c_{m_1}, \ldots, c_{m_q}) = (e', c_{n_1}, \ldots, c_{n_p})\ \texttt{with}\ D'\ \texttt{and}\ D_e\end{array}}$$

(At-P)

$$\frac{H\ \texttt{at}\ A | G \vdash D : H'/\ell/T \stackrel{A}{\Longrightarrow} D'}{H | G \vdash D\ \texttt{at}\ A : H'/\ell/T \stackrel{A}{\Longrightarrow} D'}$$

(Def-P)

$$\frac{H | G \vdash e : t/\ell/T \stackrel{A}{\Longrightarrow} e'/D}{H | G \vdash x = e : [t/x]/\ell/T \stackrel{A}{\Longrightarrow} x_A = e'\ \texttt{and}\ D}$$

(App-P)

$$\frac{\begin{array}{c}H | G \vdash f : t_1 \multimap\langle \ell_1/T_1\rangle\rightarrow t_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} f'/D_1 \\ H | G \vdash e : t_1/\ell_3/T_3 \stackrel{A}{\Longrightarrow} e'/D_2 \qquad T_1' \cong T_1 \qquad T_1' \uparrow A = \{A \stackrel{n_1}{\mapsto} A_1, \ldots, A \stackrel{n_p}{\mapsto} A_p\} \\ T_1' \downarrow A = \{A_1 \stackrel{m_1}{\mapsto} A, \ldots, A_q \stackrel{m_q}{\mapsto} A\} \qquad n_i < n_{i+1} \text{ and } m_i < m_{i+1}\end{array}}{\begin{array}{c}H | G \vdash x = f(e) : [t_2/x]/\ell_1 \cup \ell_2 \cup \ell_3/T_1' \uplus T_2 \uplus T_3 \\ \stackrel{A}{\Longrightarrow} (x, c_{m_1}, \ldots, c_{m_q}) = f'(e', c_{n_1}, \ldots, c_{n_p})\ \texttt{and}\ D_1\ \texttt{and}\ D_2\end{array}}$$

(And-P)

$$\frac{H | G \vdash D_1 : H_1/\ell_1/T_1 \stackrel{A}{\Longrightarrow} D_1' \qquad H | G \vdash D_2 : H_2/\ell_2/T_2 \stackrel{A}{\Longrightarrow} D_2'}{H | G \vdash D_1\ \texttt{and}\ D_2 : H_1, H_2/\ell_1 \cup \ell_2/T_1 \uplus T_2 \stackrel{A}{\Longrightarrow} D_1'\ \texttt{and}\ D_2'}$$

(If-P-From)

$$\frac{\begin{array}{c}H | G \vdash e : c\ \texttt{at}\ s/\ell/T \stackrel{A}{\Longrightarrow} e'/D \qquad H | G \vdash D_1 : H'/\ell_1/T_1 \stackrel{A}{\Longrightarrow} D_1' \\ H | G \vdash D_2 : H'/\ell_2/T_2 \stackrel{A}{\Longrightarrow} D_2' \qquad C = \{s \triangleright s' | s' \in \ell_1 \cup \ell_2\} \\ \mathcal{L} \models C \qquad T' = \mathrm{channels}(C) \qquad T' \uparrow A = \{A \stackrel{n_1}{\mapsto} A_1, \ldots, A \stackrel{n_p}{\mapsto} A_p\} \qquad x \notin \mathrm{dom}(H)\end{array}}{\begin{array}{c}H | G \vdash \texttt{if}\ e\ \texttt{then}\ D_1\ \texttt{else}\ D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T \uplus T_1 \uplus T_2 \uplus \mathrm{channels}(C) \\ \stackrel{A}{\Longrightarrow} x = e'\ \texttt{and}\ c_{n_1} = x\ \texttt{and}\ \ldots\ \texttt{and}\ c_{n_p} = x\ \texttt{and if}\ x\ \texttt{then}\ D_1'\ \texttt{else}\ D_2'\end{array}}$$

(If-P-To)

$$\frac{\begin{array}{c}H | G \vdash e : c\ \texttt{at}\ s/\ell/T \stackrel{A}{\Longrightarrow} e'/D \\ H | G \vdash D_1 : H'/\ell_1/T_1 \stackrel{A}{\Longrightarrow} D_1' \qquad H | G \vdash D_2 : H'/\ell_2/T_2 \stackrel{A}{\Longrightarrow} D_2' \\ C = \{s \triangleright s' | s' \in \ell_1 \cup \ell_2\} \qquad \mathcal{L} \models C \qquad T' = \mathrm{channels}(C) \qquad T' \downarrow A = \{A' \stackrel{n}{\mapsto} A\}\end{array}}{\begin{array}{c}H | G \vdash \texttt{if}\ e\ \texttt{then}\ D_1\ \texttt{else}\ D_2 : H'/\ell \cup \ell_1 \cup \ell_2/T \uplus T_1 \uplus T_2 \uplus \mathrm{channels}(C) \\ \stackrel{A}{\Longrightarrow} \texttt{if}\ c_n\ \texttt{then}\ D_1'\ \texttt{else}\ D_2'\end{array}}$$

Figure 11: Rules for the projection operation (equations)

