IMT School for Advanced Studies, Lucca
Lucca, Italy

Type discipline for message-passing components in distributed systems

PhD Program in Computer Science and Engineering

XXXIII Cycle

By

Zorica Savanović

2023.

The dissertation of Zorica Savanović is approved.

PhD Program Coordinator: Rocco De Nicola, IMT School for Advanced Studies Lucca, Italy

Advisor: Dr. Letterio Galletta, IMT School for Advanced Studies Lucca, Italy

The dissertation of Zorica Savanović has been reviewed by:

Prof. Carbone Marco, IT University of Copenhagen, Denmark

Prof. Dezani-Ciancaglini Mariangiola, Università di Torino, Italy

IMT School for Advanced Studies Lucca
2023.

Contents

List of Figures

List of Tables

Acknowledgements

I wish to express my deep gratitude to my advisor Dr. Letterio Galletta that was more than supportive during my PhD research. I want to thank him for helping me grow as a researcher and for making all these years easy with his kindness and patience. I want to thank my parents for being the most supportive parents during my whole education. Also, thanks to family, my partner and friends from Serbia and from Italy for being always a big encouragement. Next, I want to thank professor Jovanka Pantovic for her support and her advice to enroll at IMT School for Advanced Studies, which has been one of the best experiences. Special thanks to my committee Marco Carbone and Mariangiola Dezani-Ciancaglini whose attentive reading and invaluable suggestions improved this dissertation. Finally, I want to thank all the professors from IMT for their knowledge transfer.

Vita

February 29, 1992	Born, Novi Sad , Serbia
2011-2015	Professor of Mathematics Final mark: 8.92/10.00 Faculty of Natural Sciences, Novi Sad, Serbia
2015-2016	Engineer of Applied Mathematics Final mark: 10.00/10.00 Faculty of Technical Sciences, Novi Sad, Serbia
2016-2017	High School Teacher Subjects: Mathematics and Discrete mathematics IT High School "Smart", Novi Sad, Serbia
2017-	PhD Student Program: Computer Science, SySMA research group IMT School for Advancd Studies, Lucca, Italy
2020-2022	Teaching Assistant Subjects: Mathematical Analysis, Numerical Mathematics, Discrete Mathematics and Linear Algebra Alfa BK University, Belgrade, Serbia
2022-	System Test Engineer and Software System Test Manager (SSTM) Continental Automotive, Novi Sad, Serbia

Publications

1. Zorica Savanović, Letterio Galletta, and Hugo Torres Vieira (2020). "A type language for message passing component-based systems". In: *Proceedings 13th Interaction and Concurrency Experience, ICE 2020, On-line, 19 June 2020.* Ed. by Julien Lange et al. Vol. 324. EPTCS, pp. 3–24. DOI:10.4204/EPTCS.324.3.URL:https://doi.org/10.4204/EPTCS.324.3.31
2. Zorica Savanovic and Letterio Galletta. "A type language for distributed reactive components governed by communication protocols". In: Journal of Logical and Algebraic Methods in Programming 132 (2023), p. 100848. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2023.100848. URL: https: //www.sciencedirect.com/science/article/pii/S2352220823000020

Presentations

1. Zorica Savanović, Letterio Galletta, and Hugo Torres Vieira, “A type language for message passing component-based systems”, *ICE 2020, On-line, 19 June 2020.*
2. Zorica Savanović, Letterio Galletta, and Hugo Torres Vieira, “A type language for message passing component-based systems”, *Summer school on behavioural APIs, Leicester, UK, July 2019.*

Abstract

Component based software engineering (CBSE) is a methodology that aims to design and build software systems by assembling together reusable and loosely coupled components. Applying CBSE in a distributed setting is appealing but challenging: distributed applications require different remote components to interact following a well-defined protocol. This thesis addresses a model for message passing component-based systems where components are assembled together with the protocol itself. Components can therefore be independent from the protocol, and can react to messages in a flexible way. This thesis studies how types can capture component behaviour and can enable checking the compatibility with a protocol. In particular, this thesis proposes two type languages for reactive components: the first language excludes choice terms, whereas the second one includes them. We show the correspondence of component and type behaviours, which entails a progress property for components.

Chapter 1

Introduction

Code reusability is an important principle to support the development of software systems in a cost-effective way. It is a key principle in Component-Based System Engineering (CBSE) [18], where the idea is to build systems relying on the composition of loosely-coupled and independent units called components.

The motivations behind CBSE are, on the one hand, to increase development efficiency and lower the costs (by building a system from preexisting components, instead of building from scratch), and on the other hand, to improve quality of the software for instance to what concerns software errors (components can be tested over and over again in different contexts). Consider, for example, microservices (see, e.g., [10]) that have been recently adopted by massively deployed applications such as Netflix, eBay, Amazon and Uber, and that are reusable distributed software units. In such a distributed setting, composing software elements necessarily involves some form of communication scheme, for instance based on message passing.

1.1 Research motivation

The internet has evolved so much that nowadays we have a collective network of connected devices and the technology that promote communication between devices and the cloud, as well as between the devices themselves. Such a network that allows the physical world to be digitally monitored or even controlled is called Internet of Things[16] (IoT). The internet is not only a network of computers, but it has evolved into a network of devices of all types: vehicles, smart phones, home appliances, toys, cameras, medical instruments and industrial systems, animals, people, buildings, all connected, all communicating and sharing information based on protocols in order to achieve smart analysis, tracing, safe and control and even personal real time online monitoring. In order to be considered an IoT system, four components must be properly integrated into the system: sensors/devices, connectivity, data processing, and user interface. The IoT technology in the device allows the device to create an interaction between internal components and the outside world, that assists in decision-making. An IoT system can be seen as a set of distributed interactive components, which could be designed applying Component-Based Software Engineering (CBSE). In recent years, component-based design has shown great promise in dealing with the complexity in modern systems. Component-Based Software Engineering uses the approach to form a complete system from pre-designed components. This approach has the potential of increasing design productivity by reusing the same components in multiple designs. One of the fundamental questions is: when we compose components to form a system, how can we ensure that they will work together?

In order for the functionality to be achieved, communication among components should follow a well-defined protocol of interaction, that may be specified in terms of some choreography language like, for example, WS-CDL [25] or the choreography diagrams of BPMN [20]. A component should be able to carry out a certain sequence of I/O actions in order to fulfil its role in the protocol. One way to accomplish this is to implement a component in a way that executes a strict sequence of I/O actions, that precisely matches the actions expected by the protocol. However, this choice interferes with reusability, since such a component can be used only in an environment that expects that exact sequence of actions. For instance, if a component receives an image and outputs its classification just once, what will happen if we need to use this component in a context where the classification is sent multiple times?

In contrast, a more flexible design choice comes from reactive programming and consists of designing components so that they can respond to external stimulus without any specific I/O sequence. The reactive programming principle for building such components considers that as soon as the data is available, it can be received or emitted. For example, we can design a component that is able to output a classification after receiving an image, as long as required. In such a way, reusability is promoted since such components can be used in different environments thanks to the flexibility given by the reactive behaviour. However, such a flexibility at the composition level may be too wild if all components are able to send/receive data as soon as it is available. Hence, we need to discipline the interactions at the level of the environment where the composition takes place. What if, for example, we have different images that need to be classified and the classifying component is continuously emitting the result for the first image?

Carbone et al. [5] proposed a language that supports the development of distributed systems by combining the notions of reactive components with choreographic specifications of communication protocols [19]. The proposal considers components that can dynamically send/receive data as soon as it is available, while considering that an assembly of components is governed by a protocol. Hence, among all the possible reactions that are supported by the composed components, the only ones that will actually be carried out are the ones allowed by the protocol. A composition of components defines itself a component that can be further composed (under the governance of some protocol) also providing a reactive behaviour. This approach promotes reusability thanks to the flexibility of the reactive behaviour. For instance, it abstracts from the number of supported reactions, because if a component can (always) perform a computation reacting to some inputs, then it can be used in different protocols that require such computation independently of the number of times; it also abstracts from message ordering, indeed, if a component needs some values to perform a computation, it may be used with any protocol that provides them in any order.

Component implementations should be hidden, so it should not be necessary to inspect its internals in order to asses if it is usable in a determined context for the purpose of *off-the-shelf* reuse. Hence, a component should be characterised with a signature that allows checking its compatibility when used in a composition. In particular, it must be ensured that each component provides (at least) the behaviour prescribed by the protocol in which the component participates. Carbone et al. [5] propose a verification technique that ensures communication safety and progress. However, the approach requires checking the implementation of components each time the component is put in a different context, i.e., each component should provide (at least) the behaviour as needed by the protocol that it participates in. This approach gives the answer to the question addressed to component: "Can you do this?". In this thesis our goal is to have a property of a component that allows the component to answer to the question: "What can you do?".

1.2 Contributions of the thesis

This thesis follows the research line of Carbone et al. [5] and considers an approach where we avoid the check each time a component is to be used. We introduce two different type languages that describe component behaviour and we check component implementation only once, during the type extraction of a component. After that, the type of a component is enough to capture the component reactive behaviour. The first type language, named **EC** type language (**EC**-excluding choices) characterises the behaviour of components governed by the protocol that excludes choices in its description. This approach is useful for many simpler system (such as [1]) and the type extraction procedure is far more easier and in some sense “elegant”. Then, we extend the protocol description with choice terms obtaining a new type language named **IC** type language (**IC**-including choices). This language captures the behaviour of greater number of components than the previous type language, but the type extraction procedure is more involved. The development of both languages follows the same steps: First, we introduce a type language that characterises the reactive behaviour of components, i.e., we assign them a type that labels components and provides all the information about their reactive behaviour. Secondly, we devise an inference technique that identifies the types of components, based on which we can verify whether the component provides the reactive behaviour required by a context. The motivation is in tune with reusability: once the component's type is identified, there is no further need to check the implementation, because the type is enough to describe “what the component can do".

Basically, our types specify the ability of components to receive values of a prescribed basic type. Moreover, they track different kinds of dependencies, for instance that certain values require a specific set of inputs (dubbed *per-each-value* dependencies) to be emitted always for each output. Our types can also describe the fact that a component needs to be, in some sense, initialised by receiving specific values before proceeding with other reactive behaviour (dubbed *initial* dependencies). Furthermore, our types also identify constraints on the number of values that a component can send. Finally, we ensure the correctness of our type system by proving that our type extraction procedures are sound with respect to the semantics of the Governed Components (GC) language [5], considering first the choice-free subset of the GC language and then a full account of the language. Moreover, we ensure that whenever a type of a component prescribes an action, a component will not be stuck, i.e., it will eventually carry out the matching action.

In this thesis we provide two type languages that ensure that the type of a component is enough to verify that the component provides (or not) the reactive behaviour required by any context. These types enable component reuse, which is a key benefit of component-based design. Moreover, by ensuring component compatibility, our type systems can greatly increase the robustness of a system, that is particularly valuable for IoT devices.

1.3 Outline

The thesis is organised in the following way:

Chapter 1 recalls the Behavioural types (Section 2.1) and Governed Components (GC) language (Section 2.2) that models components whose behaviour is captured by the research of this thesis.

Chapter 3 introduces an EC type language that characterises the reactive behaviour of components modelled in a choice-free subset of GC language. First, we intuitively introduce our type language through a motivating example based on AWS Lambda [1] where we point out different scenarios that might occur while composing components and how our types allow describing certain behavioural patterns in Section 3.1; in Section 3.2 we introduce the syntax and semantics of the type language; Then, we define the type extraction for base components in Section 3.3, whereas the type extraction for composite components in Section 3.4; finally, in Section 3.5 and Section 3.6 we state and proof the type safety of EC type language.

Chapter 4 introduces the IC type language that characterises the reactive behaviour of components modelled in (full) GC language. First we intuitively introduce our type language through a motivating example in Section 4.1; in Section 4.2 we introduce the syntax and semantics of IC type language; Then, we define the type extraction for base components in Section 4.3, whereas the type extraction for composite components in Section 4.4.3; finally, in Section 4.5 and Section 4.6 we state and proof the type safety of EC type language.

Chapter 5 concludes our work, relates it to other researches and discuss perspectives of future work.

1.3.1 Published papers based on our research

The content of the thesis is a collection, revision and extension of the material which we developed during the Ph.D.:

1. Zorica Savanovic, Letterio Galletta, and Hugo Torres Vieira (2020). "A type language for message passing component-based systems". In: Proceedings 13th Interaction and Concurrency Experience, ICE 2020, On-line, 19 June 2020.Ed. by Julien Lange et al. Vol. 324. EPTCS, pp. 3–24.

2. Zorica Savanovic and Letterio Galletta. "A type language for distributed reactive components governed by communication protocols". In: Journal of Logical and Algebraic Methods in Programming 132 (2023), p. 100848. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2023.100848. URL: https://www.sciencedirect.com/science/article/pii/S2352220823000020

Chapter 2

Background

This chapter is an introduction to the relevant background for the thesis. First, we recall behavioural types in terms of session types and then we introduce the GC language as a fundamental model of reactive components whose behaviour we consider in this thesis.

2.1 Behavioural types

Type systems *"A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute."* ([22])

The essential idea behind the type systems is to prevent execution errors during the execution of a program, i.e. to reduce the possibilities for bugs in computer programs. Essentially, it is a deduction system comprising a set of rules that assigns a property called *type* to the various constructs of a programming language. Types play a central role in the design of modern programming languages, because they characterise the form expected by the result of a computation. Type systems are a simple form of static analysis which compute an over-approximation of program behaviour.

Typically formalising the type system of a programming language consists of three steps:

1. defining the syntax of the types, the typing environment and the type judgements;

2. defining the type rules;

3. proving the soundness of the type system with respect to the dynamic semantics of the language.

In this section we focus on the *behavioural type theory* that is the basis for the development of communication-intensive distributed systems. The key idea of a behavioural type theory is to enrich the expressiveness of types so that it becomes possible to formally describe systems in distributes settings where participants exchange messages among them. It encompasses concepts such as interfaces, communication protocols and choreography, and describes a software entity, such as a component, in terms of the sequences of operations it is allowed to perform.

In order to formally introduce behavioural types, we present an example of behavioural type system based on session types that works on a (slightly changed version of) π-calculus [21], following the work by [7].

A basic concept for structuring communication-based programs is the notion of session. A session is designated by a private port called *channel*, through which interactions are performed. A channel is an abstraction of a communication link between processes.

Syntax We introduce the following syntactic categories and notations:

- $P, Q, \ldots$ are the processes
- $n, m, p, ... \in$ Int, where Int is a set of Integers;
- $c \in \{l, r\}$, where $\{l, r\}$ is set of channels (left and right, respectively);
- $\ell \in \{inl, inr\}$, where $\{inl, inr\}$ is a set of selectors;
- $v, w \ldots$ is a set of values;
- $s, t \cdots \in$ Bool $\cup$ Int represent basic types: Boolean constants [Bool $= \{true, false\}$] and Integers [Int $= \mathbb{Z}$];

- $x, y, z \ldots$ is a countable set of variables;
- $e_1, e_2, \ldots$ is a set of expressions (variables, values (constants) or equality $e_1 = e_2$).

Similar to the π-calculus, the syntax of processes is defined by following grammar:

$$\begin{array}{rl} P ::= & \mathbf{0} \\ & \mid c?(x:t).P \\ & \mid c!\langle e\rangle.P \\ & \mid c \lhd \ell.P \\ & \mid c \rhd \{P, Q\} \\ & \mid \text{if } e \text{ than } P \text{ else } Q \\ & \mid P\rfloor\lceil Q, \end{array}$$

Term **0** represents a process that is terminated; $c?(x : t).P$ is an input process (receiving message), announcing the reception of the message x with a basic type t on a channel c; $c!\langle e\rangle.P$ is an output process (sending message) of the expression e from the channel c; $c \lhd \ell.P$ is a labelled-driven selection where selection of the process depends on selector ℓ that can be either `inl` or `inr`; and $c \rhd \{P, Q\}$ denotes branching; term "if e than P else Q" denotes a conditional process and the last one $P\rfloor\lceil Q$ is for the parallel composition.

This calculus describes communication only between adjacent processes. Each channel of the process has its left and right side, and it can both receive and send a message. If we have parallel composition $P\rfloor\lceil Q$ only the right side of the process P can communicate with the left side of the process Q as graphically presented in Figure 1.

Notice that for that reason parallel composition operator is associative, but not symmetric in general. In the picture below is presented the parallel composition of three processes P, R and Q. The process R filters or/and transforms the message exchange between processes P and Q.

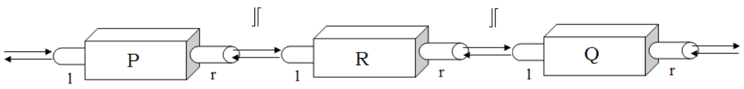

Figure 1: Parallel adapters/processes

2.1.1 Operational semantics

The operational semantics of processes is formalised as a reduction relation. This relation is closed by reduction contexts and a structural congruence relation, which we define next.

Process environment or context

Process environment plays a key role, because depending on environment process behaves differently. Formally we define a *reduction context* with the following grammar:

$$\mathcal{C} ::= \; [\,] \mid \mathcal{C}\rfloor\lceil P \mid P\rfloor\lceil\mathcal{C}$$

Intuitively, a *reduction context* is a process which has a "hole" marked as "[ ]". The hole serves as a place keeper for writing the next executed process. $\mathcal{C}[P]$ stands for processes obtained by replacing the hole in $\mathcal{C}$ with the process P; $\mathcal{C}\rfloor\lceil P$ and $P\rfloor\lceil\mathcal{C}$ says that we can add a parallel process to the process P from the left or the right side, respectively.

Notice that we denote with $\mathcal{C} ::= [\,]$ the identity process, i.e., for every process Q put in this context we have $\mathcal{C}[Q] = Q$.

Structural congruence

We introduce a structural congruence as the least congruence satisfying the following equations:

1. $\mathbf{0}\rfloor\lceil\mathbf{0} \equiv \mathbf{0}$
2. $P\rfloor\lceil(Q\rfloor\lceil R) \equiv (P\rfloor\lceil Q)\rfloor\lceil R$

The first congruence states that parallel composition of two terminated processes is a terminated process itself. The second one claims the associativity of operator $\rfloor\lceil$.

Reduction

Reduction relation $\longrightarrow$ is the least relation inductively defined by the rules of Table 1. We assume a deterministic evaluation relation $e \downarrow v$, expressing that expression e evaluates to value v. Value v has a type t, and we write it as $v \in t$, where $t \in \{Bool, Nat\}$.

$$\frac{e \downarrow v \quad v \in t}{r!\langle e\rangle.P\rfloor\lceil l?(x:t).Q \longrightarrow P\rfloor\lceil Q\{v/x\}}\ [R-COMM\ R]$$

$$\frac{e \downarrow v \quad v \in t}{r?(x:t).P\rfloor\lceil l!\langle e\rangle.Q \longrightarrow P\{v/x\}\rfloor\lceil Q}\ [R-COMM\ L]$$

$$\frac{}{r \triangleleft \ell.P\rfloor\lceil l \triangleright \{Q_{inl}, Q_inl\} \longrightarrow P\rfloor\lceil Q_\ell}\ [R-CHOICE\ R]$$

$$\frac{}{r \triangleright \{P_{inl}, P_{inr}\}\rfloor\lceil l \triangleleft \ell.Q \longrightarrow P_\ell\rfloor\lceil Q}\ [R-CHOICE\ L]$$

$$\frac{e \downarrow v \quad v \in \text{Bool}}{\text{if } e \text{ then } P_{true} \text{ else } P_{false} \longrightarrow P_v}\ [R-COND]$$

$$\frac{P \longrightarrow Q}{\mathcal{C}\ [P] \longrightarrow \mathcal{C}\ [Q]}\ [R-CONTEXT]$$

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}\ [R-STRUCT]$$

Table 1: Reduction relation

We briefly comment below the rules of Table 1. Rule [R-Choice R] states that having a process that selects (via selector ℓ) from its right channel and continues as P, in parallel with a left side of two processes (branching), the resulting process will be P parallel with the process which selector ℓ chose. Rule [R-COMM R] states that if v is the value of e, with a basic type t, then the parallel communication between the processes where we have sending an expression on the right channel (then continue as P) and receiving some variable x of the type t on the left channel (then continue as Q) reduces to a parallel communication between P and Q, where all occurrences of the variable (x) are evaluated with v.

Rule [R-COND] states that depending on evaluation of expression e, P reduces to P_{false} or P_{true}. Rule $[R-CONTEXT]$ states that if process P reduces to process Q, then if put in the same context, the reduction between two processes remains.

In order to define equivalence between processes we need to introduce a few concepts. We denote with $\longrightarrow^*$ the reflexive, transitive closure of relation $\longrightarrow$, i.e., $P \longrightarrow^* Q$ that means: process P reduces to process Q in finite number of steps by applying reduction rules. If there is no such of process Q we will write $P \not\longrightarrow$.

Correct process

Informally, we can say the set of *correct processes* include those processes that terminate every interaction and reduce to **0** in finite number of steps.

Definition 2.1.1 (Correct processes)**.** *We say that process P is correct if $P \longrightarrow^* Q$ and $Q \not\longrightarrow$ implies $Q \equiv \mathbf{0}$.*

Definition 2.1.2 (Equivalent processes)**.** *Two processes P and Q are equivalent, written $P \approx Q$, whenever for every context $\mathcal{C}$ we have that P is correct in $\mathcal{C}$ if and only if Q is correct in $\mathcal{C}$. Formally,*

$$\forall \mathcal{C}.\ P \approx Q \iff \mathcal{C}[P] \textit{ is correct if and only if } \mathcal{C}[Q] \textit{ is correct.}$$

The following picture clarifies the notion of process equivalence:

$$\begin{tikzcd}
 & P \approx Q & \\
\mathcal{C}[P] \arrow[d, "*"] & \text{iff} & \mathcal{C}[Q] \arrow[d, "*"] \\
Q'_1 \arrow[d, "/" marking] & & Q'_2 \arrow[d, "/" marking] \\
{} \arrow[d, Rightarrow] & & {} \arrow[d, Rightarrow] \\
Q'_1 \equiv \mathbf{0} & & Q'_2 \equiv \mathbf{0}
\end{tikzcd}$$

The left side of the picture looking from the top to bottom represents the correctness of a process $\mathcal{C}[P]$ and the right side a correctness of a process $\mathcal{C}[Q]$.

Example 2.1.1. *Let P and Q be two processes defined as follows:*

$$P := l?(x : Nat).r!\langle 5\rangle.0$$

$$Q := r!\langle 5\rangle.l?(x : Nat).0.$$

We want to find a context in which both processes are correct. We assume that:

$$\mathcal{C}\,[\,l?(x : Nat).r!\langle 5\rangle.0\,] \longrightarrow^* \mathbf{0}.$$

One simple solution could be the following context:

$$\mathcal{C} := r!\langle n\rangle.0 \rfloor\lceil [\quad] \rfloor\lceil l?(x : Nat).0,$$

for some $n \in \mathbb{N}$.

The question is: Will the process Q terminate in this context? If we put Q in the same context we get the following:

$$\mathcal{C}\,[\,r!\langle 5\rangle.l?(x : Nat).0\,] =$$

$$r!\langle n\rangle.0 \rfloor\lceil r!\langle 5\rangle.l?(x : Nat).0 \rfloor\lceil l?(x : Nat).0 \longrightarrow$$

$$r!\langle n\rangle.0 \rfloor\lceil l?(x : Nat).0 \rfloor\lceil 0 \longrightarrow$$

$$0 \rfloor\lceil 0 \rfloor\lceil 0 \equiv 0.$$

We can conclude that both processes do terminate in this context, i.e.:

$$\begin{tikzcd} \mathcal{C}[P] \arrow[d, "*"] & \mathcal{C}[Q] \arrow[d, "*"] \\ \boldsymbol{0} & \boldsymbol{0} \end{tikzcd}$$

However, these two processes are not equivalent, because exists a context $\mathcal{C} := [\quad] \rfloor\lceil r!\langle n\rangle.0 \rfloor\lceil l?(x : Nat).0$ in which P is correct, but Q is not.

2.1.2 Session types

We chose to introduce session types as an example of behavioural types. The idea behind session types is to describe communication protocols as types, that can be checked either statically (at compile-time) or dynamically (at runtime). These types are used to ensure properties desirable in concurrent and distributed systems, i.e. absence of communication errors and deadlocks, and protocol conformance.

Session types are ranged over $T, S \ldots$ and defined by the following grammar:

$$T ::= \textbf{end} \mid t?.T \mid t!.T \mid T + S \mid T \oplus S,$$

Type **end** types a channel end on which no further interaction is possible; term $t?.T$ is an input of a value of type t, and $t!.T$ is for an output.

Branching $T+S$ and selection $T \oplus S$ are binary operators, consistent with the process language that we explained below.

The type system enforces duality of behaviours on endpoints. The dual of the session type S is a session type $\overline{S}$, obtained by switching an input and an output or a selection and a branching in S. Duality plays a key role when we talk about session types because it allows smooth communication between two ends of a channel.

To check if some type T_1 is a dual of type T_2, we must construct $\overline{T_1}$ first, then check if those two types are equivalent.

Example 2.1.2. *Let us have a type* $T = s?.t!.\mathbf{end}$. *Check if type* $S = s!.t?.\mathbf{end}$ *is the dual of the type* T.

We know that $\overline{s?} = s!$; $\overline{t!} = t?$; *and* $\overline{\mathbf{end}} = \mathbf{end}$. *Then constructed dual type* $\overline{T}$ *of type* T *is*

$$\overline{T} = s!.t?.\mathbf{end}$$

and $\overline{T} = S$.

The fundamental element of the type rules is the *type environment*, denoted by Γ, that is a finite function mapping variables to types. For example,

$$\Gamma \vdash e : \tau$$

asserts that e has the type τ under the assumption that free variables occurring in e have the types specified in Γ. These assumptions are called the *typing judgements*.

The main judgements of this calculus are:

- $\Gamma \vdash x : t$
- $\Gamma \vdash P \blacktriangleright \{c : T, \overline{c} : S\}$

First judgement says that x must have a type t in environment Γ. The other one states that "under the environment Γ, a well-typed process P has a typing $c : T, \overline{c} : S$". Typing $c : T, \overline{c} : S$ specifies P's behaviour at its free channels. Process P:

- is well-typed in environment Γ ;
- uses a channel c according to type T, and channel $\overline{c}$ according to type S.

To be sure that channels of the session have the corresponding (dual) type at the beginning of a session, we use typing rules from the Table 2. We now briefly explain some of the rules: Rule $[T - RCV]$ states that if P uses channel c according to type T, and channel $\overline{c}$ according to type S in a type environment where x has a type t, then the process $c?(x : t).P$ (input of variable x of type t on the channel c) uses channel c according to the type $t?.T$. Rule $[T - SEND]$ is similar, but for the output.

Rule $[T - PAR]$ states that two processes in order to be put in a parallel composition the right channel of the process on the right side (P) of a composition and the left channel of the process on the left side of the composition (Q) have to have dual types, since the communication happens on those two channels. Then when composed, that process has the channels of the remaining types (left as left in P and right as right in Q).

Using the rules from the Table 2 we can prove the following theorem that states that if both channels of a process have a termination type, then the process is correct.

Theorem 2.1.1. *If* $\vdash P \blacktriangleright \{l : \mathbf{end}, r : \mathbf{end}\}$, *then P is correct process.*

Proof. Using the type rules, it is clear that P can be only one of three processes:

- **0** process;
- parallel composition;
- conditional process.

The first one and the third one are obvious. In the case of parallel composition, lets take $P = P_1 \rfloor\lceil P_2 \rfloor\lceil \ldots \rfloor\lceil P_i \rfloor\lceil \ldots \rfloor\lceil P_n$, where $P_1, P_2, \ldots, P_n$ are processes that terminate in one step. Then the rule [T-PAR] requires:

$\vdash P_1 \blacktriangleright \{l : \mathbf{end}, r : T_1\}, \vdash P_i \blacktriangleright \{l : \overline{T}_{i-1}, r : T_i\}$ for $2 \leq i \leq n-1$ and $\vdash P_n \blacktriangleright \{l : \overline{T}_{n-1}, r : \mathbf{end}\}$,

for some types $T_1, T_2 \ldots T_n$. The proof is by induction on $T_1, T_2 \ldots T_n$.

- **Base of induction** Coincides on the first case.
- **Inductive step** Assume that $P_j,\ j \in \{1, 2, \ldots, n\}$ is not a conditional process. If it is, it can be reduced by the rule [T-COND].
- Notice that r and l are the only channels in P_1 and P_n (respectively). Then there must exist at least one index $j \in \{1, 2, \ldots, n-1\}$), such

$$\frac{}{\Gamma, x : t \vdash x : t}\ [T-VAR]$$

$$\frac{v \in t}{\Gamma \vdash v : t}\ [T-VAL]$$

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \mathit{bool}}\ [T-EQUIV]$$

$$\frac{\Gamma, x : t \vdash P \blacktriangleright \{c : T, \overline{c} : S\}}{\Gamma \vdash c?(x : t).P \blacktriangleright \{c : t?.T, \overline{c} : S\}}\ [T-RCV]$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash P \blacktriangleright \{c : T, \overline{c} : S\}}{\Gamma \vdash c!\langle e\rangle.P \blacktriangleright \{c : t!.T, \overline{c}.S\}}\ [T-SEND]$$

$$\frac{\Gamma \vdash P_k \blacktriangleright \{c : T_k, \overline{c} : S\}^{\{k=1,2\}}}{\Gamma \vdash c \triangleright \{P_1, P_2\} \blacktriangleright \{c : T_1 + T_2, \overline{c} : S\}}\ [T-BCH]$$

$$\frac{\Gamma \vdash P \blacktriangleright \{c : T_1, \overline{c} : S\}}{\Gamma \vdash c \triangleleft \mathit{inl}.P \blacktriangleright \{c : T_1 \oplus T_2, \overline{c} : S\}}\ [T-LSELECT]$$

$$\frac{\Gamma \vdash P \blacktriangleright \{c : T_2, \overline{c} : S\}}{\Gamma \vdash c \triangleleft \mathit{inr}.P \blacktriangleright \{c : T_1 \oplus T_2, \overline{c} : S\}}\ [T-RSELECT]$$

$$\frac{}{\Gamma \vdash \mathbf{0} \blacktriangleright \{l : \mathbf{end}, r : \mathbf{end}\}}\ [T-ID]$$

$$\frac{\Gamma \vdash e : \mathit{Bool} \qquad \Gamma \vdash P_k \blacktriangleright \{l : T, r : S\}^{\{k=1,2\}}}{\Gamma \vdash \text{if}\, e\, \text{than}\, P_1\, \text{else}\, P_2 \blacktriangleright \{l : T, r : S\}}\ [T-COND]$$

$$\frac{\Gamma \vdash P \blacktriangleright \{l : T, r : T'\} \qquad \Gamma \vdash Q \blacktriangleright \{l : \overline{T}', r : S\}}{\Gamma \vdash P \rfloor\lceil Q \blacktriangleright \{l : T, r : S\}}\ [T-PAR]$$

Table 2: Typing rules

that P_j starts with a communication/selection/branching on channel r, and P_{j+1} starts with a communication/selection/branching on channel l.

Observe the case $T_j = T_{inl} \oplus T_{inr}$ (other cases similar). Rules [T-R-SELECT], [T-R-SELECT] and [T-BCH] require:

$$P_j \equiv r \triangleleft \ell.Q$$

and

$$P_{j+1} \equiv l \triangleright \{Q_{inl}, Q_{inr}\}.$$

According to rules [R-CHOICE-R] and [R–CONTEXT] we have:

$$P \longrightarrow P_1 \rfloor\!\lceil \ldots \rfloor\!\lceil Q \rfloor\!\lceil Q_1 \rfloor\!\lceil \ldots \rfloor\!\lceil P_n.$$

This concludes the proof, since
$\vdash Q \blacktriangleright \{l : T_{j-1}, r : T_1\}, \vdash Q_1 \blacktriangleright \{l : \overline{T_1}, r : T_{j+1}\}.$

□

The following theorem states that if a well-typed process P can be reduced to Q by applying reduction rules a finite number of times, then Q is well-typed and has the same type as P.

Theorem 2.1.2 (Subject reduction). *If* $\Gamma \vdash P \blacktriangleright \{c : T, \overline{c} : S\}$ *and* $P \longrightarrow^* Q$ *then* $\Gamma \vdash Q \blacktriangleright \{c : T, \overline{c} : S\}$.

In conclusion, types are used to check statically some properties of processes, and also to help in proving theorems about behavioural properties of processes. Next, we present a language that describes entities called components, in distributed systems, where the interaction among them is done by message passing. This language model is the base of the thesis, since the main goal of this thesis consists of describing the reactive behaviour of such components.

2.2 GC language

In this section, we briefly report the GC language following the presentation of Carbone, Montesi, and Viera [5]. We focus on the main points that allow grasping the essence of the model and supporting a self-contained understanding of the rest of the thesis. For details about the language, we refer the reader to the original paper [5].

In GC, the computation performed by components is defined in the reactive style, using binders that dynamically produce results as soon as they get the input data that they need. Components can be composed, where a composition of components is always associated to a protocol, given as a choreography, that governs the flow of communications among the components. The composition of some components is itself a component which can be used in further compositions.

2.2.1 Syntax of GC language

The syntax of the GC language is in Table 3. There are two kinds of components (K): base and composite. Both kinds interact with the external environment by means of input and output ports exposed as the component's interface. Besides of the interface, components are defined by their implementation.

Components	$K ::=$	$[\tilde{x}\rangle\,\tilde{y}]\{L\}$ (base)
		$[\tilde{x}\rangle\,\tilde{y}]\{G; R; D; r[F]\}$ (composite)
Local Binders	$L ::=$	$y = f(\tilde{x})$
		L, L
Protocol	$G ::=$	$p\xrightarrow{\ell}\tilde{q}; G$ (communication)
		$p\xrightarrow{\ell}\tilde{q}(G, G)$ (choice)
		$\mu\mathbf{X}.G$ (recursion)
		$\mathbf{X}$ (recursion variable)
		end (termination)
Role Assignments	$R ::=$	$p = K$
		R, R
Distribution Binders	$D ::=$	$p.x \xleftarrow{\ell} q.y$
		D, D
Forwarders	$F ::=$	$z \leftarrow w$
		F, F

Table 3: Syntax of Governed Components.

In the case of a base component the implementation is given by a list of local binders ($\{L\}$). A local binder specifies a function, denoted as $y = f(\tilde{x})$, which is used to compute the output values for port y relying on values received on list of (input) ports $\tilde{x}$. We say that component's ability to output a value may depend on the received ones, where instead, components are always able to receive values. We abstract from the definition of such functions f and assume them to be total. Received values are processed in a FIFO discipline, so queues are added to the local binders at run-time (noted as $y = f(\tilde{x}) \langle \tilde{\sigma})$. Each element ($\sigma$) in a queue ($\tilde{\sigma}$) is a store defined as a partial mapping from input ports to values ($\tilde{\sigma} = \sigma_1, \sigma_2, \ldots, \sigma_k$, where the oldest received values are stored in σ_1, the second-oldest values in σ_2, and so on and so forth up to σ_k). For example, consider a component with a local binder $y = f(x_1, x_2) < \cdot$ (where the dot in the term "$< \cdot$" represents the empty queue) and assume that the component receives v_1 and v_2 on port x_1, and v_3, v_4 and v_5 on port x_2. Once all values are received, the queue contains the following three mappings $(x_1 \to v_1, x_2 \to v_3), (x_1 \to v_2, x_2 \to v_4), (x_2 \to 5)$ where two are "complete", i.e., have all the arguments to compute the function f and one is not.

The implementation of a composite component, in symbols

$$\{G; R; D; r[F]\},$$

is an assembly of subcomponents whose interaction is governed by a protocol (G). The set of subcomponents are given in R together with their *roles* in the interaction, e.g., we write $r = K$ to denote that component K is assigned to role r. Composite components also specify a list of *distribution binders* (D) that provide an association between the messages exchanged in the protocol (ℓ) and the ports (x, y) of the components. For example, the binder $p.x \xleftarrow{\ell} q.y$ states that a message with a label ℓ is emitted on port y of the component assigned to role q, and it is received on port x by the component assigned to role p. Ports are uniquely associated to message labels (ℓ) in such a way that each communication step in the protocol has a precise mapping to a port: all values emitted on a port will be carried in messages with the same label and all values received on a port will be delivered in messages with the same label. For example, every time a value is emitted from some port y it will be carried in a message labelled with ℓ and a value delivered on some port x is delivered on a message with the same label ℓ. Other labels cannot be attached to y, otherwise the association would not be unique. For example, the $class$ message label (or some other) cannot be attached to y_p otherwise the association would not be unique. Finally, subterm $r[F]$ is used to specify the *interfacing component*. This is the only subcomponent responsible for the interaction with the external environment and it is identified by its role r and by *forwarders* F, i.e., special connections with the ports of the composite components. The idea underlying forwarders is that values received on the input ports of the composite component are directly forwarded to the input ports of the interfacing subcomponent; and values emitted on the output ports of the interfacing subcomponent are forwarded to the output ports of the composite component. For example, the term $x' \leftarrow x$ is for forwarding an input, and the term $y \leftarrow y'$ is for forwarding an output, where x and y are the ports of the composite component and x' and y' are ports of the interfacing subcomponent.

Protocol specifications prescribe the interaction among a set of parties identified by roles. A communication term $p \xrightarrow{\ell} \tilde{q}; G$ specifies that role p sends the message labelled ℓ to the (nonempty) set of roles $\tilde{q}$, after which the protocol continues as specified by G. In a choice term $p \xrightarrow{\ell} \tilde{q}(G, G)$ role p sends the message labelled with ℓ to roles $\tilde{q}$ announcing the choice of proceeding either according to protocol G_1 or G_2. Then, the terms $\mu\mathbf{X}.G$ and $\mathbf{X}$ are for specifying recursive protocols. Finally, term $\mathbf{end}$ defines the termination of the protocol.

2.2.2 Operational semantics of GC language

We now present the operational semantics of the GC in terms of a labelled transition system (LTS). We denote by $K \xrightarrow{\lambda} K'$ that a component K evolves in one computational step to K', where observations are captured by labels defined as follows

$$\lambda ::= x?v \mid y!v \mid \tau.$$

Transition label $x?v$ represents an input on port x of a value v; label $y!v$ denotes an output on port y of a value v; and label τ stands for an internal move.

We present the rules that describe the behaviour of components in two parts, addressing base and composite components separately.

Table 4 shows rules OutBase and InpBase that capture base component behaviour. These rules rely on an auxiliary transition system for local binders, denoted by $L \xrightarrow{\lambda} L'$. Rule OutBase states that if local binders L can perform the output of a value v on port y, and y is part of the component's interface, then the corresponding output can be exhibited by the base component. Rule InpBase follows the same lines.

Notice that the transition of the local binder specifies a final configuration L' which is accounted for in the evolution of the base component.

$$\frac{L \xrightarrow{y!v} L' \quad y \in \tilde{y}}{[\tilde{x}\,\rangle\,\tilde{y}]\{L\} \xrightarrow{y!v} [\tilde{x}\,\rangle\,\tilde{y}]\{L'\}}\ \textsf{OutBase}$$

$$\frac{L \xrightarrow{x?v} L' \quad x \in \tilde{x}}{[\tilde{x}\,\rangle\,\tilde{y}]\{L\} \xrightarrow{x?v} [\tilde{x}\,\rangle\,\tilde{y}]\{L'\}}\ \textsf{InpBase}$$

$$\frac{f() \downarrow v}{y \;=\; f()\,\langle\,\cdot \xrightarrow{y!v} y \;=\; f()\,\langle\,\cdot}\ \textsf{LConst}$$

$$\frac{\{\tilde{x}\}{=}\mathsf{dom}(\sigma) \quad f(\sigma(\tilde{x})) \downarrow v}{y \;=\; f(\tilde{x})\,\langle\,\sigma, \tilde{\sigma} \xrightarrow{y!v} y \;=\; f(\tilde{x})\,\langle\,\tilde{\sigma}}\ \textsf{LOut}$$

$$\frac{x \in \bigcap_{\sigma_i \in \tilde{\sigma}} \mathsf{dom}(\sigma_i) \quad x \in \tilde{x}}{y \;=\; f(\tilde{x})\,\langle\,\tilde{\sigma} \xrightarrow{x?v} y \;=\; f(\tilde{x})\,\langle\,\tilde{\sigma}, \{x \mapsto v\}}\ \textsf{LInpNew}$$

$$\frac{x \in \bigcap_{\sigma_i \in \tilde{\sigma}_1} \mathsf{dom}(\sigma_i) \quad x \notin \mathsf{dom}(\sigma) \quad x \in \tilde{x}}{y \;=\; f(\tilde{x})\,\langle\,\tilde{\sigma}_1, \sigma, \tilde{\sigma}_2 \xrightarrow{x?v} y \;=\; f(\tilde{x})\,\langle\,\tilde{\sigma}_1, \sigma[x \mapsto v], \tilde{\sigma}_2}\ \textsf{LInpUpd}$$

$$\frac{x \notin \tilde{x}}{y \;=\; f(\tilde{x})\,\langle\,\tilde{\sigma} \xrightarrow{x?v} y \;=\; f(\tilde{x})\,\langle\,\tilde{\sigma}}\ \textsf{LInpDisc}$$

$$\frac{L_1 \xrightarrow{y!v} L_1'}{L_1, L_2 \xrightarrow{y!v} L_1', L_2}\ \textsf{LOutLift} \qquad \frac{L_1 \xrightarrow{x?v} L_1' \quad L_2 \xrightarrow{x?v} L_2'}{L_1, L_2 \xrightarrow{x?v} L_1', L_2'}\ \textsf{LInpList}$$

Table 4: Semantics of base components.

$$\frac{K \xrightarrow{u!v} K' \qquad D = q.z \xleftarrow{\ell} p.u, D' \qquad G \xrightarrow{p!\ell\langle v\rangle} G'}{[\tilde{x}\,\rangle\,\tilde{y}]\{G; p{=}K, R; D; r[F]\} \xrightarrow{\tau} [\tilde{x}\,\rangle\,\tilde{y}]\{G'; p{=}K', R; D; r[F]\}}\ \textsf{OutChor}$$

$$\frac{K \xrightarrow{z?v} K' \qquad D = q.z \xleftarrow{\ell} p.u, D' \qquad G \xrightarrow{q?\ell\langle v\rangle} G'}{[\tilde{x}\,\rangle\,\tilde{y}]\{G; q{=}K, R; D; r[F]\} \xrightarrow{\tau} [\tilde{x}\,\rangle\,\tilde{y}]\{G'; q{=}K', R; D; r[F]\}}\ \textsf{InpChor}$$

$$\frac{K \xrightarrow{\tau} K'}{[\tilde{x}\,\rangle\,\tilde{y}]\{G; s{=}K, R; D; r[F]\} \xrightarrow{\tau} [\tilde{x}\,\rangle\,\tilde{y}]\{G; s{=}K', R; D; r[F]\}}\ \textsf{Internal}$$

$$\frac{K \xrightarrow{z!v} K' \qquad F = y \leftarrow z, F' \qquad y \in \tilde{y}}{[\tilde{x}\,\rangle\,\tilde{y}]\{G; r{=}K, R; D; r[F]\} \xrightarrow{y!v} [\tilde{x}\,\rangle\,\tilde{y}]\{G; r{=}K', R; D; r[F]\}}\ \textsf{OutComp}$$

$$\frac{K \xrightarrow{z?v} K' \qquad F = z \leftarrow x, F' \qquad x \in \tilde{x}}{[\tilde{x}\,\rangle\,\tilde{y}]\{G; r{=}K, R; D; r[F]\} \xrightarrow{x?v} [\tilde{x}\,\rangle\,\tilde{y}]\{G; r{=}K', R; D; r[F]\}}\ \textsf{InpComp}$$

Table 5: Semantics of composite components.