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 D, projected on the locations $\mathcal{S}$, is then defined by:

$$D_1 \texttt{ and } \ldots \texttt{ and } D_n$$
$$\text{where } \forall i, H|G \vdash D : H'/\ell/T \overset{A_i}{\Longrightarrow} D_i$$

In order to relate a target program with its source, we define a relation on values, denoted $\cdot \preccurlyeq^{\cdot}_{\cdot} \cdot$, such that $v' \preccurlyeq^A_t v$ means that the value v', emitted from an expression of type t, represents the value v at the location A. We have:

$$\frac{v' = v}{v' \preccurlyeq^A_{t \texttt{ at } A} v} \qquad \frac{A \neq A'}{() \preccurlyeq^A_{t \texttt{ at } A'} v} \qquad \frac{v'_1 \preccurlyeq^A_{t_1} v_1 \qquad v'_2 \preccurlyeq^A_{t_2} v_2}{(v'_1, v'_2) \preccurlyeq^A_{t_1 \times t_2} (v_1, v_2)}$$

We can then relate two reaction environments R and R_p w.r.t. a typing environment H:

$$R \overset{H}{\cong} R_p \text{ iff } \forall x \in \mathrm{dom}(R), \forall A \in \mathcal{S}, R(x) \preccurlyeq^A_{H(x)} R_p(x_A)$$

Theorem 3 states that the projection operation is correct, i.e., the projection of a source program D into D_i (for every location A_i) defines a new target program D_1 `and` $\ldots$ `and` D_n, which is semantically equivalent, taking into account spacial types' values, with the source declaration D.

Theorem 3 (Correction of the declarations projection)**.** *For all H, H', D, D', ℓ, T, D_i, R, R', if $R \vdash D \xrightarrow{R'} D'$, $H|G \vdash D : H'/\ell/T$ and $\forall i, H|G \vdash D : H'/\ell/T \overset{A_i}{\Longrightarrow} D_i$, then there exists R_p, R'_p, D_p, D'_p such that $D_p = D_1$ `and` $\ldots$ `and` D_n, $R \overset{H}{\cong} R_p$, $R_p \vdash D_p \xrightarrow{R'_p} D'_p$, and $R' \overset{H'}{\cong} R'_p$.*

5 Discussion

5.1 Related Work

Various solutions have emerged in order to use synchronous languages for the design of distributed systems [12]. Some of them operate on a compiled model of the program, by "coloring" atomic instructions with localization informations, inferred from inputs and outputs locations [9]. 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 [4] as well as LUSTRE [8]. In both cases, the correction of the distribution algorithm has been proved [7, 15], 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 spacial 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 can be given to our language, the semantical 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 [17], 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. Type systems have been used to ensure memory consistency [19], or for pointer analysis within a distributed architecture [14]. The ACUTE language [18] 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. Finally, Oz and its distributed extension [5] 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. This language 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.

5.2 Conclusion and Future Work

We have proposed a spacial 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 points in the program communication primitives must be inserted. The compilation of a correctly spacially 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 the compiler of Lucid Synchrone [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.

Acknowledgments. We gratefully thank Jean-Pierre Talpin for constructive discussions.

References

[1] Lucid synchrone v3. `http://www.lri.fr/~pouzet/lucid-synchrone`.

[2] Architecture analysis & design language (AADL). SAE Standard (AS5506), Nov. 2004.

[3] C. André, F. Boulanger, and A. Girault. Software implementation of synchronous programs. In *International Conference on Application of Concurrency to System Design, ICACSD'01*, pages 133–142, Newcastle, UK, June 2001. IEEE.

[4] P. Aubry and P. Le Guernic. On the desynchronization of synchronous applications. In *11th International Conference on Systems Engineering, ICSE'96*, Las Vegas, USA, June 1996.

[5] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems*, 91(1):64–83, Jan. 2003.

[7] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing automata for asynchronous networks of processors. *European Journal of Automated Systems*, 31(3):503–524, 1997. Research report Inria 2341.

[8] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/Lustre to TTA: A layered approach for distributed embedded applications. In *International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES'03*, pages 153–162, San Diego, USA, June 2003. ACM.

[9] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering*, 25(3):416–427, May 1999.

[10] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[11] P. Cuoq and M. Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[12] A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, Apr. 2005. Elsevier Science.

[13] F. Jondral. Software-defined radio — basics and evolution to cognitive radio. *EURASIP Journal on Wireless Communications and Networking*, 3:275–283, 2005.

[14] B. Liblit and A. Aiken. Type systems for distributed data structures. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 199–213, New York, NY, USA, 2000. ACM Press.

[15] O. Maffeïs. *Ordonnancements de graphes de flots synchrones ; Application à la mise en œuvre de Signal*. Phd thesis, University of Rennes I, Rennes, France, Jan. 1993.

[16] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5):26–38, May 1995.

[17] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, New York, NY, USA, 2005. ACM Press.

[18] P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of ICFP 2005: International Conference on Functional Programming (Tallinn)*, Sept. 2005.

[19] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992.