Essentially, a (run-time) local binder $y = f(\tilde{x}) \langle \tilde{\sigma}$ is always receptive to an input $x?v$: if x is not used in the function ($x \notin \tilde{x}$), the value v is simply discarded; otherwise, the value is added to the (oldest) entry in mapping queue $\tilde{\sigma}$ that does not have an entry for x (possibly originating a new mapping at the tail of $\tilde{\sigma}$). All local binders in L synchronise on an input, so each local binder will store (or discard) its own copy of the value. Instead, local binder outputs are not synchronised among them: if a local binder outputs a value, other local binders will not react. When the oldest mapping in queue $\tilde{\sigma}$ is complete, i.e., it assigns values to all of $\tilde{x}$, the function f may be computed, and its result is then carried in the transition label (i.e., the v in $y!v$).

We now introduce the rules that capture the behaviour of the composite components, displayed in Table 5. Notice that a composite component may itself be used as a subcomponent of another composition (of a "bigger" component), and base components provide the syntactic leaves.

Rules OutComp and InpComp capture the interaction of a composite component with an external environment, realised by the interfacing subcomponent. The role assignment $r = K$ captures the relation between component K and role r, which is specified as the interfacing role ($r[F]$). Rule OutComp allows for the interfacing component K to send a value v to the external environment via one of the ports y of the composite component. Notice that the connection between the port z of the interfacing component and the port y of the composite component is specified in a forwarder $F = y \leftarrow z, F'$. Rule InpComp follows the same lines to model an externally-observable input. Rule Internal allows for internal actions in a subcomponent K, where the final configuration K' is registered in the final configuration of the composite component.

Rules OutChor and InpChor capture the interaction among subcomponents of a composite component. Rule OutChor addresses the case when a component is sending a message to another one. The premises, together with role assignment $p = K$, establish the connection among sender component K, the component port u, sender role p, and message label ℓ. Premise $K \xrightarrow{u!v} K'$ says that the sender component K can perform an output of value v on port u. Premise $D = q.z \xleftarrow{\ell} p.u, D'$ says that the distribution binders specify the (unique) relation between port u of sender role p and message label ℓ (receiver role q and associated port z are not important here). The last premise $G \xrightarrow{p!\ell\langle v\rangle} G'$ realises the component governing the protocol, i.e., saying that the communication is only possible if the protocol prescribes it. Namely, the premise says that the protocol exhibits an output of a value v carried in message ℓ from role p. Naturally, operational semantics has an impact on our technical development (namely regarding end-point projection in local protocols), but to some extent can be addressed in a modular way (i.e., up to the existence of the end-point projection). Notice that the transitions of component K and protocol G specify final configurations K' and G' which are accounted for in the evolution of the composite component.

Rule InpChor is similar, but instead of message sending, it addresses the case when a subcomponent receives a message from another subcomponent. The premises are equivalent to the ones for Rule OutChor, but now regard reception. Namely, it says that the receiving component K performs a corresponding input transition, that the distribution binder specifies the relation of message label ℓ with receiver role q and port z, and that protocol G allows the input of a value.

The premises of rules InpChor and OutChor include the protocol transitions. Figure shows how the protocol evolves when some role p sends (denoted by $p!\ell\langle v\rangle$) or receives (denoted by $p?\ell\langle v\rangle$) a value v on the interaction ℓ. Below, we use α to range over these labels. We also want that the values are captured at the intermediate communication states, so the following run-time terms need to be added to the syntax of the protocols:

$$G ::= \ldots \mid \xrightarrow{\ell,v}\tilde{q};G \mid \xrightarrow{\ell,v}\tilde{q}(G_1,G_2)$$

Both terms above indicate that the message has been sent by a sender, but still not received by the receiver. We now comment on the semantics rules for protocols shown in Figure 6. Rule GSVal represents the outputs. Role p sends a value v labelled by ℓ ($p!\ell\langle v\rangle$) to multiple roles $\tilde{q}$. The transition to the run-time term$\xrightarrow{\ell,v}\tilde{q}$, registers the communication of a value v that still needs to be received by $\tilde{q}$. Rule GRVal models an input: if there is a value in passage, it will be consumed by one of the receivers.

Rule GSChoice is similar to the rule GSVal, but for choices. The value in the transmission needs to be one of the constants `inl` or `inr` which give the information to receivers whether the sender chose that communication proceeds as protocol G_1 or G_2. Rule GRChoice is similar as rule GRVal, but for choices.

When there is only one receiver, the communication proceeds following the continuation, as described in the rules GRVal2 and GRChoice2. Rule GRec captures recursion in the standard way. The rules labelled with GConc describe the asynchronous and concurrent execution of the protocol, i.e, if an action does not include the role in the prefix of the protocol, the continuation is allowed to make a transition, leaving the prefix unmodified. Note that in these rules the notation $role(\alpha)$ is used to extract the name of the role from the label.

$$\frac{}{p \xrightarrow{\ell} \tilde{q};\ G \quad \xrightarrow{p!\ell\langle v\rangle} \quad \xrightarrow{\ell,v} \tilde{q};\ G} \ \textsf{GSVal}$$

$$\frac{v \in \{\texttt{inl}, \texttt{inr}\}}{p \xrightarrow{\ell} \tilde{q}(G_1, G_2) \quad \xrightarrow{p!\ell\langle v\rangle} \quad \xrightarrow{\ell,v} \tilde{q}(G_1, G_2)} \ \textsf{GSChoice}$$

$$\frac{\tilde{q} \text{ nonempty}}{\xrightarrow{\ell,v} \tilde{q}, q;\ G \quad \xrightarrow{q?\ell\langle v\rangle} \quad \xrightarrow{\ell,v} \tilde{q};\ G} \ \textsf{GRVal}$$

$$\frac{}{\xrightarrow{\ell,v} q;\ G \quad \xrightarrow{q?\ell\langle v\rangle} \quad G} \ \textsf{GRVal2}$$

$$\frac{\tilde{q} \text{ nonempty}}{\xrightarrow{\ell,v} \tilde{q}, q(G_1, G_2) \quad \xrightarrow{q?\ell\langle v\rangle} \quad \xrightarrow{\ell,v} \tilde{q}(G_1, G_2)} \ \textsf{GRChoice}$$

$$\frac{v \in \{\texttt{inl}, \texttt{inr}\}}{\xrightarrow{\ell,v} q(G_{\texttt{inl}}, G_{\texttt{inr}}) \quad \xrightarrow{q?\ell\langle v\rangle} \quad G_v} \ \textsf{GRChoice2}$$

$$\frac{G\{^{\mu\mathbf{X}.G}/_{\mathbf{X}}\} \ \xrightarrow{\alpha} \ G'}{\mu\mathbf{X}.G \ \xrightarrow{\alpha} \ G'} \ \textsf{GRec}$$

$$\frac{G \ \xrightarrow{\alpha} \ G' \quad role(\alpha) \notin p, \tilde{q}}{p \xrightarrow{\ell} \tilde{q};\ G \quad \xrightarrow{\alpha} \quad p \xrightarrow{\ell} \tilde{q};\ G'} \ \textsf{GConc1}$$

$$\frac{G_1 \ \xrightarrow{\alpha} \ G_1' \quad G_2 \ \xrightarrow{\alpha} \ G_2' \quad role(\alpha) \notin p, \tilde{q}}{p \xrightarrow{\ell} \tilde{q}(G_1, G_2) \quad \xrightarrow{\alpha} \quad p \xrightarrow{\ell} \tilde{q}(G_1', G_2')} \ \textsf{GConc2}$$

$$\frac{G \ \xrightarrow{\alpha} \ G' \quad role(\alpha) \notin \tilde{q}}{\xrightarrow{\ell,v} \tilde{q};\ G \quad \xrightarrow{\alpha} \quad \xrightarrow{\ell,v} \tilde{q};\ G'} \ \textsf{GConc3}$$

$$\frac{G_1 \ \xrightarrow{\alpha} \ G_1' \quad role(\alpha) \notin \tilde{q}}{\xrightarrow{\ell,\texttt{inl}} \tilde{q}(G_1, G_2) \quad \xrightarrow{\alpha} \quad \xrightarrow{\ell,\texttt{inl}} \tilde{q}(G_1', G_2)} \ \textsf{GConc4}$$

$$\frac{G_2 \ \xrightarrow{\alpha} \ G_2' \quad role(\alpha) \notin \tilde{q}}{\xrightarrow{\ell,\texttt{inr}} \tilde{q}(G_1, G_2) \quad \xrightarrow{\alpha} \quad \xrightarrow{\ell,\texttt{inr}} \tilde{q}(G_1, G_2')} \ \textsf{GConc5}$$

Table 6: Semantics of protocols.

In GC language, the communication among the components inside the composite component is governed by the protocol. Each component needs to be able to carry-out the protocol, i.e., each component must be capable of performing the behaviour prescribed by the protocol, for the corresponding role. For this reason, we introduce an operation that returns the expected behaviour of the component according to its role in the composition and a protocol. The operation is named Protocol Projection and it is shown in Figure 7. We denote with $G \downarrow_{p,D,\gamma}$ the projection of protocol G to role p with the connection binder D and the mapping γ. The mapping γ maps the interaction labels to the base types of communicated values and it ensures that both sender and receiver agree on the type of the value. Moreover, we denote with $z!.B$ and $w?.B$ an output from the port z of a value of the type B and an input on the port w of a value of type B. This operation is crucial for extracting the type of a composite component. Indeed, as we discuss later, we use this operation to get the local protocols describing the behaviour of subcomponents.

Later on we show the illustrating examples of GC language, i.e. the description of components modelled in GC language.

$$
\begin{array}{ll}
(p \xrightarrow{\ell} \tilde{q}; G)\downarrow_{p,D,\gamma} & \triangleq\ z!B.G\downarrow_{p,D,\gamma} \\
 & \text{where } (\ D = q.w \xleftarrow{\ell} p.z, D' \ \wedge\ \gamma(\ell) = B) \\
(p \xrightarrow{\ell} \tilde{q}, q; G)\downarrow_{q,D,\gamma} & \triangleq\ w?B.G\downarrow_{q,D,\gamma} \\
 & \text{where } (D = q.w \xleftarrow{\ell} p.z, D' \ \wedge\ \gamma(\ell) = B) \\
(p \xrightarrow{\ell} \tilde{q}; G)\downarrow_{r,D,\gamma} & \triangleq\ G\downarrow_{r,D,\gamma} \\
 & \text{where } (r \notin p, \tilde{q}) \\
(\xrightarrow{\ell,v} \tilde{q}, q; G)\downarrow_{q,D,\gamma} & \triangleq\ w?B.G\downarrow_{q,D,\gamma} \\
 & \text{where } (D = q.w \xleftarrow{\ell} p.z, D' \ \wedge\ v : B) \\
(\xrightarrow{\ell,v} \tilde{q}; G)\downarrow_{r,D,\gamma} & \triangleq\ G\downarrow_{r,D,\gamma} \\
 & \text{where } (r \notin \tilde{q}) \\
(p \xrightarrow{\ell} \tilde{q}(G_1, G_2))\downarrow_{p,D,\gamma} & \triangleq\ z \oplus (G_1\downarrow_{p,D,\gamma}, G_2\downarrow_{p,D,\gamma}) \\
 & \text{where } (D = q.w \xleftarrow{\ell} p.z, D') \\
(p \xrightarrow{\ell} \tilde{q}, q(G_1, G_2))\downarrow_{q,D,\gamma} & \triangleq\ w\,\&(G_1\downarrow_{q,D,\gamma}, G_2\downarrow_{q,D,\gamma}) \\
 & \text{where } (D = q.w \xleftarrow{\ell} p.z, D') \\
(p \xrightarrow{\ell} \tilde{q}(G_1, G_2))\downarrow_{r,D,\gamma} & \triangleq\ G_1\downarrow_{r,D,\gamma} \\
 & \text{where } (r \notin p, \tilde{q} \ \wedge\ G_1\downarrow_{r,D,\gamma} = G_2\downarrow_{r,D,\gamma}) \\
(\xrightarrow{\ell,v} \tilde{q}, q(G_1, G_2))\downarrow_{q,D,\gamma} & \triangleq\ w\,\&(G_1\downarrow_{q,D,\gamma}, G_2\downarrow_{q,D,\gamma}) \\
 & \text{where } (D = q.w \xleftarrow{\ell} p.z, D' \ \wedge\ v : \texttt{ChoT}) \\
(\xrightarrow{\ell,\texttt{inl}} \tilde{q}(G_1, G_2))\downarrow_{r,D,\gamma} & \triangleq\ G_1\downarrow_{r,D,\gamma} \\
 & \text{where } (r \notin \tilde{q}) \\
(\xrightarrow{\ell,\texttt{inr}} \tilde{q}(G_1, G_2))\downarrow_{r,D,\gamma} & \triangleq\ G_2\downarrow_{r,D,\gamma} \\
 & \text{where } (r \notin \tilde{q}) \\
(\mu\mathbf{X}.G)\downarrow_{r,D,\gamma} & \triangleq\ \mu\mathbf{X}.(G\downarrow_{r,D,\gamma}) \\
 & \text{where } (r \in \mathit{roles}(G)) \\
(\mu\mathbf{X}.G)\downarrow_{r,D,\gamma} \triangleq\ \mathbf{end} & \\
 & \text{where } (r \notin \mathit{roles}(G)) \\
\mathbf{X}\downarrow_{r,D,\gamma} & \triangleq\ \mathbf{X} \\
\mathbf{end}\downarrow_{r,D,\gamma} & \triangleq \mathbf{end}
\end{array}
$$

Table 7: Protocol projection (including run-time terms).

Chapter 3

The EC type language

In this chapter we introduce an EC type language that characterises the reactive behaviour of components modelled in a choice-free subset of GC language. First, we intuitively introduce our type language through a motivating example based on AWS Lambda [1] and then we formalise the language.

3.1 Informal introduction of EC type language

In order to motivate GC language and also to introduce our typing approach, we now informally discuss an example inspired by a microservices scenario [1] that addresses an Image Recognition System (IRS). The basic idea is that users upload images and receive back the resulting classification. Moreover, users can get the current running version of the system whenever desired.

The IRS is made of two microservices, $Portal$ and $Recognition$ $Engine$ (RE), that interact according to a predefined protocol.

The classification task is achieved according to the following workflow: $Portal$ sends the $image$ loaded by a user to RE to be processed. When RE service finishes its $classification$, it sends the $class$ as the result of the $classification$ to $Portal$. We model the scenario in the GC language by assigning to each microservice the corresponding role and using components to represent them. We assign role $Portal$ to component K_{Portal} and role RE to component K_{RE}, where K_{Portal} and K_{RE} are base components.

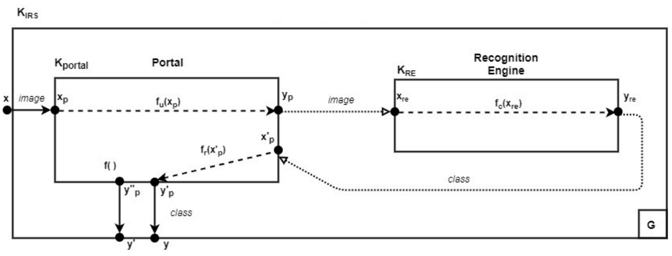

Figure 2: Image Recignition System

Interaction between these two components is governed by global protocol G, that is described as follows:

$$Portal \xrightarrow{image} RE; RE \xrightarrow{class} Portal.$$

This (the part of G) protocol exactly specifies the workflow described above: $Portal$ sends an $image$ to RE ($Portal \xrightarrow{image} RE$) that answers with the computed $class$ ($RE \xrightarrow{class} Portal$). If we add the termination (**end**) we obtain (complete G) protocol

$$Portal \xrightarrow{image} RE; RE \xrightarrow{class} Portal; \mathbf{end}$$

which may be described as a one-shot protocol, since the interaction is over (**end**) after the components exchange the two messages.

We obtain composite component K_{IRS} by assembling K_{Portal} and K_{RE} together with protocol G that governs the interaction.

Figure 2 shows how it is possible to graphically represent component K_{IRS}, where we represent K_{Portal} and K_{RE} as its subcomponents: The subcomponent K_{Portal} is the interfacing component (hence is the only one connected to the external environment via forwarders). We can specify K_{Portal} in the GC language as

$$[x_p, x'_p \rangle\, y_p, y'_p, y''_p]\{y_p = f_u(x_p) < \tilde{\sigma}^{y_p}, y'_p = f_r(x'_p) < \tilde{\sigma}^{y'_p}, y''_p = f() < \cdot\}$$

As previewed in the graphical illustration, from the specification we can see that K_{Portal} component has two input ports (x_p, x'_p), three output ports (y_p, y'_p, y''_p), and three local binders that at runtime are equipped with queues ($\tilde{\sigma}^{y_p}$, $\tilde{\sigma}^{y'_p}$ and empty queue $\cdot$ given that the respective binder does not use any input ports). Notice that the queues are only required at runtime and are initially empty.

The idea of our type description is to provide an abstract characterisation of component's behaviour. Types provide information about the set of input ports, namely the types of values that can be received on them, and about the output ports, namely their behavioural capabilities. In particular, for each output port there are constraints which comprise three pieces of information: (i) what type of values are emitted; (ii) what is the maximum number of values that can be emitted; and (iii) what are the dependencies on input ports, possibly including the number of currently available values that satisfy the dependency at runtime.

Informally, the type of K_{Portal} announces the following: In the two input ports x_p and x'_p the component can receive an $image$ and a $class$, respectively ($\{x_p(image), x'_p(class)\}$). Also, the type says the component emits $image$s from port y_p and it can do so an unbounded number of times (denoted by ∞), as the underlying local binder imposes no boundary constraints. In particular, the local binder can send an $image$ as soon as one is received in x_p. Hence, we have a *per each* value dependency of y_p on x_p. Formally, we write this constraint as $y_p(image) : \infty : [\{x_p : N_p\}]$, where N_p is the number of values received on x_p that are available to be used to produce the output on y_p. We may describe constraint $y'_p(class) : \infty : [\{x'_p : N'_p\}]$ in a similar way. In constraint $y''_p(version) : \infty : [\emptyset]$ there are no dependencies from input ports specified, hence the reading is only that a $version$ can be emitted an unbounded number of times. We may specify the type of K_{Portal} as:

$$\begin{aligned}
&T_{Portal} = < X_b; \mathbf{C} > \\
&X_b = \{x_p(image), x'_p(class)\} \\
&\mathbf{C} = \{C_1, C_2, C_3\} \\
&C_1 = y_p(image) : \infty : [\{x_p : N_p\}] \\
&C_2 = y'_p(class) : \infty : [\{x'_p : N'_p\}] \\
&C_3 = y''_p(version) : \infty : [\emptyset]
\end{aligned}$$

Composite component K_{IRS} is an assembly of two base components K_{Portal} and K_{RE} whose communication is governed by global protocol G. The description of K_{IRS} in GC language is the following:

$$K_{IRS} = [x \rangle\, y, y']\{G; Portal = K_{Portal}, RE = K_{RE}; D; Portal[F]\}$$

where G is the already described one-shot protocol

$$G = Portal \xrightarrow{image} RE; RE \xrightarrow{class} Portal; \mathbf{end}$$

Interfacing component K_{Portal} forwards the values from/to the external environment as specified in the forwarders ($F = x_p \leftarrow x, y \leftarrow y'_p, y' \leftarrow y''_p$). The forwarding implies that the characterisation of ports x, y and y' in the type of K_{IRS} relies on one of the ports x_p, y'_p and y''_p, respectively, in the type of K_{Portal}.

The type of K_{IRS} then says that it can always input on x values of type $image$ accordingly to the input receptiveness principle. The constraint for y' is the same as for y''_p since y''_p does not depend on the protocol (in fact it has no dependencies). However, this is not the case for y: in order for a *class* of an image to be forwarded from y'_p there is a dependency (identified in T_{Portal}) on port x'_p. Furthermore, component K_{Portal} will only receive a value on x'_p accordingly to the protocol specification, in particular upon the second message exchange. Hence, there is also a protocol dependency since the first message exchange has to happen first, so there is a transitive dependency to an $image$ being sent in the first message exchange, emitted from port y_p of component K_{Portal}. Finally, notice that y_p depends on x_p which is linked by forwarding to port x of component K_{IRS}, thus we have a sequence of dependencies that link y to x.