A Proofs

A.1 Distributed and centralized semantics

(Lemma 2). The proof lies in the extension of the $|\cdot|$ operator to the predicates $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$ and $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$. For each rule of the distributed semantics, we can replace the predicate $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$ by $|\hat{R}| \vdash e \xrightarrow{|\hat{v}|} e'$ and $\hat{R} \overset{\ell}{\Vdash} D \xrightarrow{\hat{R}'} D'$ by $|\hat{R}| \vdash D \xrightarrow{|\hat{R}'|} D'$ to obtain an equivalent centralized semantics. □

A.2 Type system correction

The proof of the theorem 1 is based upon the following lemmas.

The first states that the set of locations needed for the computation of an expression typed from a typing environment located on A is the singleton $\{A\}$.

Lemma 4. *For all H, A, G, D, H', ℓ, if H $\texttt{at}$ $A|G \vdash D : H'/\ell$ then $\ell = \{A\}$.*

Proof. From the definition of the instantiation, for all H, A, x, C, t such that $(t, C) \leq (H \texttt{ at } A)(x)$, $\exists t'$, such that $t = t' \texttt{ at } A$. Thus locations$(t) = \{A\}$.

The lemma is then proved by induction on the structure of e. The other rules only perform union of location sets from the spacial types of compounds of e, thus the property holds. □

The following lemma is the equivalent of the theorem 1 for expressions:

Lemma 5 (Correction for expressions)**.** *For all $e, e', H, G, t, \ell, R, v$, if $H|G \vdash e : t/\ell$ and $R \vdash e \xrightarrow{v} e'$, then there exists $\hat{R}, \hat{v}$ such that $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$, $\hat{R} : H$, $\hat{v} : t$, $|\hat{R}| = R$ and $|\hat{v}| = v$.*

Proof. By induction on the structure of the typing derivation tree of e. For each applied rule of the type system, we can apply a matching rule of the distributed semantics.

The induction operates on the last applied rule of the type system.
Case of rule IMM: $e = i$. Let s such that $H|G \vdash i : c \texttt{ at } s/\{s\}$. For any $R, \hat{R}$ such that $|\hat{R}| = R$, $R \vdash i \xrightarrow{i} i \Rightarrow \hat{R} \overset{\{s\}}{\Vdash} i \xrightarrow{i \texttt{ at } s} i$ and $i \texttt{ at } s : c \texttt{ at } s$.
Case of rule INST: $e = x$. Let $H, \mathcal{L}, \mathcal{S}, t, \ell$ such that $H|\langle \mathcal{L}, \mathcal{S} \rangle \vdash e : t/\ell$. Let R, v, e' such that $R \vdash e \xrightarrow{v} e'$.

- Case $|\ell| > 1$. For all $\hat{R}$ such that $|\hat{R}| = R$ and $\hat{R} : H$, then there exists dv such that $\hat{R}(x) = dv \texttt{ at } \top$. Then the rule INST applies: $\hat{R}, [dv \texttt{ at } \top/x] \overset{\ell}{\Vdash} x \xrightarrow{dv \texttt{ at } \top} x$. Moreover, $|dv \texttt{ at } \top| = v$ since $|\hat{R}| = R$, and $dv \texttt{ at } \top : t$ since $\hat{R} : H$.
- Case $|\ell| = 1$ ($\ell = \{s\} = $ locations(t)). For all $\hat{R}$ such that $|\hat{R}| = R$ and $\hat{R} : H$, then there exists dv such that $\hat{R}(x) = dv \texttt{ at } s$. The rule INST applies: $\hat{R}, [dv \texttt{ at } s/x] \overset{\{s\}}{\Vdash} x \xrightarrow{dv \texttt{ at } s} x$.

Case of rule PAIR: $e = (e_1, e_2)$.

- $H|G \vdash e : t/\ell$ iff $\exists \ell_1, \ell_2, t_1, t_2$ such that $H|G \vdash e_1 : t_1/\ell_1$, $H|G \vdash e_2 : t_2/\ell_2$, $t = t_1 \times t_2$ and $\ell = \ell_1 \cup \ell_2$.
- $R \vdash (e_1, e_2) \xrightarrow{v} e'$ iff $\exists v_1, v_2, e'_1, e'_2$ such that $v = (v_1, v_2)$, $e' = (e'_1, e'_2)$, $R \vdash e_1 \xrightarrow{v_1} e'_1$ and $R \vdash e_2 \xrightarrow{v_2} e'_2$.
- By induction hypothesis, there exists $\hat{R}, \hat{v}_1, \hat{v}_2$ such that $\hat{R} \overset{\ell_1}{\Vdash} e_1 \xrightarrow{\hat{v}_1} e'_1$, $\hat{R} \overset{\ell_2}{\Vdash} e_2 \xrightarrow{\hat{v}_2} e'_2$, $\hat{v}_1 : t_1$ and $\hat{v}_2 : t_2$. Let $\hat{v}_1 = dv_1$ `at` s_1 and $\hat{v}_2 = dv_2$ `at` s_2.
- Then, rule Pair applies: $\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} (e_1, e_2) \xrightarrow{(\hat{v}_1, \hat{v}_2)\ \texttt{at}\ s_1 \sqcup s_2} (e'_1, e'_2)$, $\overline{\ell_1 \cup \ell_2} = s_1 \sqcup s_2$ and $((\hat{v}_1, \hat{v}_2)\ \texttt{at}\ s_1 \sqcup s_2) : (t_1 \times t_2)$.

Case of rule Fst: $e =$ `fst` e_1.

- From the definition of H_0, $H|G \vdash e : t/\ell$ iff $\exists s_1, s_2, t_1, t_2$ such that $H|G \vdash e_1 : (t_1\ \texttt{at}\ s_1, t_2\ \texttt{at}\ s_2)/\{s_1, s_2\}$.
- From rule Fst of the centralized semantics, $R \vdash \texttt{fst}\ e_1 \xrightarrow{v_1} \texttt{fst}\ e'_1$ iff $\exists v_2, R \vdash e_1 \xrightarrow{(v_1, v_2)} e'_1$.
- By induction hypothesis, there exists $\hat{R}, \hat{v}$ such that $\hat{R} \overset{\{s_1, s_2\}}{\Vdash} e_1 \xrightarrow{\hat{v}} e'_1$. $|\hat{v}| = (v_1, v_2)$ then $\exists \hat{v}_1, \hat{v}_2, s'$ such that $\hat{v} = (\hat{v}_1, \hat{v}_2)\ \texttt{at}\ s'$.
- Applying rule Fst: $\hat{R} \overset{\{s_1, s_2\}}{\Vdash} \texttt{fst}\ e_1 \xrightarrow{\hat{v}_1} \texttt{fst}\ e'_1$. $\hat{v} : t_1 \times t_2$, then $\hat{v}_1 : t_1$.

Case of rule Snd (same as rule Fst): $e =$ `snd` e_1.

- From the definition of H_0, $H|G \vdash e : t/\ell$ iff $\exists s_1, s_2, t_1, t_2$ such that $H|G \vdash e_1 : (t_1\ \texttt{at}\ s_1, t_2\ \texttt{at}\ s_2)/\{s_1, s_2\}$.
- From rule Snd of the centralized semantics, $R \vdash \texttt{snd}\ e_2 \xrightarrow{v_2} \texttt{snd}\ e'_2$ iff $\exists v_1, R \vdash e_2 \xrightarrow{(v_1, v_2)} e'_2$.
- By induction hypothesis, there exists $\hat{R}, \hat{v}, s$ such that $\hat{R} \overset{\{s_1, s_2\}}{\Vdash} e_2 \xrightarrow{\hat{v}} e'_2$. $|\hat{v}| = (v_1, v_2)$ then $\exists \hat{v}_1, \hat{v}_2, s'$ such that $\hat{v} = (\hat{v}_1, \hat{v}_2)\ \texttt{at}\ s'$.
- Applying rule Snd: $\hat{R} \overset{\{s_1, s_2\}}{\Vdash} \texttt{snd}\ e_1 \xrightarrow{\hat{v}_1} \texttt{snd}\ e'_1$. $\hat{v} : t_1 \times t_2$, then $\hat{v}_2 : t_2$.

Case of rule FBY: $e = e_1 \texttt{ fby } e_2$.

- From the definition of H_0, $H|G \vdash e : t/\ell$ iff $H|G \vdash e_1 : t/\ell$ and $H|G \vdash e_2 : t/\ell$.
- From rule FBY of the centralized semantics, $R \vdash e_1 \texttt{ fby } e_2 \xrightarrow{v} e$ iff $\exists v_1, v_2, e'_1, e'_2$ such that $v = v_1$, $e' = v_2 \texttt{ fby } e'_2$, $R \vdash e_1 \xrightarrow{v_1} e'_1$ and $R \vdash e_2 \xrightarrow{v_2} e'_2$.
- By induction hypothesis, there exists $\hat{R}, \hat{v}_1, \hat{v}_2$ such that $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{\hat{v}_1} e'_1$, $\hat{R} \overset{\ell}{\Vdash} e_2 \xrightarrow{\hat{v}_2} e'_2$, $|\hat{v}_1| = v_1$, $|\hat{v}_2| = v_2$, $\hat{v}_1 : t$ and $\hat{v}_2 : t$.
- Applying rule FBY: $\hat{R} \overset{\ell}{\Vdash} e_1 \texttt{ fby } e_2 \xrightarrow{\hat{v}_1} |\hat{v}_2| \texttt{ fby } e'_2$.

Case of rule AT: $e = e_1 \texttt{ at } s$.

- $H|G \vdash e_1 \texttt{ at } s : t/\ell$ iff $H \texttt{ at } s|G \vdash e_1 : t/\ell$.
- $R \vdash e_1 \texttt{ at } s \xrightarrow{v} e'$ iff $\exists e'_1$, $e' = e'_1 \texttt{ at } s$ and $R \vdash e_1 \xrightarrow{v} e'_1$.
- By induction hypothesis, there exists $\hat{R}, \hat{v}, s$ such that $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{\hat{v}} e'_1$, $|\hat{R}| = R$, $|\hat{v}| = v$, $\hat{R} : H$, $\hat{v} : t$.
- From lemma 4, $\ell = \{s\}$.
- Applying rule AT: $\hat{R} \overset{\{s\}}{\Vdash} e_1 \texttt{ at } s \xrightarrow{\hat{v}} e'_1 \texttt{ at } s$.