Since we have a one-shot protocol, the communications happens only once, which implies that one *class* is produced for the first $image$ received. We therefore consider that the dependency of y on x is *initial* (since one value suffices to break the one-shot dependency), and that the maximum number of values that can be emitted on y is 1. This constraint is formally written as $y(class) : 1 : [\{x : \Omega\}]$. The constraint for y' is $y'(version) : \infty : [\emptyset]$, where the set of dependencies is empty, i.e., it does not depend on any input. We then have the following type for component K_{IRS}:

$$T_{IRS} =< \{x(image)\}; \{y(class) : 1 : [\{x : \Omega\}], y'(version) : \infty : [\emptyset]\} >$$

Let us now assume a recursive version of protocol

$$G' = \mu \mathbf{X}.Portal \xrightarrow{image} RE; RE \xrightarrow{class} Portal; \mathbf{X}$$

is used instead. In other words, we consider the following component

$$K'_{IRS} = [x \rangle y, y']\{G'; Portal = K_{Portal}, RE = K_{RE}; D; Portal[F]\}.$$

The idea now is that for each $image$ received a *class* is produced. So, *class* may be emitted by y an unbounded number of times and the dependency of y on x is of a *per each* kind. Notice that the chain of dependencies can be described as before, but the one-shot dependency from before is now renewed at each protocol iteration.

The constraint for y in this settings is $y(class)\!:\!\infty\!:\![\{x\!:\!N_i\}]$, where N_i captures the number of values received on x that are currently available to produce the outputs on y. The constraint for y' is the same as in the previous case. We then have that the type of K'_{IRS} is

$$< \{x(image)\}; \{y(class)\!:\!\infty\!:\![\{x\!:\!N_i\}], y'(version)\!:\!\infty\!:\![\emptyset]\} >$$

Imagine that K'_{Portal} is now a composite component that has an initialisation phase such that, first it receives a message about what kind of classification is required (e.g., "classify the image by the number of faces found on it"), then it sends it to K'_{RE}, after which the uploading and classification of the images can start (all other characteristics remain). Let x_1 be the port of K'_{IRS} on which this message is received. Let us consider the following protocol

$$G'' = Portal \xrightarrow{kind} RE; \mu\mathbf{X}.Portal \xrightarrow{image} RE; RE \xrightarrow{class} Portal; \mathbf{X}$$

where after component K'_{Portal} sends the required kind of classifications (labelled as $kind$), the communication between K'_{Portal} and K'_{RE} is governed by a recursive protocol as described in the previous example. The type of the component K'_{IRS} is similar to the type from the previous example, but now announces that the output on y requires an initial value to be received on port x_1, as the image classification process can only start after that. We then have the type of K'_{IRS}

$$< \{x(image)\}; \{y(class)\!:\!\infty\!:\![\{x\!:\!N_i, x_1 : \Omega\}], y'(version)\!:\!\infty\!:\![\emptyset]\} >$$

3.2 Formal introduction of EC type language

In this section we define the type language that captures the behaviour of components in an abstract way, starting with the presentation of the syntax which is followed by the operational semantics. Then, in the next sections we present two procedures that define how to extract the type of a component. The first procedure is for base, and the second one is for composite components.

3.2.1 Syntax of EC type language

Types	$T \triangleq < X_b; \mathbf{C} >$
Input interfaces	$X_b \triangleq \{x_1(b_1), \ldots, x_k(b_k)\}$
Constraints	$\mathbf{C} \triangleq \{y_1(b_1) : \mathbf{B}_1 : [\mathbf{D}_1], \ldots, y_k(b_k) : \mathbf{B}_k : [\mathbf{D}_k]\}$
Dependencies	$\mathbf{D} \triangleq \{x_1 : M_1, \ldots, x_k : M_k\}$
Dependency kinds	$M ::= N \mid \Omega$
Boundaries	$\mathbf{B} ::= N \mid \infty$
Additional notations	$k \geq 0 \quad N \in \mathbb{N}_0$

Table 8: Type Syntax (EC type language)

The syntax of types is presented in Table 8 and some explanations follow. A type T consists of two elements: a (possibly empty) set of input ports, where each one is associated with a basic type b (i.e., int, string, etc.), and a (possibly empty) set of constraints **C**, one for each output port. Basic types (ranged over by $b, b_1, b_2, b^x, b^y, b', \ldots$) specify the type of the values that can be communicated through ports, so as to ensure that no unexpected values arise.

Each constraint in **C** contains a triple of the form $y(b) : \mathbf{B} : [\mathbf{D}]$, which describes the type (b) of values sent via y, the capability (**B**) of y and the dependencies (**D**) of y on the input ports. The set of constraints **C** is ranged over $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}', \mathbf{C}'', \mathbf{C}^y, \ldots$ (likewise other syntactic categories like N, **B**, **D**, $\ldots$). Capability **B** identifies the upper bound on the number of values that can be sent from the output port: a natural number N denotes a bounded capability, whereas ∞ an unbounded one. Dependencies are of two kinds: *per-each-value* dependencies are of the form $x : N$ and *initial* dependencies are given by $x : \Omega$. A dependency $x : N$ says that each value emitted on y requires the reception of one value on x, and furthermore N provides the (runtime) number of values available on x (hence, initially $N = 0$). Instead, a dependency $x : \Omega$ says that y initially depends on a (single) value received on x, hence the dependency is dropped after the first input on x.

Note that there are only two kinds of dependencies: a per-each-value dependency and an initial one. Since we aim at static typing, the dependencies that appear after the extraction of a type are either $x : 0$ or $x : \Omega$, but for the sake of showing our results, we need to capture these values in the evolution of the types. So, we need to capture the number of values available on the input ports, hence we have dependencies of the kind

$x\!:\!1, x\!:\!2, \ldots$ (for $N = 1, N = 2 \ldots$).

This thesis investigates the mathematical model for the purpose of showing our results, but we may already point towards practical applications. In particular, for the purpose of the (static) type verification we are aiming at, the *counting* required for the theoretical model would not be involved and the component type information available for developers would be as follows:

1. set of input ports with their basic types;
2. set of constraints for each output port with the following information:
   - 2.1 the basic type associated to the output port;
   - 2.2 one of two possibilities for output port capability: bounded or unbounded;
   - 2.3 one of two possibilities for each kind of dependency: per-each-value or initial.

Hence, for the sake of static type checking and from a developers perspective, apart the expected information regarding basic (value) types, the type information would be `y:bounded` or `y:unbounded` to what concerns output port capabilities and `x:per-each` or `x:initial` to what concerns dependencies.

3.2.2 Semantics of EC type language

We now define the operational semantics of the type language, that is required to show that types faithfully capture component behaviour. The semantics is given by the LTS shown in Table 9. There are four kinds of labels λ described by the following grammar:

$$\lambda ::= x? \mid x?(b) \mid y!(b) \mid \tau.$$

Label $x?$ denotes an input on x; whereas, label $x?(b)$ denotes an input of a value of type b; then, label $y!(b)$ represents an output of a value of type b; finally, τ captures an internal step.

$$\frac{x \notin dom[\mathbf{D}]}{y(b)\!:\!\mathbf{B}\!:\![\mathbf{D}] \xrightarrow{x?} y(b)\!:\!\mathbf{B}\!:\![\mathbf{D}]}\;[T1]$$

$$\frac{}{y(b)\!:\!\mathbf{B}\!:\![\{x:\Omega\} \uplus \mathbf{D}] \xrightarrow{x?} y(b)\!:\!\mathbf{B}\!:\![\mathbf{D}]}\;[T2]$$

$$\frac{}{y(b)\!:\!\mathbf{B}\!:\![\{x\!:\!N\} \uplus \mathbf{D}] \xrightarrow{x?} y(b)\!:\!\mathbf{B}\!:\![\{x\!:\!N+1\} \uplus \mathbf{D}]}\;[T3]$$

$$\frac{}{T \xrightarrow{\tau} T}\;[T4]$$

$$\frac{\forall i \in 1,2,\ldots,k \quad y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathbf{D}_i] \xrightarrow{x?} y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathbf{D}'_i]}{\begin{array}{c}<\{x(b^x) \uplus X_b\};\{y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathbf{D}_i]|1 \le i \le k\}> \xrightarrow{x?(b^x)} \\ <\{x(b^x) \uplus X_b\};\{y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathbf{D}'_i]|1 \le i \le k\}>\end{array}}\;[T5]$$

$$\frac{\forall i \in 1,2,\ldots,k \quad N_i \ge 1 \quad \mathbf{B} > 0}{\begin{array}{c}< X_b;\{y(b^y)\!:\!\mathbf{B}\!:\![\{x_i\!:\!N_i|1 \le i \le k\}]\} \uplus \mathbf{C} > \xrightarrow{y!(b^y)} \\ < X_b;\{y(b^y)\!:\!\mathbf{B}-1\!:\![\{x_i\!:\!N_i-1|1 \le i \le k\}]\} \uplus \mathbf{C} >\end{array}}\;[T6]$$

Table 9: Type Semantics (EC type language)

We briefly describe the rules shown in Table 9. Note that Rules [T1,T2,T3] describe inputs of a (single) constraint, while [T4, T5, T6] capture type behaviour.

Rule [T1] says a constraint for y can receive (and discard) an input on x in case y does not depend on x, i.e., if x is not in the domain of **D** ($dom(\mathbf{D}) = \{x \mid \mathbf{D} = \{x\!:\!M\} \uplus \mathbf{D}'\}$), leaving the constraint unchanged.

Rule [T2] addresses the case of an initial dependency, where after receiving the value on x the dependency is removed. Rule [T3] captures the case of a per-each-value dependency, where after the reception the number of values available on x for y is incremented.

With respect to type behaviour, Rule [T4] says that the type can exhibit an internal step and remain unchanged, used to mimic component internal steps (which have no impact on the interface). Rule [T5] states that if all type constraints can exhibit an input on x and x is part of the type input interface, then the type can exhibit the input on x considering the respective basic type. Notice that rules [T1,T2,T3] say that constraints can always exhibit an input (simply the effect may be different). Finally, Rule [T6] says that if one of the constraints has all of the dependencies met, i.e., has at least one value for each x for which there is a dependency, and also that the boundary has not been reached (i.e., it is greater than zero), then the type can exhibit the corresponding output implying the decrement of the boundary and of the number of values available in dependencies. Notice that in order for a port to output a value, there can be no initial dependencies present (which are dropped once satisfied), only per-each-value dependencies.

In the following example and in the rest of the chapter (where appropriate) we adopt the following notation: i abbreviates the $image$ type, c abbreviates the $class$ type and v abbreviates the $version$ type.

Example 3.2.1. *We revisit the type of component K_{Portal} shown in Section 3.1*

$T_{Portal} =< X_b; \boldsymbol{C} >$
$X_b =< \{x_p(i), x'_p(c)\}$
$\boldsymbol{C} = \{C_1, C_2, C_3\}$
$C_1 = y_p(i) : \infty : [\{x_p : N_p\}]$
$C_2 = y'_p(c) : \infty : [\{x'_p : N'_p\}]$
$C_3 = y''_p(v) : \infty : [\emptyset]$

for some N_1 and N_2. Recall also type

$$< \{x(i)\}; \{y(c) : 1 : [\{x : \Omega\}], y'(v) : 1 : [\emptyset]\} >$$

that may evolve upon the reception of an input on x as follows:

$$\dfrac{\dfrac{}{y(c):1:[\{x:\Omega\}] \xrightarrow{x?} y(c):0:[\emptyset]}\,[T2] \qquad \dfrac{x \notin dom[\boldsymbol{D}]}{y'(v):1:[\emptyset] \xrightarrow{x?} y'(v):1:[\emptyset]}\,[T1]}{< \{x(i)\}; \{y(c):1:[\{x:\Omega\}], y'(v):1:[\emptyset]\} > \xrightarrow{x?(i)} < \{x(i)\}; \{y(c):0:[\emptyset], y'(v):1:[\emptyset]\} >}\,[T5]$$

The type language serves as a means to capture component behaviour, and types for components may be obtained (inferred) as explained below. The results presented afterwards ensure that when the type extraction is possible, then each behaviour in the component is explained by a behaviour in the type, and that each behaviour in the type can eventually be exhibited by the component.

3.3 EC type extraction for base components

In this section we describe the procedure that allows to (automatically) extract the type of a component, focusing first on the case of base components, remembering their reactive flavour. The goal is to identify the basic types associated to the communication ports, as well as the dependencies between them, while checking that their usage is consistent throughout.

In order to extract the type of a base component we need to define two auxiliary functions and to introduce some notations. First, we assume that we can infer the type of each function $f(\tilde{x})$ used in a local binder. Second, given a local binder $y = f(\tilde{x}) < \tilde{\sigma}$, we need to count the number of values that y has available at runtime for each of the ports in $\tilde{x}$. This corresponds to the number of elements in $\tilde{\sigma}$ that have a mapping for a port x to a value, which we denote by *count*$(x, \tilde{\sigma})$ defined as follows. Let X be the set of ports and Σ a set whose elements are the lists of mappings from ports to values. Then function *count* : $X \times \Sigma \to \mathbb{N}_0$ is defined as follows:

$$\mathit{count}(x,\tilde{\sigma}) = \begin{cases} j & \text{if } \tilde{\sigma} = \sigma_1, \dots, \sigma_j, \sigma_{j+1}, \dots, \sigma_l \wedge \\ & x \in \bigcap_{1 \le i \le j} \mathsf{dom}(\sigma_i) \ \wedge x \notin \bigcup_{j+1 \le i \le l} \mathsf{dom}(\sigma_i) \\ 0 & \text{otherwise} \end{cases}$$

Notice that mappings in $\tilde{\sigma}$ are handled following a FIFO discipline, so the first (oldest) mappings are the ones that need to be accounted for. Finally, we introduce the notation $\gamma(\cdot)$ to represent a mapping from basic elements (such as values, ports, or functions) to their respective types. We also use γ for lists of elements in which case we obtain the list of respective types (e.g., $\gamma(1, \texttt{hello}) = \mathit{integer}, \mathit{string}$).

We now define our type extraction procedure for base components:

Definition 3.3.1 (Type Extraction for a Base Component)**.**

Let $[\tilde{x} > \tilde{y}]\{y_1 = f_{y_1}(\tilde{x}^{y_1}) < \tilde{\sigma}^{y_1}, \dots, y_k = f_{y_k}(\tilde{x}^{y_k}) < \tilde{\sigma}^{y_k}\}$ *be a base component, where* $\tilde{y} = y_1, y_2, \dots, y_k$*. If there exists* γ *such that* $\gamma(\tilde{x}) = \tilde{b}$ *and* $\gamma(y_1) = b'_1, \dots, \gamma(y_k) = b'_k$ *and provided that* $\gamma(f_{y_i}) = \tilde{b}^{y_i} \to b'_i$ *for any* $i \in 1, \dots, k$ *and that* $\tilde{b}^{y_i} = \gamma(\tilde{x}^{y_i})$ *for any* $i \in 1, \dots, k$ *then the extracted type of the base component is* $< X_b; \boldsymbol{C} >$ *where*

$$X_b = \{x(b) \mid x \in \tilde{x} \wedge b = \gamma(x)\}$$

and

$$\boldsymbol{C} = \{y_i(b'_i) : \infty : \boldsymbol{D}_{y_i} \mid i \in 1, \dots, k \wedge$$

$$b'_i = \gamma(y_i) \wedge \boldsymbol{D}_{y_i} = \{x : \mathit{count}(x, \tilde{\sigma}^{y_i}) \mid x \in \tilde{x}^{y_i}\}\}$$

In Definition 3.3.1 the list of local binders is specified in such a way that each function (f_{y_i}), its parameters ($\tilde{x}^{y_i}$) and the list of mappings ($\tilde{\sigma}^{y_i}$) are indexed with the output port that is associated to them (y_i), so as to allow for a direct identification. Moreover, notice that each list of arguments $\tilde{x}^{y_i}$ (of function f_{y_i}) is a permutation of list $\tilde{x}$, as otherwise they would be undefined.

Notice also that every output port of the interface of the component has a local binder associated to it and that there is no local binder $y_t = f_{y_t}(\tilde{x}^{y_t}) < \tilde{\sigma}^{y_t}$ such that y_t is not part of the component interface, i.e., we do not type components that have undefined output ports or that declare unused local binders, respectively. We also rely in Definition 3.3.1 on (the existence of) γ to ensure consistency. Namely, we consider γ provides the list of basic types for the input ports ($\gamma(\tilde{x}) = \tilde{b}$) and for the output ports ($\gamma(y_1) = b'_1, \ldots, \gamma(y_k) = b'_k$). Then, we require that $\gamma(f_{y_i})$, for each f_{y_i}, specifies the function type where the return type matches the one identified for y_i (i.e., b'_i). Furthermore, we require that the types of the parameters given in the function type ($\tilde{b}^{y_i}$) match the ones identified for the respective (permutation of) input port parameters ($\gamma(\tilde{x}^{y_i})$).

We then have that the extracted type of a base component is a composition of two elements. The first one (X_b) is a set of input ports which are associated with their basic types. The second one is a set of constraints **C**, one for each output port and of the form $y_i(b'_i) : \infty : [\mathbf{D}_{y_i}]$. The constraint specifies the basic type (b'_i) which is associated to the output port, and the maximum number of values that can be output on y_i is unbounded (∞), since local binders can potentially perform computations indefinitely.

The third element of the constraint ($\mathbf{D}_{y_i}$) is a set of per-each-value dependencies (of port y_i) on the input port parameters $\tilde{x}^{y_i}$, capturing that each value produced on y_i depends on a value being received on all of the ports in $\tilde{x}^{y_i}$. Notice that the number of values that y_i has available (at runtime) for each x in $\tilde{x}^{y_i}$ is given by $\mathit{count}(x, \sigma^{y_i})$.

From an operational perspective, Definition 3.3.1 can be implemented by first considering the type inferred for the functions in the local binders and then propagating (while ensuring consistency of) this information.

Example 3.3.1. *Consider our running example from Section 3.1, in particular, component K_{Portal} specified as*

$$[x_p, x'_p \rangle\, y_p, y'_p, y''_p]\{y_p = f_u(x_p) < \tilde{\sigma}^{y_p}, y'_p = f_r(x'_p) < \tilde{\sigma}^{y'_p}, y''_p = f() < \cdot\}.$$

Let us take γ such that $\gamma(x_p, x'_p) = i, c$ and $\gamma(y_p) = i$, $\gamma(y'_p) = c$ and $\gamma(y''_p) = v$. We know that function f_u takes an image (i) and gives an image in return, hence $\gamma(f_u) = i \to i$. Similarly, we also know that function f_r is typed as $\gamma(f_r) = c \to c$. Function f does not have any parameters hence $\gamma(time) = () \to v$. The extracted set of input ports with their types is $X_b = \{x_p(i), x'_p(c)\}$. Assume that the component is in the initial (static) state, so the queues of lists of mappings are empty (i.e., $\tilde{\sigma}^{y_p} = \cdot = \tilde{\sigma}^{y'_p}$). Hence, we have that $count(x_p, \tilde{\sigma}^{y_p}) = 0$ and $count(x'_p, \tilde{\sigma}^{y'_p}) = 0$. The extracted set of constraints is

$$\boldsymbol{C} = \{y_p(i)\!:\!\infty\!:\![\{x_p\!:\!0\}], y'_p(c)\!:\!\infty\!:\![\{x'_p\!:\!0\}], y''_p(v)\!:\!\infty\!:\![\emptyset]\}$$

and the extracted type of the component K_{Portal} is $< X_b; \boldsymbol{C} >$.

3.4 EC type extraction for composite components

Extracting the type of a composite component is more challenging than for a base component. The focus of the extraction procedure is on the interfacing subcomponent, which interacts both via forwarders and via the protocol.

For the purpose of characterising how components interact in a given protocol, we introduce local protocols LP which result from the projection of a (global) protocol to a specific role that is associated to a component. We reuse the projection operation from [5, 23], where message labels are mapped to communication ports (thanks to distribution binders D) and also to basic types that describe the communicated values (that can be inferred from the ones of the ports). The syntax of local protocols LP is:

$$LP := x?(b).LP \mid y!(b).LP \mid \mu\mathbf{X}.LP \mid \mathbf{X} \mid \mathbf{end}.$$

Term $x?(b).LP$ denotes a reception of a value of a type b on port x, upon which protocol LP is activated. Term $y!(b).LP$ describes an output in similar lines. Then we have standard constructs for recursion and for specifying termination (**end**). Our local protocols differ from the ones used in [5] since here we only consider choice-free global protocols. To simplify the setting, we consider global protocols that have at most one recursion (consequently also the projected local protocols). We also consider that message labels can appear at most once in a global protocol specification (up to unfolding of recursion), hence also ports occur only once in projected local protocols (also up to unfolding).

We omit the definition of projection and present the intuition via an example.

Example 3.4.1. *Let G be the (one-shot) protocol*

$$G = Portal \xrightarrow{image} RE; RE \xrightarrow{class} Portal; \mathbf{end}$$

from Section 3.1 and let $\gamma(image, class) = i, c$ be a function that given a list of a message labels returns a list of their types. Then, the projection of protocol G to role $Portal$, denoted by $G \downarrow_{Portal}$ is protocol

$$y_p!(i).x'_p?(c).\mathbf{end}$$

and the projection of G to role RE is local protocol

$$x_{re}?(i).y_{re}!(c).\mathbf{end}$$

where ports $x'_p, y_p, x_{re}, y_{re}$ are obtained via distribution binders $RE.x_{re} \xleftarrow{image} Portal.y_p, Portal.x'_p \xleftarrow{class} RE.y_{re}$. Essentially, the local protocol of $Portal$ describes that first it emits an image on y_p and then receives a classification on x'_p, and the local protocol of RE says that it first receives an image on x_{re} and then outputs a result of a classification on y_{re}.

We introduce some notation useful for the definition of the type extraction for composite components. We use the language context for local protocols (excluding recursion), denoted by $\mathcal{C}$, so as to abstract from the entire local protocol and focus on specific parts and we define it as:

$$\mathcal{C}[\,\cdot\,] ::= x? : b.\mathcal{C}[\,\cdot\,] \mid y! : b.\mathcal{C}[\,\cdot\,] \mid \cdot\,.$$

We denote the set of ports appearing in a local protocol by $fp(LP)$ and by $rep(LP)$ the set of ports that occur in a recursion (e.g. in LP for recursion $\mu\mathbf{X}.LP$). Considering a list of forwarders F, we define two sets: by F^i we denote the set of (internal) input ports and by F^o the set of (internal) output ports which are specified in F (e.g., if $F = x_p \leftarrow x$ then $F^i = \{x_p\}$).