Case of rule COMM.

- $H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e : t \texttt{ at } s'/\ell$ iff there exists ℓ' such that $\ell = \ell' \cup \{s\}$, $H|\langle \mathcal{S}, \mathcal{L} \rangle \vdash e : t \texttt{ at } s/\ell'$ and $\mathcal{L} \models s \triangleright s'$.
- $R \vdash e \xrightarrow{v} e'$.
- By induction hypothesis, there exists $\hat{R}, \hat{v}, s$ such that $\hat{R} \overset{\ell}{\Vdash} e_1 \xrightarrow{\hat{v}} e'_1$, $|\hat{R}| = R$, $|\hat{v}| = v$, $\hat{R} : H$, $\hat{v} : t \texttt{ at } s$.
- Applying rule COMM: $\hat{R} \overset{\ell \cup \{s\}}{\Vdash} e_1 \texttt{ at } s \xrightarrow{\hat{R}'} e'_1 \texttt{ at } s$.

□

(Theorem 1). By induction on the structure of D.
Case $D = (x = e)$.

- $H|G \vdash x = e : H'/\ell$ iff $\exists s, t$, $H' = [\forall \delta : \{s \triangleright \delta\}.t \texttt{ at } \delta/x]$ and $H|G \vdash e : t \texttt{ at } s/\ell$.
- $R \vdash x = e \xrightarrow{R'} D'$ iff $\exists e', v$, $D' = (x = e')$, $R' = [v/x]$, and $R \vdash e \xrightarrow{v} e'$.
- By application of lemma 5, $\exists \hat{R}, \hat{v}$ such that $\hat{R} \overset{\ell}{\Vdash} e \xrightarrow{\hat{v}} e'$, $|\hat{R}| = R$, $\hat{R} : H$, $|\hat{v}| = v$, $\hat{v} : t \texttt{ at } s$.
- Applying rule DEF: $\hat{R} \overset{\ell}{\Vdash} x = e \xrightarrow{[\hat{v}/x]} x = e'$. $|[\hat{v}/x]| = [v/x] = R$, $[\hat{v}/x] : [\forall \delta : \{s \triangleright \delta\}.t \texttt{ at } \delta/x]$ since $\hat{v} : t \texttt{ at } s$.

Case $D = (x = f(e))$.

- $H|G \vdash x = f(e) : H'/\ell$ iff $\exists s, t_1, t_2, \ell_1, \ell_2, \ell_3$ such that $\ell = \ell_1 \cup \ell_2 \cup \ell_3$, $H' = [t_2/x]$, $H|G \vdash f : t_1 \xrightarrow{\langle \ell_1 \rangle} t_2/\ell_2$ and $H|G \vdash e : t_1/\ell_3$.
- $R \vdash x = f(e) \xrightarrow{R'} D'$ iff $\exists e_1, D_1$ such that $R(f) = \lambda y.e_1 \texttt{ with } D_1$ and $R \vdash x = e_1 \texttt{ and } y = e \texttt{ and } D_1 \xrightarrow{R'} D'$.
- $H|G \vdash f : t_1 \xrightarrow{\langle \ell_1 \rangle} t_2/\ell_2$ iff $\exists H_1, \ell_{11}, \ell_{12}$ such that $\ell_1 = \ell_{11} \cup \ell_{12}$, $H.[y : t_1]|G \vdash D_1 : H_1/\ell_{11}$ and $H.[y : t_1].H_1|G \vdash e_1 : t_2/\ell_{12}$.
- By induction hypothesis, $\exists \hat{R}, \hat{R}'$ such that $\hat{R} \overset{\ell}{\Vdash} x = e_1 \texttt{ and } y = e \texttt{ and } D_1 \xrightarrow{\hat{R}'} D'$, $|\hat{R}| = R$, $|\hat{R}'| = \hat{R}'$, $\hat{R} : H$, $\hat{R}' : H'$.
- Applying rule APP: $\hat{R} \overset{\ell}{\Vdash} x = f(e) \xrightarrow{\hat{R}'} D'$.
- $\hat{R} \overset{\ell}{\Vdash} x = e_1 \texttt{ and } y = e \texttt{ and } D_1 \xrightarrow{\hat{R}'} D'$ iff there exists $\hat{R}_{11}, \hat{v}_{12}, \hat{v}_3, e', D'_1, e'_1$ such that
  - $\hat{R}.[\hat{v}_{12}/x, \hat{v}_3/y] \overset{\ell_{12}}{\Vdash} D_1 \xrightarrow{\hat{R}_{11}} D'_1$,
  - $\hat{R}.\hat{R}_{11}.[\hat{v}_3/y] \overset{\ell_{12}}{\Vdash} x = e_1 \xrightarrow{[\hat{v}_{12}/x]} x = e'_1$,
  - $\hat{R}.\hat{R}_{11}.[\hat{v}_{12}/x] \overset{\ell_3}{\Vdash} y = e \xrightarrow{[\hat{v}_3/y]} y = e'$.
- By induction, $\ell'_{11} = \ell_{11}$, $\ell'_{12} = \ell_{12}$, and $\ell'_3 = \ell_3$. Then $\ell' = \ell_1 \cup \ell_3$.
- From rule INST of spacial type system, $H|G \vdash f : t_1 \xrightarrow{\langle \ell_1 \rangle} t_2/\ell_2$ iff $\ell_2 = \text{locations}(t_1 \xrightarrow{\langle \ell_1 \rangle} t_2) = \ell_1$.