We now introduce the important notions that are used in our type extraction, namely that account for *values flowing* in a protocol and for the *kinds of dependencies* involved in composite components. Finally, we address the *boundaries* for the output ports.

Values flowing Our types track the dependencies between output and input ports, including per-each-value dependencies that specify how many values received on the input port are available to the output port. As discussed in the previous section, for base components this counter is given by the number of values available in the local binder queues. For composite components, as preliminary discussed in Section 3.1, per-each-value dependencies might actually result from a chain of dependencies that involve subcomponents and the protocol. So, in order to count how many values are available in such case, we need to take into account how many values are in the subcomponents (which is captured by their types) and also if a value is *flowing* in the protocol. We can capture the fact that a value is flowing by inspecting the structure of the protocol. In particular we are interested in values that flow from y to x when an output on y precedes an input in x in a recursive protocol, hence when the protocol is of the form $\mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[y!(b').\mathcal{C}''[x?(b).LP']]]$. The value is flowing when the output has been carried out but the input is yet to occur, which we may conclude if the protocol is *also* of the form $\mathcal{C}'''[x?(b).LP'']$ where $x, y \notin \mathit{fp}(\mathcal{C}'''[\,\cdot\,])$. We denote by $\mathit{vf}(LP, x, y)$ that there is a value flowing from y to x in LP, in which case $\mathit{vf}(LP, x, y) = 1$, otherwise $\mathit{vf}(LP, x, y) = 0$. We will return to this notion in the context of the extraction of the dependencies of the output ports, discussed next.

3.4.1 Dependencies extraction

Composite components comprise two kinds of dependencies between output ports and input ports, illustrated in Figure 3 and Figure 4, which are dubbed direct and transitive, respectively.

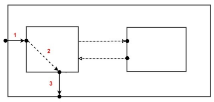

Figure 3: Direct Dependency

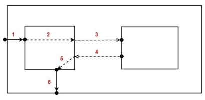

Figure 4: Transitive Dependency

Direct dependencies

We gather the set of direct dependencies, i.e., when external output ports directly depend on external input ports (see Figure 3), in $\mathbf{D}_d(\mathbf{C}, F, y)$

which is defined as follows:

$$\mathbf{D}_d(\mathbf{C}, F, y) \triangleq \{x\!:\!M \mid \mathbf{C} = \{y(b^y)\!:\!\mathbf{B}\!:\![\{x\!:\!M\}\uplus\mathbf{D}]\}\uplus\mathbf{C}' \wedge\, x \in F^i \wedge y \in F^o\}$$

Hence, in $\mathbf{D}_d(\mathbf{C}, F, y)$ we collect all the dependencies for y given in (internal) constraint **C** whenever both ports are external and preserving the kind of dependency M so as to lift it to the outer interface.

Transitive dependencies

For transitive dependencies (see Figure 4) to exist there are three necessary conditions. The first condition is to have in the description of a local protocol at least one output action, say on port y', that precedes at least one input action, say on port x'. The second condition is that such output port y' depends on some external input port x and the third condition is that there exists some external output port y that depends on the input on x'. In such cases, we say that y depends on x in a transitive way.

We introduce a relation that allows to capture the first condition above. Let LP be the local protocol that is prescribed for an interfacing component. Two ports x' and y' are in relation $\diamond_i^{LP}$ for some local protocol LP if $x', y' \in \mathit{fp}(LP)$ and where $i \in \{1, 2, 3\}$ as follows:

1. $y' \diamond_1^{LP} x'$ if $LP = \mathcal{C}[y'!(b^{y'}).\mathcal{C}'[x'?(b^{x'}).LP']]$ and $x', y' \notin \mathit{rep}(LP)$;
2. $y' \diamond_2^{LP} x'$ if $LP = \mathcal{C}[y'!(b^{y'}).\mathcal{C}'[\mu\mathbf{X}.\mathcal{C}''[x'?(b^{x'}).LP']]]$ and $y' \notin \mathit{rep}(LP)$;
3. $y' \diamond_3^{LP} x'$ if $LP = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[y'!(b^{y'}).\mathcal{C}''[x'?(b^{x'}).LP']]]$.

We distinguish three cases: when both the output and the input are non-repetitive, when only the input is repetitive, and when both the output and the input are repetitive.

We may now characterise the transitive dependencies. Let $[\tilde{x}' \rangle\, \tilde{y}']\{G; r = K, R; D; r[F]\}$ be a composite component, $T_r =< X_b, \mathbf{C} >$ the type of interfacing component K and LP its local protocol.

The set of transitive dependencies on y, denoted $\mathbf{D}_t(\mathbf{C}, F, LP, y)$, is defined relying on an abbreviation η as follows:

$$\eta \triangleq \quad \mathbf{C} = \{y(b_1)\!:\!\mathbf{B}\!:\![\{x'\!:\!M'\} \uplus \mathbf{D}'], y'(b_2)\!:\!\mathbf{B}'\!:\![\{x\!:\!M\} \uplus \mathbf{D}]\} \uplus \mathbf{C}'$$

$$\wedge\, x \in F^i \wedge y \in F^o \wedge y' \diamond_i^{LP} x'$$

$$\mathbf{D}_t(\mathbf{C}, F, LP, y) \triangleq E \cup S \cup V$$

$$\begin{aligned}
&E = \big\{x : \Omega \mid \eta \wedge i \in \{1, 2\} \wedge M \not> 0\}\\
&S = \big\{x : \Omega \mid \eta \wedge i = 3 \wedge (M = \Omega \vee (M' = \Omega \wedge M = 0 \wedge\\
&\qquad vf(LP, x', y') = 0))\big\}\\
&V = \big\{x : (N + N' + vf(LP, x', y')) \mid \eta \wedge i = 3 \wedge M = N \wedge M' = N'\big\}
\end{aligned}$$

In η we gather a conjunction of conditions that must always hold in order for a transitive dependency to exist: namely that the (internal) constraint **C** specifies dependencies between y and x' and between y' and x and also that y and x are external ports while y' precedes x' in the protocol. To simplify presentation of the definition of $\mathbf{D}_t(\mathbf{C}, F, LP, y)$ we rely on the (direct) implicit matching in η of the several mentioned elements.

There are two kinds of transitive dependencies that are gathered in $\mathbf{D}_t(\mathbf{C}, F, LP, y)$, namely initial ($x : \Omega$) and per-each-value ($x : N$). For initial dependencies there are two separate cases to consider. The first case is when the protocol specifies that the output on y' is non-repetitive ($i \in \{1, 2\}$), hence will be provided only once. Condition $M \not> 0$ says that no values are already available for that initial output to take place (internally to the component that provides them as specified in η), hence either $M = x : \Omega$ or $M = 0$.

The second case for an initial transitive dependency is when both y' and x' are repetitive in the protocol ($i = 3$) but at least one of the internal dependencies (between y' and x and between y and x', given by M and M' respectively) is an initial dependency. This means that, regardless of the protocol, such a dependency is dropped as soon as a value is provided which implies that the transitive dependency is also dropped. Since M is at the beginning of the dependency chain, if it is initial then no further conditions are necessary. However, if M' is initial we need to ensure that there is no value already flowing ($vf(LP, x', y') = 0$) or already available to be output on y' ($M = 0$), since only in such case (an initial) value is required from the external context (i.e., otherwise if $vf(LP, x', y') = 1$ or $M \geq 0$ then the chain of dependencies is already "internally" satisfied).

Finally, we have the case of per-each-value transitive dependency, that can only be when both y' and x' are repetitive in the protocol ($i = 3$) and internal dependencies M and M' are both per-each-value dependencies ($M = N$ and $M' = N'$), which means that the dependency chain is persistent. The number of values available of (external) x for y is the sum of the values available in the internal dependencies (N and N') plus one if there is a value flowing (zero otherwise).

Notice that the definition of value flowing presented previously focuses exclusively in the case when y' and x' are repetitive in the protocol, since this is the only case where values might be flowing and the dependency is still present in the protocol structure (i.e., $y' \diamond_3^{LP} x'$ holds). In contrast, a dependency $y' \diamond_i^{LP} x'$ for $i \in \{1, 2\}$ is no longer (structurally) present as soon as the value is flowing (i.e., a non-repetitive y' no longer occurs in the protocol after an output).

It might be the case that one output port depends in multiple ways on the same input port. For that reason we introduce a notion of *priority* among dependencies, denoted by **pr**(,) that gives priority to per-each-value dependencies (with respect to "initial").

The priority builds on the property that if multiple per-each-value dependencies (including direct and transitive) are collected (e.g., $x : N_1, \ldots, x : N_k$) then the number of available values specified in them is the same (i.e., $N_1 = \ldots = N_k$). The list of dependencies for port y is then given by the (prioritised) union of direct and transitive dependencies:

$$\mathbf{D}(\mathbf{C}, F, LP, y) = \mathbf{pr}(\mathbf{D}_d(\mathbf{C}, F, y) \cup \mathbf{D}_t(\mathbf{C}, F, LP, y))$$

3.4.2 Boundaries extraction

The last element that we need to determine in order to extract the type of a composite component is the boundary of output ports. The type of the interfacing component already specifies a (internal) boundary, however this value may be further bound by the way in which the component is used in the composition. In particular, if an output port depends on input ports that are not used in the protocol nor are linked to external ports, then no (further) values are received in them and the potential for the output port is consequently limited. We distinguish three cases for three possible limitations:

$$B_1 = \{N' \mid \mathbf{C} = y(b_1) : \mathbf{B} : [\{x' : N'\} \uplus \mathbf{D}'] \uplus \mathbf{C}' \wedge x' \notin (\mathit{fp}(LP) \cup F^i)\}$$

$$B_2 = \{0 \mid \mathbf{C} = \{y(b_1) : \mathbf{B} : [\{x' : \Omega\} \uplus \mathbf{D}']\} \uplus \mathbf{C}' \wedge x' \notin (\mathit{fp}(LP) \cup F^i)\}$$

$$\begin{aligned} B_3 = \{(N' + 1) \mid \mathbf{C} = \{y(b_1) : \mathbf{B} : [\{x' : N'\} \uplus \mathbf{D}']\} \uplus \mathbf{C}' \wedge x' \in \mathit{fp}(LP) \\ \wedge x' \notin (\mathit{rep}(LP) \cup F^i)\} \end{aligned}$$

In B_1 and B_2 we capture the case when there is a dependency on a port that is not used in the protocol ($x' \notin fp(LP)$) nor linked externally ($x' \notin F^i$), where the difference is in the kind of dependency. For per-each-value dependencies (if any), the minimum of the internally available values is identified as the potential boundary, while for initial dependencies (if present) the potential boundary is zero (or the empty set).

In B_3 we capture a case similar to B_1 where the port is used in the protocol but in a non-repetitive way, hence only one (further) value can be provided.

The final boundary determined for y, denoted by $\mathbf{B}(y, LP, \mathbf{C})$, is the minimum number among the internal boundary of y (i.e., $\mathbf{B}$ if $\mathbf{C} = y(b) : \mathbf{B} : [\mathbf{D}] \uplus \mathbf{C}'$) and possible boundaries B_1, B_2 and B_3 described above.

$$\mathbf{B}(y, LP, \mathbf{C}) = min(\{\mathbf{B}\} \cup B_1 \cup B_2 \cup B_3)$$

3.4.3 Type extraction

We may now present the definition of type extraction of a composite component relying on a renaming operation. Since the type extraction of a composite component focuses on the interfacing subcomponent, we single out the ports that are linked via forwarders to the external environment. To capture such links, we introduce renaming operation **ren**(,) that renames the ports of the interfacing subcomponent to the outer ones by using the forwarders as a guideline. For example, if we have that $F = x_p \leftarrow x$ than $\mathbf{ren}(F, x_p) = x$.

Definition 3.4.1 (Type Extraction for a Composite Component). *Let $[\tilde{x} \rangle \tilde{y}]\{G; r = K, R; D; r[F]\}$ be a composite component and $LP = G \upharpoonright_r$ the local protocol for component K. If $T_r =< X_b^r; \boldsymbol{C}^r >$ is the type of component K, then the extracted type from LP and T_r is*

$$T(LP, T_r, F) = \boldsymbol{ren}(F, < X_b; \boldsymbol{C} >)$$

where: $X_b = \{x(b) \mid x(b) \in X_b^r \wedge x \in F^i\}$

$\boldsymbol{C} = \{y(b') : \boldsymbol{B}(y, LP, \boldsymbol{C}^r) : [\boldsymbol{D}(\boldsymbol{C}^r, F, LP, y)] \mid \boldsymbol{C}^r = \{y(b') : \boldsymbol{B}' : [\boldsymbol{D}']\} \uplus \boldsymbol{C}' \wedge y \in F^o\}$.

Example 3.4.2. *Let us extract the type of component K_{IRS} from Section 3.1 considering protocol $G = Portal \xrightarrow{image} RE; RE \xrightarrow{class} Portal;$ **end**. The type of interfacing component K_{Portal} is*

$T_{Portal} =< X_b; \boldsymbol{C} >$
$X_b =< \{x_p(image), x'_p(class)\}$
$\boldsymbol{C} = \{C_1, C_2, C_3\}$
$C_1 = y_p(image) : \infty : [\{x_p : N_p\}]$
$C_2 = y'_p(class) : \infty : [\{x'_p : N'_p\}]$
$C_3 = y''_p(version) : \infty : [\emptyset]$

*We have that local protocol is $LP = y_p!(i).x'_p?(c).\mathbf{end}$ and sets of external ports $F^i = \{x_p\}$ and $F^o = \{y'_p, y''_p\}$, where **ren**$(F, x_p) = x$, **ren**$(F, y'_p) = y$ and **ren**$(F, y''_p) = y'$. This immediately gives us the set of input ports that is in the description of the type of component K_{IRS} which is $X_b = \{x(i)\}$.*

*Let us now determine the constraints of the output ports. Since port y''_p has no dependencies also port y' will not have any, and moreover has the same boundary (∞). So, the extracted constraint for y' will be **ren**$(y''_p(v) : \infty : [\emptyset])$, which is $y'(v) : \infty : [\emptyset]$. Port y'_p instead depends on port x'_p which is used in the protocol ($x'_p \in fp(LP)$). Since the protocol is not recursive we have the consequent limited boundary (case B_3 explained above), namely the boundary of y'_p is $min(N'_p + 1, \infty) = N'_p + 1$. Furthermore. we have that $y_p \diamond_1^{LP} x'_p$ and that y_p has per-each-value dependency $x_p : N_p$. If $N_p > 0$ then y'_p does not transitively depend on x_p, otherwise there is an initial dependency. Let us consider the initial (static) state where no image has been receive yet, i.e., $N_p = 0$. In such case we have that the resulting constraint for y'_p is $y'_p(c) : N'_p : [x_p : \Omega]$, which after renaming for y is $y(c) : N'_p : [x_p : \Omega]$. So, the extracted type of K_{IRS} is the following*

$$< \{x(i)\}; \{y(c) : N'_p : [x_p : \Omega], y'(v) : \infty : [\emptyset]\} >$$

3.5 Type safety (EC type language)

In this section we present our main results that show a tight correspondence between the behaviour of components and of their extracted types. Apart from the conditions already involved in the type extraction, for a component to be well-typed we must also ensure that any component that interacts in a protocol can actually carry out the communication actions prescribed by the protocol.

For this reason we introduce the conformance relation, denoted by $\bowtie$, that asserts the compatibility between the type of a component and the local protocol that describes the communication actions prescribed for the component. For the purpose of ensuring compatibility, in particular for the interfacing component, we also introduce an extension of our type language, dubbed modified types $\mathcal{T}$. The idea for modified types is to allow to abstract from input dependencies from the external environment, namely by considering such dependencies can (always) potentially be fulfilled, allowing conformance to focus on internal compatibility.

3.5.1 Modified type

Now we introduce the modified type denoted by $\mathcal{T}$. The interfacing component of the composite one, beside its interaction with other components, also interacts with an external environment. In this case the crucial part is that it is able to receive in any moment values that are input externally. For the purpose of observing if a type of a component can perform actions required by the protocol, we need to modify the type according to the possible inputs that a (interfacing) component can receive from the external context without any constraints. The modified type of a type T, taking into account the list of corresponding list of forwarders, is denoted by $\mathcal{T}(F, T)$. If T is the type of the interfacing component, each dependency on the external input ports is per-each-value dependency and the number of values available is unbounded (assuming that whenever the value is available it is received on the external input ports). The syntax of $\mathcal{T}$-type is given in the Table 10. It is similar to the syntax of the types which we have already shown, with the difference in the number of values received, that in the modified type can be unbounded (infinite). Moreover, the rules defining the semantics of modified type are the same as the ones shown for our typing language (Table 11), shown in Subsection 3.2.2. The the only implicit difference for modified types is that decrementing an unbounded dependency has no effect.

In order to get the modified type out of the interfacing component we use the following definition:

Definition 3.5.1. *If $T_r =< X_b; \boldsymbol{C} >$ is a type of interfacing subcomponent $\overline{K}$ of composite component $[\tilde{x} > \tilde{y}]\{G; r = \overline{K}, R; D; r[F]\}$ then $\mathcal{T}(F, T_r)$ is the T_r-modified type where:*

$\mathcal{T}(F, < X_b, \boldsymbol{C} >)$	$\triangleq$	$< X_b; \mathcal{T}(F, \boldsymbol{C}) >$
$\mathcal{T}(F, \{y(b) : \boldsymbol{B} : [\boldsymbol{D}]\} \uplus \boldsymbol{C})$	$\triangleq$	$\mathcal{T}(F, \{y(b) : \boldsymbol{B} : [\boldsymbol{D}]\}) \uplus \mathcal{T}(F, \boldsymbol{C})$
$\mathcal{T}(F, \{y(b) : \boldsymbol{B} : [\boldsymbol{D}]\})$	$\triangleq$	$\{y(b) : \boldsymbol{B} : [\mathcal{T}(\boldsymbol{D})]\}$
$\mathcal{T}(F, \{x : M\} \uplus \boldsymbol{D})$	$\triangleq$	$\mathcal{T}(F, \{x : M\}) \uplus \mathcal{T}(\boldsymbol{D})$ [*where* $M \in \{N, \Omega\}$]
$\mathcal{T}(F, x : M)$	$\triangleq$	$\{x : M\}$ [*if* $x \notin F^i$, *where* $M \in \{N, \Omega\}$]
$\mathcal{T}(F, x : M)$	$\triangleq$	$\{x : \infty\}$ [*if* $x \in F^i$]
$\mathcal{T}(F, x : \Omega)$	$\triangleq$	$\emptyset$ [*if* $x \in F^i$]

Note that for $K = [\tilde{x} > \tilde{y}]\{G, r_1 = K_1, r_2 = K_2, \ldots, r_n = K_n; D; r_1[F]\}$ we have that

$$\mathcal{T}(F, T_{r_2}) = T_{r_2}, \ldots, \mathcal{T}(F, T_{r_k}) = T_{r_k}$$

since the only component that forwards the values from/to external environment is component K_1.

Types	$\mathcal{T} \triangleq < X_b; \mathcal{C} >$
	$X_b \triangleq \{x_1(b_1), \ldots, x_k(b_k)\}$
Constraints	$\mathcal{C} \triangleq \{y_1(b_1) : \mathbf{B}_1 : [\mathcal{D}_1], \ldots, y_k(b_k) : \mathbf{B}_k : [\mathcal{D}_k]\}$
Dependencies	$\mathcal{D} \triangleq \{x_1 : \mathcal{M}_1, \ldots, x_k : \mathcal{M}_k\}$
Kinds of dependencies	$\mathcal{M} ::= \mathcal{N} \mid \Omega$
	$\mathcal{N} ::= N \mid \infty$
Boundaries	$\mathbf{B} ::= N \mid \infty$
Additional values	$k \geq 0; N \in \mathbb{N}_0$

Table 10: $\mathcal{T}$-Type syntax (EC type language)

$$\frac{x \notin dom[\mathcal{D}]}{y(b)\!:\!\mathbf{B}\!:\![\mathcal{D}] \xrightarrow{x?} y(b)\!:\!\mathbf{B}\!:\![\mathcal{D}]}\ [\mathcal{T}1]$$

$$\frac{}{y(b')\!:\!\mathbf{B}\!:\![\{x:\Omega\} \uplus \mathcal{D}] \xrightarrow{x?} y(b')\!:\!\mathbf{B}\!:\![\mathcal{D}]}\ [\mathcal{T}2]$$

$$\frac{}{y(b')\!:\!\mathbf{B}\!:\![\{x\!:\!\mathcal{N}\} \uplus \mathcal{D}] \xrightarrow{x?} y(b')\!:\!\mathbf{B}\!:\![\{x\!:\!\mathcal{N}+1\} \uplus \mathcal{D}]}\ [\mathcal{T}3]$$

$$\frac{}{\mathcal{T} \xrightarrow{\tau} \mathcal{T}}\ [\mathcal{T}4]$$

$$\frac{\forall i \in 1,2,\ldots,k \quad y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathcal{D}_i] \xrightarrow{x?} y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathcal{D}'_i]}{\begin{array}{c}<\{x(b^x) \uplus X_b\}; \{y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathcal{D}_i] | i \in 1,\ldots,k\}> \xrightarrow{x?(b^x)} \\ <\{x(b^x) \uplus X_b\}; \{y_i(b_i)\!:\!\mathbf{B}_i\!:\![\mathcal{D}'_i] | i \in 1,\ldots,k\}>\end{array}}\ [\mathcal{T}5]$$

$$\frac{\mathbf{B} > 0 \quad \mathcal{N}_i \geq 1}{\begin{array}{c}< X_b; \{y(b^y)\!:\!\mathbf{B}\!:\![\{x_i\!:\!\mathcal{N}_i | i \in 1,\ldots,k\}]\} \uplus \mathcal{C} > \xrightarrow{y!(b^y)} \\ < X_b; \{y(b^y)\!:\!\mathbf{B}-1\!:\![\{x_i\!:\!\mathcal{N}_i - 1 | i \in 1,\ldots,k\}]\} \uplus \mathcal{C} >\end{array}}\ [\mathcal{T}6]$$

Table 11: $\mathcal{T}$ - Semantics (EC type language)

Conformance relation

The definition of the conformance relation is given by induction on the structure of the local protocol and it is characterised by judgements of the form $\Gamma \vdash \mathcal{T} \bowtie LP$, where Γ is a type environment that handles protocol recursion (i.e., Γ maps recursion variables to modified types).

$$\frac{\mathcal{T} \xrightarrow{x?(b)} \mathcal{T}' \; \Gamma \vdash \mathcal{T}' \bowtie LP}{\Gamma \vdash \mathcal{T} \bowtie x?(b).LP} \; [InpConf]$$

$$\frac{\mathcal{T} \xrightarrow{y!(b)} \mathcal{T}' \quad \Gamma \vdash \mathcal{T}' \bowtie LP}{\Gamma \vdash \mathcal{T} \bowtie y!(b).LP} \; [OutConf]$$

$$\frac{}{\Gamma \vdash \mathcal{T} \bowtie \mathbf{end}} \; [EndConf] \qquad \frac{\mathcal{T}' \leq \mathcal{T}}{\Gamma, X : \mathcal{T}' \vdash \mathcal{T} \bowtie X} \; [VarConf]$$

$$\frac{\Gamma, X : \mathcal{T} \vdash \mathcal{T} \bowtie LP}{\Gamma \vdash \mathcal{T} \bowtie \mu\mathbf{X}.LP} \; [RecConf]$$

Table 12: Conformance relation (EC type language)

Definition 3.5.2. *$\mathcal{T}' \leq \mathcal{T}$ if exists a (possibly empty) set of typed input ports $\{x_1(b_1), x_2(b_2), \ldots, x_k(b_k)\}$ such that $\mathcal{T}' \xrightarrow{x_1?(b_1)} \cdots \xrightarrow{x_k?(b_k)} \mathcal{T}$.*

Rule $[InpConf]$ ensures that a modified type $\mathcal{T}$ is conformant with the protocol, where it can receive an input of a matching type with a continuation as a protocol LP, if a modified type can receive a value on port x, and assuming that port x receives a value of type b and the evolved type is conformant with LP. Similar reasoning is for an output. Rule $[EndConf]$ states that a modified type is always conformant with the termination protocol. Finally, we have two rules $[VarConf]$ and $[RecConf]$ for the recursion. The premise of Rule $[VarConf]$ requires that the type associated with the recursion variable by assumption and the type under consideration are related as $\mathcal{T}' \leq \mathcal{T}$ (Definition 3.5.2). Observing the semantics of modified types, the possible difference between types $\mathcal{T}'$ and $\mathcal{T}$ is that some initial dependencies might be dropped or that the number of values available on some input ports for some outputs might increase. Rule $[RecConf]$ states that $\mathcal{T}$ is conformant with a protocol $\mu\mathbf{X}.LP$, provided that the type is conformant with the body of the recursion under the environment extended with assumption $\mathbf{X} : \mathcal{T}$.

3.5.2 Well-typed components

We can now formally define when a component K has type T, in which case we say K is well-typed.

Definition 3.5.3. *Let K be a component, we say that K has a type T, denoted by $K \Downarrow T$:*

1. *If K is a base component, $K \Downarrow T$ when T is obtained by Definition* 3.3.1.
2. *If $K = [\tilde{x} > \tilde{y}]\{G; r_1 = K_1, \ldots, r_k = K_k; D; r_1[F]\}$ then $K \Downarrow T$ when*
   - $\exists T_{r_i} \mid K_i \Downarrow T_{r_i}$*,for $i = 1, 2, \ldots, k$;*
   - *T is extracted type from T_1 and $G \downharpoonleft_{r_1}$ by Definition* 3.4.1*;*
   - $\mathcal{T}(F, T_{r_i}) \bowtie G \downharpoonleft_{r_i}$ *for $i = 1, 2, \ldots, k$;*

Notice that the definition relies on modified types for conformance, but for any type T not associated with the interfacing component we have that $\mathcal{T}(F, T) = T$ since there can be no links to external ports (assuming that all ports have different identifiers).

We can now present our type safety results given in terms of Subject Reduction and Type fidelity, which provide the correspondence between the behaviours of well-typed components and their types.

In the statements we rely on notation $\lambda(v)$ that represents $x?(v)$, $y!(v)$ or τ and $\lambda(b)$ that represents $x?(b)$, $y!(b)$ or τ.

Theorem 3.5.1 (Subject Reduction)**.** *If $K \Downarrow T$ and $K \xrightarrow{\lambda(v)} K'$ and v has type b then $T \xrightarrow{\lambda(b)} T'$ and $K' \Downarrow T'$.*

Theorem 3.5.1 says that if a well-typed component K performs a computation step to K', then its type T can also evolve to type T' which is the type of component K'. Moreover, the theorem ensures that if K carries out an input or an output of a value v, type T performs the corresponding action at the level of types.

Theorem 3.5.1 thus attests that well-typed components always evolve to well-typed components, and furthermore that any component evolution can be described by an evolution in the types.

The type fidelity result does not describe a strong correspondence like for Subject Reduction since we need to abstract from internal computations in components. For that reason, in the Type fidelity statement we rely on $K \xRightarrow{\lambda(v)} K'$ to denote a sequence of transitions $K \xrightarrow{\tau} \cdots K'' \xrightarrow{\lambda(v)} K''' \xrightarrow{\tau} \cdots K'$, i.e, that component K may perform a sequence of internal moves, then an I/O action, after which another sequence of internal moves leading to K'.

Theorem 3.5.2 (Type fidelity). *If $K \Downarrow T$ and $T \xrightarrow{\lambda(b)} T'$ and $\lambda(b) \neq \tau$ then b is the type of a value v and $K \xRightarrow{\lambda(v)} K'$ and $K' \Downarrow T'$.*

Theorem 3.5.2 says that if type T of component K can evolve by exhibiting an I/O action to type T', then K can eventually (up to carrying out some internal computations) exhibit a corresponding action leading to K', and where K' has type T'. Theorem 3.5.2 thus ensures that the behaviours of types can eventually be carried out by the respective components, which entails components are not stuck and allows, together with Theorem 3.5.1, to attest that types faithfully capture component behaviour. Intuitively, our types can be seen as *promises of behaviour* in the sense that whatever they prescribe as possible behaviours, the components will eventually deliver. For the sake of addressing any possible component configuration, in particular when components have already all the dependencies (internally) satisfied in order to provide some behaviour, it is crucial that types capture the number of (internally) available resources.

3.6 Proof of type safety (EC type language)

First, we present some auxiliary results that simplify our proofs.

The first proposition ensures that if a local binder $y = f(\tilde{x}) < \tilde{\sigma}$ performs an input from a port x that is used as argument of f ($x \in \tilde{x}$), then the number of values of x available for y increases by one. Otherwise, if $x \notin \tilde{x}$, the number of available value for y remains unchanged.

Proposition 3.6.1. *Let L be a local binder $y = f(\tilde{x}) < \tilde{\sigma}, L_1$, if $L \xrightarrow{x?(v)} L'$ where $L' = y = f(\tilde{x}) < \tilde{\sigma}', L_1'$ then:*

$$count(x, \tilde{\sigma}') = \begin{cases} count(x, \tilde{\sigma}) + 1 & \textit{if } x \in \tilde{x} \\ count(x, \tilde{\sigma}) & \textit{otherwise} \end{cases}$$

Moreover, $\tilde{\sigma}' = \tilde{\sigma}$ when $x \notin \tilde{x}$.

Proof. Proof by induction on the derivation of $L \xrightarrow{x?(v)} L'$ and by cases on the last rule applied:

> [LInpDisc] $y = f(\tilde{x}) < \tilde{\sigma} \xrightarrow{x?(v)} y = f(\tilde{x}) < \tilde{\sigma}$, by inversion we know that $x \notin \tilde{x}$ so the property holds.
>
> [LInpNew] $y = f(\tilde{x}) < \tilde{\sigma} \xrightarrow{x?(v)} y = f(\tilde{x}) < \tilde{\sigma}, \{x \to v\}$, by inversion we know that $x \in \cap_{\sigma_i \in \tilde{\sigma}} dom(\sigma_i)$, $i \in \{1, 2, \ldots, n\}$. After an input on x, the number of mappings for x is increased by 1. So, the property holds.
>
> [LInpUpd] $y = f(\tilde{x}) < \tilde{\sigma_1}, \sigma, \tilde{\sigma_2} \xrightarrow{\text{x?(v)}} y = f(\tilde{x}) < \tilde{\sigma_1}, \sigma[x \to v], \sigma_2$. By inversion we know that $x \in \cap_{\sigma_i \in \tilde{\sigma_1}} dom(\sigma_i)$ and $x \in \tilde{x}$. After input on x, the number of mappings for x is increased by 1. So, the property holds.
>
> [LInpList] $L_1, L_2 \xrightarrow{\text{x?(v)}} L_1', L_2'$. By inversion we know that $L1 \xrightarrow{\text{x?(v)}} L_1'$, and by i.h. we know that for L_1' the property holds. The same reasoning for $L_2 \xrightarrow{\text{x?(v)}} L_2'$, so also for L_2' the property holds. Directly we can conclude that the property holds also for L_1', L_2'.

□

Note that the proposition above can be easily extended to lists of local binders.

The following proposition ensures that if a local binder $y = f(\tilde{x}) < \tilde{\sigma}$ performs an output, the number of values of input ports used as arguments of f decreases by one. Instead, the constant function $f() < \cdot$ remains having zero values available on each input port since it does not have any arguments.

Proposition 3.6.2. *If $L \xrightarrow{y!(v)} L'$, then*

- *if $L = y = f(\tilde{x}) < \tilde{\sigma}, L_1$ and $L' = y = f(\tilde{x}) < \tilde{\sigma'}, L_1'$, we have that*

$$count(x, \tilde{\sigma'}) = count(x, \tilde{\sigma}) - 1 \qquad \forall x \in \tilde{x}$$

- *if $L = f(\,) < \tilde{\sigma}, L_1$ and $L' = f(\,) < \tilde{\sigma'}, L_1'$, we have that*

$$count(x, \tilde{\sigma'}) = count(x, \tilde{\sigma}) = 0$$

Proof. Proof by induction on the derivation of $L \xrightarrow{y!(v)} L'$ and then by cases on the last rule applied:

[LOut] $y = f(\tilde{x}) < \sigma, \tilde{\sigma} \xrightarrow{\mathrm{y!(v)}} y = f(\tilde{x}) < \tilde{\sigma}$, so $f(\tilde{x}) < \tilde{\sigma}$ has one store (σ) less. Since one complete store contains the full mappings from x to v, where $x \in \tilde{x}$, the property holds.

[LConst] $f(\,) < \cdot \xrightarrow{\mathrm{y!(v)}} f(\,) < \cdot$, the property directly holds, since the queues of stores of mappings are empty.

[LOutLift] $L_1, L_2 \xrightarrow{\mathrm{y!(v)}} L_1', L_2$. By inversion we know that $L_1 \xrightarrow{\mathrm{y!(v)}} L_1'$ and by i.h. we know that the property holds for L_1'. If we add to the list any other functions, property will hold, because y is a function in L_1 (and also in L_1'). So the property holds also for L_1', L_2.

□

Combining the two propositions above we prove the following lemma ensuring that the behaviour of a base component is fully qualified by the behaviour of its local binders.

Lemma 3.6.1. *Let $K = [\tilde{x} > \tilde{y}]\{L\}$ be a base component. If $K \xrightarrow{\lambda(v)} K'$ for some $K' = [\tilde{x} > \tilde{y}]\{L'\}$, then*

1. *If $\lambda = x?$, we have that $L \xrightarrow{x?(v)} L' \wedge x \in \tilde{x}$;*

2. *If $\lambda = y!$, we have that $L \xrightarrow{y!(v)} L' \wedge y \in \tilde{y}$.*

Proof. Proof by induction on the derivation of $K \xrightarrow{\lambda(v)} K'$ and by cases on the last rule applied. Since K by hypothesis is a base component the only possible cases are:

[InpBase] $[\tilde{x} > \tilde{y}]\{L\} \xrightarrow{x?(v)} [\tilde{x} > \tilde{y}]\{L\}$, by inversion we know that $L \xrightarrow{x?(v)} L'$ and that $x \in \tilde{x}$.

[OutBase] $[\tilde{x} > \tilde{y}]\{L\} \xrightarrow{y!(v)} [\tilde{x} > \tilde{y}]\{L\}$, by inversion we know that $L \xrightarrow{x?(v)} L'$ and that $y \in \tilde{y}$.

□

In order to make proofs more concise we define the function **inc**. This function by given as an argument the constraint and the port on which the value is received gives us as a result (according to the semantics of the typing language from Table 9) the evolved constraint.

Definition 3.6.1. *By* $\textbf{\textit{inc}}(\textbf{\textit{C}}, x)$ *we denote the constraint defined as follows:*

$$\begin{array}{lcl}
\textbf{\textit{inc}}(\textbf{\textit{C}}_1, \textbf{\textit{C}}_2; x) & \triangleq & \textbf{\textit{inc}}(\textbf{\textit{C}}_1; x), \textbf{\textit{inc}}(\textbf{\textit{C}}_2; x) \\
\textbf{\textit{inc}}(y(b) : \textbf{\textit{B}} : [\{x : N\} \uplus \textbf{\textit{D}}]; x) & \triangleq & y(b) : \textbf{\textit{B}} : [\{x : N+1\} \uplus \textbf{\textit{D}}] \\
\textbf{\textit{inc}}(y(b) : \textbf{\textit{B}} : [\{x : \Omega\} \uplus \textbf{\textit{D}}]; x) & \triangleq & y(b) : \textbf{\textit{B}} : [\textbf{\textit{D}}] \\
\textbf{\textit{inc}}(y(b) : \textbf{\textit{B}} : [\textbf{\textit{D}}]; x) & \triangleq & y(b) : \textbf{\textit{B}} : [\textbf{\textit{D}}] \ (\textit{if } x \notin \textit{dom}(\textbf{\textit{D}}).
\end{array}$$

Proposition 3.6.3. *If* $T =< X_b, \textbf{\textit{C}} > \ \wedge \ \textbf{\textit{C}} \xrightarrow{x?} \textbf{\textit{C}}'$ *then* $\textbf{\textit{C}}' = \textbf{\textit{inc}}(\textbf{\textit{C}}, x)$.

Proof. By induction on the derivation of $\mathbf{C} \xrightarrow{x?} \mathbf{C}'$.

[T1] $\mathbf{C}_1, \mathbf{C}_2 \xrightarrow{x?} \mathbf{C}'_1, \mathbf{C}'_2$ by inversion we know that $\mathbf{C}_1 \xrightarrow{x?} \mathbf{C}'_1$ and by induction hypothesis we know that the property holds for $\mathbf{C}'_1$. Also, by inversion we know that $\mathbf{C}_2 \xrightarrow{x?} \mathbf{C}'_2$ and by i.h. property holds also for $\mathbf{C}'_2$. Directly we can conclude that the property will hold also for $\mathbf{C}'_1, \mathbf{C}'_2$.

[T6] $y(b) : \mathbf{B} : [\{x : N\} \uplus \mathbf{D}] \xrightarrow{x?} y(b) : \mathbf{B} : [\{x : N+1\} \uplus \mathbf{D}]$ we can directly conclude that the property holds.

[T5] $y(b) : \mathbf{B} : [\{x : \Omega\} \uplus \mathbf{D}] \xrightarrow{x?} y(b) : \mathbf{B} : [\mathbf{D}]$, we can directly conclude that the property holds.

[T4] $y(b) : \mathbf{B} : [\mathbf{D}] \xrightarrow{x?} y(b) : \mathbf{B} : [\mathbf{D}]$, by inversion we know that $x \notin \textit{dom}(\mathbf{D})$, so the number of received values on x remained the same, which implies that the property holds also for this case.

□

Lemma 3.6.2. *If* $T =< X_b; \textbf{\textit{C}} >$ *and* $T \xrightarrow{x?(b)} T'$ *then* $\textbf{\textit{C}} \xrightarrow{x?} \textbf{\textit{C}}'$.

Proof. By induction on the derivation of $T \xrightarrow{x?(b)} T'$.

[T5] $T =< X_b; \mathbf{C} > \xrightarrow{x?(b)} T' =< X_b; \mathbf{C}' >$, by inversion we know that $\mathbf{C} \xrightarrow{x?} \mathbf{C}'$, so we can directly conclude that the property holds. □

Lemma 3.6.2 states that the type suffers the changes only in the second part of the type description-the constraint, where the set of input ports together with the attached type remain the same. Moreover, the number of the constraints remains the same, but it differs from its predecessor. This coincides with the evolution of the component where the interface remains the same, but local binders evolve.

Proposition 3.6.4. *[Dependencies requirement] Let* $T =< X_b; \boldsymbol{C}_1, \boldsymbol{C}_2, \ldots, \boldsymbol{C}_k >$ *and* $\boldsymbol{C}_i = \{y_i(b_i) : \boldsymbol{B}_i : [\boldsymbol{D}_i]$. *Then,* $T \xrightarrow{y_i(b_i)} T' \Rightarrow \boldsymbol{B}_i > 0 \wedge \forall x \in dom(\boldsymbol{D}_i) : \boldsymbol{D}_i = \{x : N\} \uplus \boldsymbol{D}'_i \wedge N > 0$.

Proof. Observing the Rule [T6] the proof is direct. □

Proposition 3.6.5. *[Constraint independency] If* $K \Downarrow T =< X_b; \boldsymbol{C}_1 \uplus \cdots \uplus C_k\}$ *then* $K(y_i) \Downarrow< X_b; \boldsymbol{C}_i >= T^{y_i}$ *where* $i = 1, 2, \ldots, k$ *and* $K(y_i)$ *is the component* K *restricted to the one output port-port* y_i.

Proposition 3.6.6. *If* $x \in X_b$ *then* $< X_b, \boldsymbol{C} > \xrightarrow{x?(b)} < X_b, \boldsymbol{C}' >$.

Proof. The rules [$T1$],[$T2$] and [$T3$] imply the proposition. □

We can now prove the main theorems of this chapter. Lets recall the first theorem (Theorem 3.5.1):

$K \Downarrow T$ and $K \xrightarrow{\lambda(v)} K'$ and v has type b then $T \xrightarrow{\lambda(b)} T'$ and $K' \Downarrow T'$.

Proof. (sketch) Proof by induction on the derivation of $K \xrightarrow{\lambda(v)} K'$.

We divide the proof into two parts: first one is where we consider component K to be a base component and the second one where K is a composite one.

We start with the rules that define a transition of a base component:

[InpBase] We know that $K = [\tilde{x} > \tilde{y}]\{L\} \xrightarrow{\text{x?(v)}} [\tilde{x} > \tilde{y}]\{L'\}$, by inversion on the rule we have that $x \in \tilde{x}$ and that $L \xrightarrow{x?(v)} L'$, so the Proposition 3.6.1 holds. By hypothesis $K \Downarrow T$ and since K is a base component, by the Definition 3.3.1 $T =< X_b; \mathbf{C} >$ where

$$X_b = \{x(b) \mid x \in \tilde{x} \wedge b = \gamma(x)\}$$

and

$$\mathbf{C} = \{y_i(b'_i) : \infty : \mathbf{D}_{y_i} \mid i \in 1, \ldots, k \wedge$$
$$b'_i = \gamma(y_i) \wedge \mathbf{D}_{y_i} = \{x : \mathit{count}(x, \tilde{\sigma}^{y_i}) \mid x \in \tilde{x}^{y_i}\}\}$$

where since $x \in \tilde{x}$, then $x(b) \in X_b$ and $b = \gamma(v)$, hence $\exists T'$ such that $T \xrightarrow{x?(v)} T'$. This implies the Lemma 3.6.2, so the Proposition 3.6.3 holds. By the Definition 3.3.1 and Proposition 3.6.1 we conclude $K' \Downarrow T'$ where $T' \subseteq T$.

[OutBase] We know that $[\tilde{x} > \tilde{y}]\{L\} \xrightarrow{\text{y!(v)}} [\tilde{x} > \tilde{y}]\{L'\}$, by inversion on the rule we have that $y \in \tilde{y}$ and that $L \xrightarrow{y!(v)} L'$, so the Proposition 3.6.2 holds. Then we have that for all $x \in \tilde{x^y}$ holds: $\mathit{count}(x, \tilde{\sigma}^y) > 0$ (*).

By hypothesis we have that $K \Downarrow T$ and since K is a base component we have by the Definition 3.3.1 $T =< X_b; \mathbf{C} >$ where

$$X_b = \{x(b) \mid x \in \tilde{x} \wedge b = \gamma(x)\}$$

and

$$\mathbf{C} = \{y_i(b'_i) : \infty : \mathbf{D}_{y_i} \mid i \in 1, \ldots, k \wedge$$
$$b'_i = \gamma(y_i) \wedge \mathbf{D}_{y_i} = \{x : \mathit{count}(x, \tilde{\sigma}^{y_i}) \mid x \in \tilde{x}^{y_i}\}\}$$

where since $y \in \tilde{y}$, then $y(b) \in X_b$ and $b = \gamma(v)$.