- Then, $\ell = \ell_1 \cup \ell_3 = \ell'$.

Case $D = D_1$ `and` D_2.

- $H|G \vdash D_1$ `and` $D_2 : H'/\ell$ iff $\exists H_1, H_2, \ell_1, \ell_2$ such that $\ell = \ell_1 \cup \ell_2$, $H|G \vdash D_1 : H_1/\ell_1$ and $H|G \vdash D_2 : H_2/\ell_2$.
- $R \vdash D_1$ `and` $D_2 \xrightarrow{R'} D'$ iff $\exists R_1, R_2, D'_1, D'_2$ such that $R.R_2 \vdash D_1 \xrightarrow{R_1} D'_1$ and $R.R_1 \vdash D_2 \xrightarrow{R_2} D'_2$.
- By induction hypothesis, $\exists \hat{R}, \hat{R}_1, \hat{R}_2$ such that
  - $\hat{R}.\hat{R}_2 \overset{\ell_1}{\Vdash} D_1 \xrightarrow{\hat{R}_1} D'_1$,
  - $\hat{R}.\hat{R}_1 \overset{\ell_2}{\Vdash} D_2 \xrightarrow{\hat{R}_2} D'_2$,
  - $|\hat{R}| = R$, $|\hat{R}_1| = R_1$, $|\hat{R}_2| = R_2$,
  - $\hat{R} : H$, $\hat{R}_1 : H_1$, $\hat{R}_2 : H_2$.
- Applying rule AND: $\hat{R} \overset{\ell_1 \cup \ell_2}{\Vdash} D_1$ `and` $D_2 \xrightarrow{\hat{R}_1.\hat{R}_2} D'_1$ `and` D'_2. $(\hat{R}_1.\hat{R}_2) : (H_1.H_2)$ and $|\hat{R}_1.\hat{R}_2| = R_1.R_2$.

Case $D =$ `if` e `then` D_1 `else` D_2.

- $H|G \vdash$ `if` e `then` D_1 `else` $D_2 : H'/\ell$ iff there exists s, ℓ', ℓ_1, ℓ_2 such that $\ell = \ell' \cup \ell_1 \cup \ell_2$, $H|G \vdash e : c$ `at` s/ℓ', $H|G \vdash D_1 : H'/\ell_1$, $H|G \vdash D_2 : H'/\ell_2$ and $\forall s' \in \ell_1 \cup \ell_2, s \triangleright s'$.
- $R \vdash$ `if` e `then` D_1 `else` $D_2 \xrightarrow{R'} D'$ iff there exists e' such that $R \vdash e \xrightarrow{v} e'$, with $v =$ `true` or $v =$ `false`.
  - Case $v =$ `true`: there exists D'_1 such that $R \vdash D_1 \xrightarrow{R'} D'_1$ and $D' =$ `if` e' `then` D'_1 `else` D_2.
    - By induction hypothesis, $\exists \hat{R}, \hat{R}'$ such that $\hat{R} \overset{\ell'}{\Vdash} e \xrightarrow{\texttt{true at } s} e'$, $\hat{R} \overset{\ell_1}{\Vdash} D_1 \xrightarrow{\hat{R}'} D'_1$, $|\hat{R}| = R$, $\hat{R} : H$, $|\hat{R}'| = R'$, $\hat{R}' : H'$.
    - Applying rule IF-1: $\hat{R} \overset{\ell' \cup \ell_1}{\Vdash}$ `if` e `then` D_1 `else` $D_2 \xrightarrow{\hat{R}'}$ `if` e' `then` D'_1 `else` D'_2.
  - Case $v =$ `false`: same as case $v =$ `true`.

□

A.3 Projection correction

The proof of the theorem 3 is based on the following lemma, stating that if the projection of an expression e on every location A_i is the couple e_i/D_i, then the semantical relation between e and e_i is established by evaluating e_i in a reaction environment emitted by D_1 `and` ... `and` D_n. This equivalence relation depends upon the spacial type of e.

Lemma 6 (Projection correction (expressions))**.** *For all H, e, v, e', t, ℓ, T, e_i, D_i, R, if $R \vdash e \xrightarrow{v} e'$, $H|G \vdash e : t/\ell/T$ and $\forall i, H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} e_i/D_i$, then for all R_p such that $R \overset{H}{\cong} R_p$, there exists $v_i, e'_i, D_p, D'_p, R'_p$ such that $D_p = D_1$ **`and`** ... **`and`** D_n, $R_p \vdash D_p \xrightarrow{R'_p} D'_p$, $\mathrm{dom}(R'_p) = \{c_n | \exists A, A', A \overset{n}{\mapsto} A' \in T\}$. $R_p, R'_p \vdash e_i \xrightarrow{v_i} e'_i$ and $v_i \preccurlyeq_t^{A_i} v$.*

Proof. By induction on the structure of the typing derivation tree of e.

Let H, e, v, e', t, ℓ, T, e_i, D_i, R, R_p, such $R \vdash e \xrightarrow{v} e'$, $H|G \vdash e : t/\ell/T$ and $\forall i, H|G \vdash e : t/\ell/T \overset{A_i}{\Longrightarrow} e_i/D_i$

Case of rule Imm: $e = i$. Then, $R \vdash i \xrightarrow{i} i$. For all H, G, s, A, $H|G \vdash i : c$ `at` $s/\{s\}/\emptyset \overset{A}{\Longrightarrow} i/\epsilon$. Let $D_p = \epsilon$ and $R'_p = \emptyset$. For all R_p, $R \vdash i \xrightarrow{i} i$ and $i \preccurlyeq_c^A$ `at` s i.

Case of rule Inst: $e = x$. Then, $R \vdash x \xrightarrow{v} x$. $H|G \vdash x : t/\ell/T$, then, $\exists C, \mathcal{S}, \mathcal{L}$ such that $G = \langle \mathcal{S}, \mathcal{L} \rangle$, $(t, C) \leq H(x)$ and $\mathcal{L} \models C$. Then the rule Inst-P applies: for all A, $H|G \vdash x : t/\ell \overset{A}{\Longrightarrow} x_A$. Let $D_p = \epsilon$, $R'_p = \emptyset$ and $v_A = R_p(x_A)$. $R_p \vdash x_A \xrightarrow{v_A} x_A$, and $v_A \preccurlyeq_t^A v$ since $R \overset{H}{\cong} R_p$.

Case of rule Pair: $e = (e_1, e_2)$. By induction, and the fact that $H|G \vdash e : t/\ell/T$ iff $\exists \ell_1, \ell_2, T_1, T_2, t_1, t_2$ such that $\ell = \ell_1 \cup \ell_2$, $T = T_1 \uplus T_2$, $t = t_1 \times t_2$, $H|G \vdash e_1 : t_1/\ell_1/T_1$ and $H|G \vdash e_2 : t_2/\ell_2/T_2$. Then for a site A, $H|G \vdash e_1 : t_1/\ell_1/T_1 \overset{A}{\Longrightarrow} e'_1/D_1$ and $H|G \vdash e_2 : t_2/\ell_2/T_2 \overset{A}{\Longrightarrow} e'_2/D_2$, there exists R_p, R'_{p1}, R'_{p2}, such that $R_p \vdash D_1 \xrightarrow{R'_{p21}} D'_1$, $R_p \vdash D_2 \xrightarrow{R'_{p2}} D'_2$, and $\mathrm{dom}(R'_{p1}) \cap \mathrm{dom}(R'_{p2}) = \emptyset$. Then, taking $R'_p = R'_{p1}.R'_{p2}$, $R_p \vdash D_1$ `and` $D_2 \xrightarrow{R'_p} D'_1$ `and` D'_2, and there exists v such that $R_p, R'_p \vdash (e_1, e_2) \xrightarrow{v} (e'_1, e'_2)$.

Case of rules Fst and Snd: $e =$ `fst` e' and $e =$ `snd` e': direct induction.

Case of rule Comm: $H|G \vdash e : t/\ell/T$ iff $\exists t', A, A', T', n$ such that the architecture allows the communication from A to A', $t = t'$ `at` A', $T = T' \uplus \{A \overset{n}{\mapsto} A'\}$, and $H|G \vdash e : t'$ `at` $A_1/\ell/T$.

By induction hypothesis, for every location A_i, there exists e_i, D_i such that $H|G \vdash e : t'$ `at` $A/\ell/T' \stackrel{A_i}{\Longrightarrow} e_i/D_i$. Let $D'_p = D_1$ `and` $\ldots$ `and` D_n, D''_p and R''_p such that $R'_p \vdash D'_p \xrightarrow{R''_p} D''_p$.

For each i, let v_i, e'_i such that $R_p, R'_p \vdash e_i \xrightarrow{v_i} e'_i$.

- If $A_i = A$, then by induction hypothesis, $v_i \preccurlyeq^{A_i}_{t' \texttt{ at } A} v$. By application of the rule Comm-From, $H|G \vdash e : t'$ `at` $A'/\ell \cup \{A'\}/T \uplus \{A \stackrel{n}{\mapsto} A'\} \stackrel{A_i}{\Longrightarrow} ()/D_i$ `and` $c_n = e_i$. Then, as $A_i \neq A'$, $() \preccurlyeq^{A_i}_{t' \texttt{ at } A'} v$.
- If $A_i = A'$, then by application of the rule Comm-To, $H|G \vdash e : t'$ `at` $A'/\ell \cup \{A'\}/T \uplus \{A \stackrel{n}{\mapsto} A'\} \stackrel{A_i}{\Longrightarrow} c_n/D_i$. From rule Comm-From, $\exists v_p, e_p$ such that $D_p = D'_p$ `and` $c_n = e_p$,

□

Theorem 3. By induction on the structure of D.
Case $D = D_1$ `at` A. Direct induction.
Case $D = (x = e)$.