Since (*) holds, and we have that for the base component the boundary is infinite, we have that the hypothesis of the Rule $T6$ holds. So, exists T' such that $T \xrightarrow{y!\text{(b)}} T'$. Note that $\infty - 1 = \infty$. By the Definition 3.3.1 and Proposition 3.6.2 we conclude that $K' \Downarrow T'$ where $T' \subseteq T$.

Now, we move to the rules that characterise an evolution of a composite component.

[InpComp] $K = [\tilde{x} > \tilde{y}]\{G; r = \overline{K}, R; D; r[F]\} \xrightarrow{\text{x?v}} [\tilde{x} > \tilde{y}]\{G; r = \overline{K'}, R; D; r[F]\} = K'$, then by inversion we know that $x \in \tilde{x}$ and that exists z such that $\overline{K} \xrightarrow{z?v} \overline{K'} \wedge F = z \leftarrow x, F'$. Since K is well-typed, all its subcomponents are also well-typed, so $\overline{K} \Downarrow T_r$, for some T_r. By induction hypothesis exists T'_r such that $T_r \xrightarrow{z?(b^x)} T'_r \wedge \overline{K'} \Downarrow T'_r$, where $b^x = \gamma(v)$.

Let the type of K be $T =< X_b; \mathbf{C} >$. Since $x \in \tilde{x}$ then by the definition of the type extraction $x(b^x) \in X_b$. By 57 3.6.6 we know that $T \xrightarrow{x?(b^x)} T'$, for some T'.

Since the global protocol G remained the same, its projection to role r (denoted by LP) remains unchanged due to an input on x, since x as an external port does not affect the protocol. Moreover, the modified types of each subcomponent, remained conformant to their local protocol after the evolution of K.

Now we need to prove that evolved type T' is the type extracted from T'_r and LP, i.e., $K' \Downarrow T'$.

Since the set of input ports with an input does not change, the extracted type from T'_r and LP will have the same set of input ports as T (X_b). The only possible change can be in the set of constraints.

By Proposition 3.6.5, we analyse the constraints of output ports of T separately. We have three cases: First one is when the output port does not depend on the input on x; second one is when it depends per-each-value; the last one is when we have an initial dependency. Each of these constraints was extracted from T_r and LP. We now consider how an input on x affect these constraints.

Recall that $T =< X_b, \mathbf{C} >$ and let $T_r =< Z_b; \mathbf{C}_r >$ where $\mathbf{C}_r = \{\overline{y}(b^y) : \mathbf{B}_r : [\mathbf{D}_r]\} \uplus \mathbf{C}'_r$, with $F = y \leftarrow \overline{y}, z \leftarrow x, F'$ then:

Case 1. $\mathbf{C} = \{y(b^y) : \mathbf{B} : [\mathbf{D}]\} \uplus \mathbf{C}' \wedge x \notin \mathit{dom}(\mathbf{D})$ i.e., the first case is when input port x is not in the domain of the dependencies of some output port y.

Since $z \in F^i$ and $\overline{y} \in F^o$, then by the definition of the type extraction of T we have that $\neg \exists M \mid z : M \in \mathbf{D}_d(\mathbf{C}_r, F, \overline{y}) \uplus \mathbf{D}_t(\mathbf{C}_r, F, LP, \overline{y})$, i.e., z is not in the domain of dependencies of

$\overline{y}$ obtained in a direct nor transitive way. With an input on z we do not create any new dependencies, so the constraint for y in the type extracted from T'_r and LP is $y(b^y):\mathbf{B}:[\mathbf{D}]$.
Observing the constraint for y when T evolves by Rule [T1] we have that:
$y(b^y):\mathbf{B}:[\mathbf{D}] \xrightarrow{\text{x?}} y(b^y):\mathbf{B}:[\mathbf{D}]$.
Hence, as the extracted type, T' will also have $y(b^y):\mathbf{B}:[\mathbf{D}]$ for the constraint for port y.

Case 2. $\mathbf{C} = \{y(b^y):\mathbf{B}:[\{x:N\} \uplus \mathbf{D}]\} \uplus \mathbf{C}' \Rightarrow z:N \in \mathbf{D}(\mathbf{C}_r, F, LP, \overline{y})$ i.e., port x is in the domain of the dependencies of some port y as a per-each-value dependency.
By the definition of the type extraction we have two ways to obtain the per-each-value dependency:

(a) $z:N \in \mathbf{D}_d(\mathbf{C}_r, F, \overline{y})$ i.e., when dependency on z was obtained in a direct way.
This implies that by the definition of the type extraction $\mathbf{C}_r = \{\overline{y}(b^y):\mathbf{B}_r:[\{z:N\} \uplus \mathbf{D}'_r]\} \uplus \mathbf{C}'_r$ and by inversion of Rule [T5], applying Rule [T3] we have that:
$\overline{y}(b^y):\mathbf{B}_r:[\{z:N\} \uplus \mathbf{D}'_r] \xrightarrow{\text{z?}} \overline{y}(b^y):\mathbf{B}_r:[\{z:N+1\} \uplus \mathbf{D}'_r]$.
Since it is a dependency obtained in a direct way, by the definition of the type extraction the constraint for y in the extracted type from LP and T'_r is $y(b^y):\mathbf{B}:[\{x:N+1\} \uplus \mathbf{D}']$.

Observing the constraint for y when T evolves with an input on x, by inversion on Rule [T5] and applying Rule [T3] we have that: $y(b^y):\mathbf{B}:[\{x:N\} \uplus \mathbf{D}] \xrightarrow{x?} y(b^y):\mathbf{B}:[\{x:N+1\} \uplus \mathbf{D}]$.

(b) $z:N \in \mathbf{D}_t(\mathbf{C}_r, F, LP, \overline{y})$ i.e., when the dependency on z was obtained in a transitive way.
By the definition of the type extraction exist ports y' and z' such that $y', z' \in fp(LP)$ and $y' \diamond_3^{LP} z'$, where port $\overline{y}$ ($\overline{y} \in F^o$) depends on port z', and port y' depends on port z ($z \in F^i$) namely: $\mathbf{C}_r = \{\overline{y}(b^y):\mathbf{B}_r:[\{z':N'\} \uplus \mathbf{D}'_r]\} \uplus \{y'(b^{y'}):\mathbf{B}'':[\{z:N''\} \uplus \mathbf{D}''_r]\} \uplus \mathbf{C}'_r$.
By the definition of the type extraction we know that the number of values received on x for y (N) is computed as

$N' + N'' + vf(LP, z', y')$. So, $N = N' + N'' + vf(LP, z', y')$. By inversion on Rule [T5] and applying Rule [T3] we have that:
$y'(b^{y'}) : \mathbf{B}'' : [\{z : N''\} \uplus \mathbf{D}''_r] \xrightarrow{z?} y'(b^{y'}) : \mathbf{B}'' : [\{z : N''{+}1\} \uplus \mathbf{D}''_r]$
We know that $T_r \xrightarrow{z?(b)} T'_r$. Let $T'_r =< X_b; \overline{\mathbf{C}_r} >$. Since local protocol LP remains the same, number of values flowing $vf(LP, z', y')$ remained the same after an input on z. Then in the extracted type from LP and T'_r we have that $z : \overline{N} \in \mathbf{D}_t(\overline{\mathbf{C}}_r, F, LP, \overline{y})$ where $\overline{N} = N' + N'' + 1 + vf(LP, z', y')$. This number can be written as

$$\overline{N} = (N' + N'' + vf(LP, z', y')) + 1 = N + 1.$$

We conclude that $z : N + 1 \in \mathbf{D}_t(\overline{\mathbf{C}}, F, LP, \overline{y})$ and by the definition of the type extraction we have that $y(b^y) : \mathbf{B} : [\{x : N{+}1\} \uplus \mathbf{D}]$ is the constraint for port y in the extracted type.

For $T =< X_b; \{y(b^y) : \mathbf{B} : [\{x : N\} \uplus \mathbf{D}]\} \uplus \mathbf{C}' >$ by inversion of Rule [T5] and applying Rule [T3] we have that: $y(b^y) : \mathbf{B} : [\{x : N\} \uplus \mathbf{D}] \xrightarrow{x?} y(b^y) : \mathbf{B} : [\{x : N{+}1\} \uplus \mathbf{D}]$

So, $y(b^y) : \mathbf{B} : [\{x : N{+}1\} \uplus \mathbf{D}]$ is the constraint for y in T'.

Case 3. $\mathbf{C} = \{y(b^y) : \mathbf{B} : [\{x : \Omega\} \uplus \mathbf{D}]\} \uplus \mathbf{C}'$ i.e., port x is in the domain of the dependencies of some output port y as an initial dependency. Consider again $T_r =< Z_b; \mathbf{C}_r >$.
We have two possible ways of obtaining the initial dependency:

(a) $z : \Omega \in \mathbf{D}_d(\mathbf{C}_r, F, \overline{y}) \wedge z : \Omega \notin \mathbf{D}_t(\mathbf{C}_r, F, LP, \overline{y})$ i.e., we obtained the initial dependency in a direct way.
Then we have that $\mathbf{C}_r = \{\overline{y}(b^y) : \mathbf{B}_r : [\{z : \Omega\} \uplus \mathbf{D}'_r]\} \uplus \mathbf{C}'_r$.
By Rule [T2] we have that: $\overline{y}(b^y) : \mathbf{B}_r : [\{z : \Omega\} \uplus \mathbf{D}'_r] \xrightarrow{z?} \overline{y}(b^y) : \mathbf{B}_r : [\mathbf{D}'_r]$.
This means that dependency is dropped, so it will also be dropped in the extracted type.
By Rule [T2] in T' the dependency of y on x will be dropped: $y(b^y) : \mathbf{B} : [\{x : \Omega\} \uplus \mathbf{D}] \xrightarrow{z?} y(b^y) : \mathbf{B} : [\mathbf{D}]$.

(b) $z : \Omega \in \mathbf{D}_t(\mathbf{C}_r, F, LP, \overline{y})$, i.e., the initial dependency was obtained in a transitive way. We do not exclude the pos-

sibility of having $z : \Omega$ in the set of direct dependencies, since we saw that with an input on z it will be dropped, so this possibility does not interfere with this case.
By the definition of the type extraction we have that there exist ports y' and z' such that $y', z' \in \mathit{fp}(LP)$ and $y' \diamond_i^{LP} z'$, where
$\mathbf{C}_r = \{\overline{y}(b^y) : \mathbf{B}' : [\{z' : M'\} \uplus \mathbf{D}'], y'(b^{y'}) : \mathbf{B}'' : [\{z : M\} \uplus \mathbf{D}'']\} \uplus \mathbf{C}_{r_1}$.
One of the conditions of having a transitive initial dependencies are:

I $i = 3 \wedge M = 0 \wedge M' = \Omega \wedge \mathit{vf}(LP, z', y') = 0$
II $i = 3 \wedge M = \Omega \wedge \mathit{vf}(LP, z', y') = 0$
III $i \in \{1, 2\} \wedge M \not> 0$

Number $\mathit{vf}(LP, z', y')$ remains the same (it is zero) due to the fact that the protocol did not evolve. With an input on z transitive dependency on z is dropped due to the rules of the semantic where either it is dropped since the dependency of y' on x is dropped as initial dependency (Rule [T2]) or $M = \overline{M} + 1 \geq 1$, for some $\overline{M} \in \mathbb{N}_0$ (Rule [T3]).
Due to the semantics of the typing language by inversion on Rule [T5], applying Rule [T2] we have that also the dependency of y on x is dropped:
$y(b^y) : \mathbf{B} : [\{x : \Omega\} \uplus \mathbf{D}] \xrightarrow{x?} y(b^y) : \mathbf{B} : [\mathbf{D}]$.

Since all the constraints in the extracted type from T_r' and LP match the constraints in T' and the set of input ports remain the same, we can conclude that the extracted type and T' are the same and that $K' \Downarrow T'$.

[OutComp] $K = [\tilde{x} > \tilde{y}]\{G; r = \overline{K}, R; D; r[F]\} \xrightarrow{\text{y(v)}} [\tilde{x} > \tilde{y}]\{G; r = \overline{K'}, R; D; r[F]\} = K'$, then by inversion we know that $y \in \tilde{y}$ that exist $\overline{y}$ such that $\overline{K} \xrightarrow{\overline{y}!(v)} \overline{K'} \wedge F = y \leftarrow \overline{y}, F'$. Since K is well-typed, so are its subcomponents, thus, $\overline{K} \Downarrow T_r$. By induction hypothesis we have that $\exists b^y, T_r'$ such that $T_r \xrightarrow{\overline{y}!(b^y)} T_r' \wedge \overline{K'} \Downarrow T_r'$, where $\gamma(v) = b$.

Let $T =< X_b; \mathbf{C} >$, since $\overline{y} \in F^o$ and T_r could do an output on $\overline{y}$, the value is directly forwarded to y, then $\exists T' : T \xrightarrow{y!(b^y)} T'$.

After an output on y, global protocol G remains unchanged, so is its projection to role r (denoted by LP). Also, for the same reason all the subcomponents remain conformant to their local protocols.Moreover, the set of input ports with their basic types in type T' and in the extracted type from T_r' and LP remains unchanged.

Now we need to prove that type T' is the type extracted from T_r' and LP i.e., that $K' \Downarrow T'$.

The set of input ports in the type extracted from T_r' and LP remains unchanged, i.e., it is the same as in type T, since due to the rules of the semantics we cannot lose or gain new input ports.

Let $T_r =< Z_b; \mathbf{C}_r >$ and $\mathbf{C}_r = \{\overline{y}(b^y) : \mathbf{B}_r : [\mathbf{D}_r]\} \uplus \mathbf{C}_{r_1}$ and we have that $F = y \leftarrow \overline{y}, F' \Rightarrow \overline{y} \in F^o$.

For the extraction of the dependencies, we consider two cases: First case is that y has no dependencies and the second one is when it has and due to the rules of the semantics (Rule [T6]), all of them are per-each-value dependencies.

I If $\mathbf{C} = \{y(b^y) : \mathbf{B} : [\emptyset]\} \uplus \mathbf{C}' \Rightarrow \neg\exists x \mid x : N \in (\mathbf{D}_d(\mathbf{C}_r, F, \overline{y}) \cup \mathbf{D}_t(\mathbf{C}_r, F, LP, \overline{y}))$, i.e., if y had no dependencies in type T, then in the type extraction the sets of dependencies of y obtained in a direct or transitive way are empty.

   By induction hypothesis we have that $T_r \xrightarrow{\overline{y}!(b^y)} T_r'$, thus, we have that:

   $< Z_b; \{\overline{y}(b^y) : \mathbf{B}_r : [\mathbf{D}_r]\} \uplus \mathbf{C}_{r_1} > \xrightarrow{\overline{y}!(b^y)} < Z_b; \{\overline{y}(b^y) : \mathbf{B}_r - 1 : [\mathbf{D}_r']\} \uplus \mathbf{C}_{r_1} >$

   Observing Rule [T6] and Definition 3.4.1, we conclude that the boundary in the extracted type is $\mathbf{B}_r' = min\{B_1 - 1, B_2 - 1, B_3 - 1, \mathbf{B}_r - 1\}$. So, the extracted type from T_r' and LP is

$$(< \mathbf{ren}(F, Z_b) >; \mathbf{ren}(\{\overline{y}(b^y) : \mathbf{B}_r' : [\emptyset]\}) \uplus \mathbf{C}'\}$$

   However, we cannot consider the set of possible boundaries $\{B_2\}$, because in that case exists some input port on which $\overline{y}$ initially depends that is not in *fp*(LP) nor in F^i (extracted boundary, the minimum is zero). This implies that $\overline{y}$ is not able to have an output, that as a consequence has that a value cannot be emitted from y. If boundary is 0 a type cannot perform an output ([Rule T6]), since in that case for the boundary of the type extracted from T' and LP is $0-1$. So we have that $B'_r = min\{B_1-1, B_3-1, \mathbf{B}_r-1\}$.

Now, for type T, applying Rule $[T6]$

$< X_b; \{y(b^y) : \mathbf{B} : [\emptyset]\} \uplus \mathbf{C}' > \xrightarrow{y!(b^y)} < X_b; \{y(b^y) : \mathbf{B} - 1 : [\emptyset]\} \uplus \mathbf{C}' >$ and we have that $min\{B_1 - 1, B_3 - 1, \mathbf{B}_r - 1\} = min\{B_1, B_2, B_3, \mathbf{B}_r\} - 1 = \mathbf{B}-1$ and we conclude that the extracted type from T'_r and LP matches with T'.

– If $\mathbf{C} = \{y(b^y) : \mathbf{B} : [\{x_1 : N_1, \ldots, x_k : N_k\} \uplus \mathbf{D}]\} \uplus \mathbf{C}'$, i.e., if y had dependencies on some input ports $x_1, \ldots, x_k$.

By the definition of the type extraction we know that there exist a set $\{z_1 : N_1, \ldots, z_k : N_k\}$ where $\{z_1 : N_1, \ldots, z_k : N_k\} = \mathbf{ren}(F, \{x_1 : N_1, \ldots, x_k : N_k\})$ and
$\{z_1 : N_1, \ldots, z_k : N_k\} = \mathbf{D}_t(\mathbf{C}_r, F, LP, \overline{y}) \uplus \mathbf{D}_d(\mathbf{C}_r, F, \overline{y})$.
Considering that $T_r =< Z_b; \mathbf{C}_r >$, then $z_1 : N_1, \ldots, z_k : N_k \in Z_b$.
Observing how we obtained the dependencies of $\overline{y}$ on z_i ($i = 1, 2, \ldots, k$) in the extracted component from T_r to LP, we have the following cases that focus on one input port, without the loss of generality:

Case 1. $z_i : N_i \in \mathbf{D}_d(\mathbf{C}_r, F, \overline{y})$, i.e., the dependencies are obtained in a direct way.
Then we have that $\mathbf{C}_r = \{\overline{y}(b^y) : \mathbf{B}_r : [\{z_i : N_i\} \uplus \mathbf{D}'_r]\} \uplus \mathbf{C}'_r$.
Since T_r had an output from port $\overline{y}$, by Rule [T6] we have that:
$T_r \xrightarrow{\overline{y}!(b^y)} < Z_b; \{\overline{y}(b^y) : \mathbf{B}_r{-}1 : [\{z_i : N_i{-}1\} \uplus \mathbf{D}''_r\}] \uplus \mathbf{C}'_r >= T'_r$.
By Definition 3.4.1 in the extracted type obtained from LP and T'_r, we have that $z_i : N_i{-}1$ is the element of the set of the dependencies of $\overline{y}$.

Case 2. $z_i : N_i \in \mathbf{D}_t(\mathbf{C}_r, F, LP, \overline{y})$, i.e., the dependencies are obtained in a transitive way.
We have to notice that it is a per-each-value dependency and that there is only one possible way to obtain it, when we consider transitive dependencies:

Let $T_r =< Z_b; \mathbf{C}_r >$ then there must exist $y', z' \in \mathit{fp}(LP)$ such that $y' \diamond_3^{LP} z'$ (recap: recursive protocol, where both y' and z' are in $\mathit{rep}(LP)$ in such an order that an output on y' precedes the input on z'), where
$\mathbf{C}_r = \{\overline{y}(b^y) : \mathbf{B}_r : [\{z' : N'\} \uplus \mathbf{D}_{\overline{y}}], y'(b^{y'}) : \mathbf{B}_r{}^1 : [\{z_i : N_i^{y'}\} \uplus \mathbf{D}^{y'}]\} \uplus \mathbf{C}'_r$.
Since $T_r \xrightarrow{\overline{y}!(b^y)} T'_r$, by Rule [T6] we have the following:
$< Z_b; \mathbf{C}_r > \xrightarrow{\overline{y}!(b^y)} < Z_b; \{\overline{y}(b^y) : \mathbf{B}_r - 1 : [\{z' : N' - 1\} \uplus \mathbf{D}'_{\overline{y}}], y'(b^{y'}) : \mathbf{B}_r^r : [\{z_i : N_i^{y'}\} \uplus \mathbf{D}']\} \uplus \mathbf{C}'_r >= T'_r$, where by inversion we know that $N' > 0, \mathbf{B}_r > 0$.

By the type extraction procedure from LP and T'_r, the number of values from port z_i available for $\overline{y}$ is $N^{z_i} = (N' - 1) + N_i^{y'} + \mathit{vf}(LP, x', y')$.
Since $T =< X_b; \{y(b^y) : \mathbf{B} - 1 : [\{x_1 : N_1, \dots, x_k : N_k\} \uplus \mathbf{D}]\} \uplus \mathbf{C}' >$ and by the type extracting procedure we know that $N_i = N' + N_i^{y'} + \mathit{vf}(LP, z', y')$. By Rule [T6] we have that $T \xrightarrow{y!(b^y)} < X_b; \{y(b^y) : \mathbf{B} - 1 : [\{x_1 : N_1 - 1, \dots, x_i : N_i - 1, \dots, x_k : N_k - 1\} \uplus \mathbf{D}]\} \uplus \mathbf{C}' >= T'$.
This implies that the number of values from port x_i available for y ($i = 1, \dots, k$) in type T' is $N_i - 1 = N' - 1 + N_i^{y'} - +\mathit{vf}(LP, z', y') = N^{z_i}$.
Note that $\mathit{vf}(LP, z', y')$ remained the same since the protocol did not evolve.
The boundary of y decreases by one compared to the one in T and that all the ports in the dependency of y have one value less available for computing y, applying Rule [T6] we have that the extracted type and T' match so $K' \Downarrow T'$.