- $R \vdash D \xrightarrow{R'} D'$ iff there exists e', v, $D' = (x = e')$, $R' = [v/x]$, and $R \vdash e \xrightarrow{v} e'$.
- $H|G \vdash x = e : H'/\ell$ iff $\exists s, t$, $H' = [\forall \delta : \{s \triangleright \delta\}.t$ `at` $\delta/x]$ and $H|G \vdash e : t$ `at` s/ℓ.
- Let $\{A_1, \ldots, A_n\} = \mathcal{S}$. For all i, $H|G \vdash D : H'/\ell/T \stackrel{A_i}{\Longrightarrow} D_i$ iff there exists e'_i, D'_i such that $H|G \vdash e : t$ `at` $s/\ell/T \stackrel{A}{\Longrightarrow} e_i/D'_i$.
- Let R_p such that $R_p \stackrel{H}{\cong} R$. By application of lemma 6, there exists $v_i, e'_i, D_e, D'_e, R'_e$, $D_e = D_1$ `and` $\ldots$ `and` D_n, $R_p \vdash D_e \xrightarrow{R'_e} D'_e$, $R_p, R'_e \vdash e_i \xrightarrow{v_i} e'_i$ and $v_i \preccurlyeq^{A_i}_{t \texttt{ at } s} v$.
- Then, by application of rule Def-P, taking $D_i = (x_{A_i} = e_i)$ `and` D'_i, and $D_p = D_1$ `and` $\ldots$ `and` D_n and $R'_p = R'_e.[v_1/x_{A_1}, \ldots, v_n/x_{A_n}]$, $R_p \vdash D_p \xrightarrow{R'_p} D'_p$.

Case $D = (x = f(e))$.

- $R \vdash D \xrightarrow{R'} D'$ iff there exists e_1, D_1 such that $R(f) = \lambda y.e_1$ `with` D_1 and $R \vdash x = e_1$ `and` $y = e$ `and` $D_1 \xrightarrow{R'} D'$.
- $H|G \vdash D : H'/\ell/T$ iff $\exists t_1, t_2, \ell_1, \ell_2, \ell_3, T_1, T'_1, T_2, T_3$,such that:

- $\ell = \ell_1 \cup \ell_2 \cup \ell_3$,
- $T = T_1' \uplus T_2 \uplus T_3$,
- $T_1 \cong T_1'$,
- $H|G \vdash f : t_1 \multimap\langle \ell_1/T_1 \rangle\rightarrow t_2/\ell_2/T_2$,
- $H|G \vdash e : t_1/\ell_3/T_3$,
- $H' = [t_2/x]$.

• Let $\{A_1, \ldots, A_n\} = \mathcal{S}$. For all i,

- $H|G \vdash D : H'/\ell/T \stackrel{A_i}{\Longrightarrow} D_i$ iff there exists $f', e', D_2, D_3,\ c_{m_1}, \ldots, c_{m_q}$, $c_{n_1}, \ldots, c_{n_p}$ such that
  * $T_1' \uparrow A_i = \{A \stackrel{n_1}{\mapsto} A_{11}, \ldots, A_i \stackrel{n_p}{\mapsto} A_{1p}\}$,
  * $T_1' \downarrow A_i = \{A_{21} \stackrel{m_1}{\mapsto} A, \ldots, A_{2q} \stackrel{m_q}{\mapsto} A_i\}$,
  * $H|G \vdash f : t_1 \multimap\langle \ell_1/T_1 \rangle\rightarrow t_2/\ell_2/T_2 \stackrel{A_i}{\Longrightarrow} f'/D_2$,
  * $H|G \vdash e : t_1/\ell_3/T_3 \stackrel{A_i}{\Longrightarrow} e'/D_3$,
  * $D_i = (x, c_{m_1}, \ldots, c_{m_q}) = f'(e', c_{n_1}, \ldots, c_{n_p})$ `and` D_2 `and` D_3.
- From rules Inst-P-Here and Node-P, $f' = f_{A_i}$ and f_{A_i} is defined by

$$\texttt{node}\ f_{A_i}(x, c_{n_1'}, \ldots, c_{n_p'}) = (e_1', c_{m_1'}, \ldots, c_{m_q'})\ \texttt{with}\ D_1'$$

  Where:
  * $T_1 \uparrow A_i = \{A \stackrel{n_1'}{\mapsto} A_{11}, \ldots, A_i \stackrel{n_p'}{\mapsto} A_{1p}\}$,
  * $T_1 \downarrow A_i = \{A_{21} \stackrel{m_1'}{\mapsto} A, \ldots, A_{2q} \stackrel{m_q'}{\mapsto} A_i\}$,
- By application of lemma 6, there exists $v_i, e_i', D_e, D_e', R_e', D_e = D_1$ `and` $\ldots$ `and` D_n, $R_p \vdash D_e \stackrel{R_e'}{\longrightarrow} D_e'$, $R_p, R_e' \vdash e_i \stackrel{v_i}{\longrightarrow} e_i'$ and $v_i \preccurlyeq^{A_i}_{t\ \texttt{at}\ s} v$.
- From the fact that the renamings of channels are monotone, the application of the semantics on each projection keeps the consistency of the communicated values between each projected node and its projected instantiation.

□

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399