[Internal] $K = [\tilde{x} > \tilde{y}]\{G; r = \overline{K}, R; D; r[F]\} \xrightarrow{\tau} K' = [\tilde{x} > \tilde{y}]\{G; r = \overline{K'}, R; D; r[F]\}$, then by inversion we know that $\overline{K} \xrightarrow{\tau} \overline{K'}$. If $\overline{K}$ has a type T_r, by induction hypothesis we know that there exist type T'_r such that $T_r \xrightarrow{\tau} T_r \wedge \overline{K'} \Downarrow T_r$. We can conclude that each type of the subcomponents remained the same, and also the global protocol did not evolve, so these types remained conformant with their local protocols. Therefore, the extracted type from T'_r and LP is the one extracted from T_r and LP, and by Rule [T4] we know that $T \xrightarrow{\tau} T$.

[InpChor] $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{\tau} [\tilde{x} > \tilde{y}]\{G'; r = K'_r, R; D; r[F]\} = K'$, then by inversion we know that $K_r \xrightarrow{z?(v)} K'_r \ \wedge \ D = r.z' \leftarrow p.u, D' \ \wedge \ G \xrightarrow{r?l < v >} G'$.

Let $R = r_1 = K_1, r_2 = K_2, \ldots, r_m = K_m$. Since K is well-typed, all its subcomponents are also well-typed, hence, exist types $T_r, T_1, \ldots, T_m$ such that $K_r \Downarrow T_r,\ K_1 \Downarrow T_1,\ \ldots,\ K_m \Downarrow T_m$.

Let the projection of protocol G to role r ($G \downarrow_r$) be local protocol LP. Since component K_r, assigned to role r, can input a value v on port z, then $LP = z?(b).LP'$.

By Definition 3.5.1, since K_r is the only interfacing component, then $\mathcal{T}(F, T_i) = T_i$, where $i = 1, \ldots, m$.

Since all the subcomponents are well-typed, their modified types are conformant to their local protocols: $\mathcal{T}(F, T_r) \bowtie z?(b).LP'$, $T_1 \bowtie G \downharpoonleft_{r_1}, \ \ldots, \ T_m \bowtie G \downharpoonleft_{r_m}$.

Since x is a port of K_r, after the input on z all the other subcomponents are conformant to their local protocol.

By induction hypothesis, we know that exist b, T'_r such that $\gamma(v) = b$ and $T_r \xrightarrow{z?(b)} T'_r$.

Let $T_r =< Z_b; \mathbf{C}_r >$, $T_i =< Z^i_b; \mathbf{C}_i >$, where $i = 1, \ldots, m$. Since $z(b) \in Z_b$ and $z(b) \notin Z^i_b, \forall i = 1, \ldots, m$, by Rule [$\mathcal{T}5$] we have that:

$\mathcal{T}(F, T_r) \xrightarrow{z?(b)} \mathcal{T}'(F, T_r), T_1 \xrightarrow{z?(b)} T_1, \ldots, T_m \xrightarrow{z?(b)} T_m$.

By Rule $[InpConf]$ we have that: $\mathcal{T}'(F, T_r) \bowtie G' \downharpoonleft_r, T_1 \bowtie G' \downharpoonleft_{r_1} \ldots, T_m \bowtie G' \downharpoonleft_{r_m}$.

Let $T =< X_b; \mathbf{C} >$, be the extracted type from LP and T_r. Since $T_r \xrightarrow{z?(b)} T'_r$, then T does an internal move, i.e., $T \xrightarrow{\tau} T$. We need to prove that $K' \Downarrow T$.

Since the set of input ports X_b after any transition remains the same, we need to prove that the set of constraints extracted from LP' and T'_r will be exactly $\mathbf{C}$. Precisely, since we do not output a value from the external ports it is enough to prove that the dependencies of the output ports remained the same.

In reminder $T_r \xrightarrow{z?(b)} T'_r$ and $T_r =< Z_b; \mathbf{C}_r >$. Since $z \in fp(LP)$ we need to consider the following cases:

Case 1. $\neg\exists\overline{y} \in F^o$ such that $\overline{y}(b^{\overline{y}}) : \mathbf{B}^{\overline{y}} : [\{z : M\} \uplus \mathbf{D}^{\overline{y}}]$, i.e., there is no external port depending on z. By the definition of the type extraction we cannot obtain the transitive dependency (on some external port) of any output port via z. So, we cannot obtain any new ones after an input on z, so the dependencies of the external output ports remain unchanged.

Case 2. $\exists\overline{y} \in F^o$ such that $\overline{y}(b^{\overline{y}}) : \mathbf{B}^{\overline{y}} : [\{z : M\} \uplus \mathbf{D}^{\overline{y}}]$, i.e., some external output port depends on the input on z.

Since we have local protocol $LP = z'? : t'_b.LP'$ we consider the following scenarios:

a. $\neg\exists y' \mid y' \diamond_3^{LP} z$, i.e., one of the conditions of obtaining the transitive dependency of en external port $\overline{y}$, where z is the port involved, fails. By the definition of the type extraction the dependencies obtained in a transitive way remain the same after an input on z since port z does not have any impact on obtaining them.

b. $\exists y' \mid y' \diamond_3^{LP} z$, i.e., one of the conditions for obtaining the transitive dependency of port $\overline{y}$ is fulfilled.

We now have to consider other two possibilities:

I $\neg\exists z_1 \in F^i \mid y'(b^{y'}) : \mathbf{B}^{y'} : [\{z_1 : M_1 \uplus \mathbf{D}^{y'}\}]$, one of the conditions of obtaining the transitive dependency of $\overline{y}$ where in the extraction port z is included, fails, so an input on z will not change any dependencies obtained in a transitive way.

II $\exists z_1 \in F^i \mid y'(b^{y'}) : \mathbf{B}^{y'} : [\{z_1 : M_1\} \uplus \mathbf{D}^{y'}]$ i.e., there exist an external port z_1 such that y' depends on it.

Combining cases 2,b and II, by the extraction procedure, we have the dependency of $\overline{y}$ on z_1 obtained in a transitive way. To sum up we have that set of constraints $\mathbf{C}_r$ is

$$\mathbf{C}_r = \{\overline{y}(b^{\overline{y}}) : \mathbf{B}^{\overline{y}} : [\{z : M\} \uplus \mathbf{D}^{\overline{y}}],$$

$$y'(b^{y'}) : \mathbf{B}^{y'} : [\{z_1 : M_1\} \uplus \mathbf{D}^{y'}]\} \uplus \mathbf{C}_r^1\}$$

where $y' \diamond_3^{LP} z$. Recall that then

$$LP^1 = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[y'!(t'_b).\mathcal{C}''[x'?(.)LP']]]$$

Since we have that $LP = z?(b).LP'$, indicates that the component already had an output from y'. Since y' depends on an input on z_1 implies that the value is already received.
If $M_1 = \Omega$ then the dependency of $\overline{y}$ on z_1 was already dropped in T, so an input on z does not change that fact.
If instead, $M_1 = N$ and $M = \Omega$, by the type extraction procedure in type T we do not have a dependency of $\overline{y}$ on z_1 (up to renaming), since $vf(LP, z, y') = 1$. After an input on z, $vf(LP, z, y') = 0$, but the dependency of $\overline{y}$ on z will be dropped, hence, by the type extraction procedure, there is no dependency of $\overline{y}$ on z_1 obtained in a transitive way .
Now, we consider the case where $M = N$ and $M_1 = N_1$. By the definition of the type extraction the number of values from port z_1 available for $\overline{y}$ in T (up to renaming) is $N + N_1 + vf(LP, z, y') = N + N_1 + 1$ ($vf(LP, z, y') = 1$). After an input on z, the number of values of z available for $\overline{y}$ (that was N) will increase by 1 (Rule [T5]), hence in the extracted type from T'_r and LP' we have that the number of values of port z_1 available for $\overline{y}$ is $N{+}1{+}N_1{+}vf(LP, z, y') = N{+}1{+}N_1{+}0$ ($vf(LP, z, y') = 0$ since the value that was flowing was input on z. We conclude that for this case the dependencies in the constraint will remain the same in T', as it is in T.

We can conclude that dependencies of the output ports did not change. We conclude that the extracted type from LP' and T'_r is type T. Therefore, $K' \Downarrow T$.

[OutChor] $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{\tau} [\tilde{x} > \tilde{y}]\{G'; r = K'_r, R; D; r[F]\} = K'$, then by inversion we know that $K_r \xrightarrow{y'!(v)} K'_r \;\wedge\; D = p.u \leftarrow r.y', D' \;\wedge\; G \xrightarrow{r!l < v >} G'$.

The first part of the proof for Rule [OutChor] and the assumptions are the same as for Rule [InpChor], but for an output (from port y').

Below we assume:

- $T =< X_b; \mathbf{C} >$;
- $R = r_1 = K_1, r_2 = K_2, \ldots, r_m = K_m$.
- $\exists T_r, T_1, \ldots, T_m$ such that $K_r \Downarrow T_r,\ K_1 \Downarrow T_1,\ \ldots,\ K_m \Downarrow T_m$;
- $\exists b, T_r' \mid \gamma(v) = b\ \wedge\ T_r \xrightarrow{y'!(b)} T_r'$;
- LP is the projection of G to role r and $LP = y'!(b).LP'$;
- $\mathcal{T}(F, T_i) = T_i$, where $i = 1, \ldots, m$;
- $\mathcal{T}(F, T_r) \bowtie z?(b).LP',\ T_1 \bowtie G \downharpoonright_{r_1},\ \ldots,\ T_m \bowtie G \downharpoonright_{r_m}$;
- $G \xrightarrow{r?l < v >} G'$ then $G \downarrow_i = G' \downarrow_i,\ \ i = 1, \ldots m$;
- Let $T_r =< Z_b; \mathbf{C}_r >, T_i =< Z_b^i; \mathbf{C}_i >$, where $i = 1, \ldots, m$;
- $\mathcal{T}(F, T_r) \xrightarrow{y'!(b)} \mathcal{T}'(F, T_r), T_1 \xrightarrow{y'!(b)} T_1, \ldots, T_m \xrightarrow{y'!(b)} T_m$;
- By Rule $[OutConf]$ we have that: $\mathcal{T}'(F, T_r) \bowtie G' \downharpoonright_r, T_1 \bowtie G' \downharpoonright_{r_1} \ldots, T_m \bowtie G' \downharpoonright_{r_m}$.

  Let us prove that the extracted type from T_r' and LP' is T and that $K' \Downarrow T$.

  Since the set of input ports X_b after any transition remains the same (we do not lose or gain any new input ports due to the semantics of (modified) type), we need to prove that the set of constraints extracted from LP' and T_r' are exactly the ones in $\mathbf{C}$. Again, since we do not output a value from the external ports it is enough to prove that the dependencies of the output ports remained the same.

In reminder we have that $T_r \xrightarrow{y'!(b)} T_r'$ and $LP = y'!(b).LP'$.
Let $T_r =< Z_b; \{\overline{y}(b) : \mathbf{B}^{\overline{y}} : [\mathbf{D}^{\overline{y}}]\} \uplus \mathbf{C}_r^1 >$.

Since T_r can do an output from port y', by Rule [T6] we know that $T_r' =< Z_b; \{y'(b) : \mathbf{B}^{y'} - 1 : [\mathbf{D}_1^{y'}]\} \uplus \mathbf{C}_r^1 >$, where $\forall z : N \in \mathbf{D}^{y'} \Rightarrow z : N - 1 \in \mathbf{D}_1^{y'}$.

If in the description of LP' we do not have any input ports or if we had ones such that there is no external output port depending on them, output on y' does not have any impact on the dependencies obtained in a transitive way. Moreover, by the type extraction procedure, it is also irrelevant for creating the dependencies obtained in a direct way.

Consider now the case where we have in the description of local protocol LP' an input port z' and that $\exists y \in F^o$ such that y depends on z'. We have two cases:

Case 1. $\neg\exists z \in F^i$ such that y' depends on it. If this is the case, we do not have the dependency of port y obtained in a transitive way, hence, the output from the port does not have any impact on the extracted constraints of y.

Case 2. $\exists z \in F^i$ such that y' depends on it.
Similar to the proof for Rule [InpChor] we have that set of constraints **C** is

$$\mathbf{C}_r = \{y'(b^{y'}) : \mathbf{B}^{y'} : [\{z : M\} \uplus \mathbf{D}_2^{y'}], y(b) : \mathbf{B}^y : [\{z' : M'\} \uplus \mathbf{D}^y]\} \uplus \mathbf{C}_r^2$$

Extracting type T, port y cannot initially depend on port z, since we know that y' can output, which implies that a value was already received on z, i.e. before an output on y' the dependency was already dropped.
If there was a per-each-value dependency of y on z in type T, then by the type extraction procedure we know that $y' \diamond_3^{LP} z'$, $M = N$ and $M' = N'$. Before an output on y', we have in T that the number of values on z available for y is $N + N' + 0$ ($\mathit{vf}(LP, z', y') = 0$ before an output on y'). After an output on y', N will decrease by 1 (Rule [T6]), but $\mathit{vf}(LP, z', y')$ will increase by 1 ($\mathit{vf}(LP, z', y') = 1$). Hence, in the extracted type from T'_r and LP' the number of values of port z available for y is $N - 1 + N_1 + \mathit{vf}(LP, z, y') = N - 1 + N_1 + 1$. Therefore, the dependencies of the output ports in type T' are the same as those in T. Therefore, $K \Downarrow T'$, where $T' = T$.

□

First, we prove the following lemma that we use to prove the Type fidelity theorem.

Lemma 3.6.3. *[Protocol Fidelity] Let* $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\}$ *be a well-typed composite component. Assuming all subcomponents enjoy the type fidelity property:*

$$K \Downarrow T \text{ and } T \xrightarrow{\lambda(b)} T' \text{ and } \lambda(b) \neq \tau \text{ then } b \text{ is the type of a value } v \text{ and}$$
$$K \overset{\lambda(v)}{\Longrightarrow} K' \text{ and } K' \Downarrow T'.$$

Then for any trace such that

$$G \xrightarrow{p_1\ell_1(v_1)} \cdots \xrightarrow{p_k\ell_k(v_k)} G'$$

we have that

$$[\tilde{x} > \tilde{y}]\{G; R; D; r[F]\} \xrightarrow{\tau} \cdots \xrightarrow{\tau} [\tilde{x} > \tilde{y}]\{G'; R'; D; r[F]\}$$

Proof. By induction on the size of the trace.

Base case. $k = 1$, i.e., the size of the trace is one. Then we have that $G \xrightarrow{p\ell(v)} G'$, where $p \in \{p!, p?\}$. Let $R = p = K_p, R_1$. Since K is a well typed component, there exists some T such that , $K \Downarrow T$. Then all its subcomponents are well-typed, so exists T_p such that $K_p \Downarrow T_p$ and let $LP = G \downharpoonright_p$. Moreover, all the subcomponents' modified types remained conformant with their local protocols after protocol G evolved, besides the component with role p, since all the other protocol projections remained the same. Instead, $\mathcal{T}(F, T_p) \xrightarrow{a(b)} \mathcal{T}(F, T'_p)$, where $a = x?$ if $p = p?$, $a = y!$ if $p = p!$, and $\gamma(v) = b$, where x and y are ports explained in a distribution binder (**D**). After protocol G evolved to G', by Rule $[InpConf]$ for $a = x?$ or Rule $[OutConf]$ for $a = y!$, we have that $\mathcal{T}(F, T'_p) \bowtie G' \downharpoonright_p$, so $T_p \xrightarrow{\tau} \cdots T_p \xrightarrow{a(b)} T'_p$ (in reminder Rule $[T4]$ $T_p \xrightarrow{\tau} T_p$). Subcomponent K_p enjoys the type fidelity property, so exists some value v of type b, and K'_p such that $K_p \overset{a(v)}{\Longrightarrow} K'_p$ and $K'_p \Downarrow T'_p$, K'_p also enjoys the type fidelity property. Applying Rule[Internal] some number of times and then Rule [InpChor] (or [OutChor] depending on the nature of I/O action) we have that $K \xrightarrow{\tau} [\tilde{x} > \tilde{y}]\{G; R_1; D; r[F]\} \xrightarrow{\tau} \cdots [\tilde{x} > \tilde{y}]\{G'; R'; D; r[F]\} = K'$. We have that $K \Downarrow T$, by Theorem 3.5.1 then $T \xrightarrow{\tau} \cdots T$ and $K' \Downarrow T$.

Induction hypotehsis. Assume that the property holds for any trace of size $k = n - 1$.

Inductive step. We prove that the property holds for any trace of size $k = n$, i.e.,

$$G \xrightarrow{p_1\ell_1(v_1)} \cdots \xrightarrow{p_{n-1}\ell_{n-1}(v_{n-1})} G^{n-1} \xrightarrow{p_n\ell_n(v_n)} G'.$$

By induction hypothesis exists K'' such that $K \xrightarrow{\tau} \cdots \xrightarrow{\tau} [\tilde{x} > \tilde{y}]\{G^{n-1}; R''; D; r[F]\} = K''$. Since $K \Downarrow T$ and K has some number of internal steps e.g., m steps, applying the Theorem 3.5.1 m times we know that exists T'' such that $T \xrightarrow{\tau} \cdots \xrightarrow{\tau} T''$ and $K'' \Downarrow T''$. By inversion on rules [InpChor], [OutChor] or [Internal] we know that then for some subcomponents $K_1, K_2, \ldots K_l$, $l \geq 1$ such that $R = p_1 = K_1, p_2 = K_2, \ldots p_l = K_l, R'$ (R' possibly empty list) with possibility of having the case where $p_i = K_i = r = K_r$ for some $i \in \{1, 2, \ldots l\}$ holds the following

$$K_i \xRightarrow{a(v)} K_i'$$

or

$$K_i \xRightarrow{\tau} K_i'$$

Since each subcomponent K_i is well-typed, i.e., exists T_i such that $K_i \Downarrow T_i$, by the Theorem 3.5.1 (applied multiple times) exists T_i' such that $K_i' \Downarrow T_i'$. Each K_i' enjoys the type fidelity property.

If we apply the reasoning for the base case having $G^{n-1} \xrightarrow{p_n\ell_n(v_n)} G'$, $K'' \Downarrow T''$ and all subcomponents of K'' that enjoy the type fidelity property, we conclude that $K'' \xRightarrow{\tau} K'$, so in conclusion, for n-size trace where

$$G \xrightarrow{p_1\ell_1(v_1)} \cdots \xrightarrow{p_{n-1}\ell_{n-1}(v_{n-1})} G^{n-1} \xrightarrow{p_n\ell_n(v_n)} G'.$$

exists K' such that

$$K \xRightarrow{\tau} K'.$$

□

Let us now recall the Type fidelity theorem (Theorem 3.5.2):

Theorem 3.6.1 (Type fidelity). *If $K \Downarrow T$ and $T \xrightarrow{\lambda(b)} T'$ and $\lambda(b) \neq \tau$ then b is the type of a value v and $K \xRightarrow{\lambda(v)} K'$ and $K' \Downarrow T'$.*

Proof. (Sketch) Proof by induction on the structure of K.

- First, we prove the base case, where K is a base component by inspection on the rules for types.
- Then we assume that the type fidelity property holds for all the subcomponents of K.
- Finally, we prove that the type fidelity property holds for K by inspection on the rules for types.

Base case. Let K be a base component $[\tilde{x}\rangle\,\tilde{y}]\{L\}$ of type T. Then, we prove the statement by induction of the derivation of $T \xrightarrow{\lambda(b)} T'$ and by cases on the last rule applied:

[T5] $< \{x(b^x)\} \uplus X_b; \{y_i(b_i) : \mathbf{B}_i : [\mathbf{D}_i] | i \in 1, \dots, k\} > \xrightarrow{x?(b^x)} < \{x(b^x)\} \uplus X_b; \{y_i(b_i) : \mathbf{B}_i : [\mathbf{D}'_i] | i \in 1, \dots, k\} >$, then by inversion we know that $\forall i \in 1, 2, \dots, k$ it holds that $y_i(b_i) : \mathbf{B}_i : [\mathbf{D}_i] \xrightarrow{x?} y_i(b_i) : \mathbf{B}_i : [\mathbf{D}'_i]$. Since $x(b^x) \in \{x(b^x)\} \uplus X_b$, by the Definition 3.3.1, we have that $x \in \tilde{x}$ and that there exist a component K' and a value v of type b ($\gamma(v) = b^x$) such that $K \xrightarrow{x?(v)} K'$. We need to prove that $K' \Downarrow T'$.

We have that $T =< X_b; \{\mathbf{C}_i | i = 1, 2, \dots, k\} >$ and $T \xrightarrow{x?(b^x)} T'$ then, by Lemma 3.6.2, $T' =< X_b; \{\mathbf{inc}(\{\mathbf{C}_i, x\}) | i = 1, 2, \dots, k\} >$ (by the type semantics, T' is unique).

Since K is a base component and $[\tilde{x}\rangle\,\tilde{y}]\{L\} \xrightarrow{x?(v)} [\tilde{x}\rangle\,\tilde{y}]\{L'\}$, then by inversion on the Rule [InpBase] we know that $L \xrightarrow{x?v} L'$ and that $x \in \tilde{x}$, so the Lemma 3.6.1 holds.

By the Definition 3.3.1 $K' \Downarrow T'$.

[T6] $< X_b; \{y(b^y) : \mathbf{B} : [\{x_i : N_i | i \in 1, \dots, k\}]\} \uplus \mathbf{C} > \xrightarrow{y!(b^y)} < X_b; \{y(b^y) : \mathbf{B} - 1 : [\{x_i : N_i - 1 | i \in 1, \dots, k\}]\} \uplus \mathbf{C} >$. By inversion we know that $B > 0$ and $N_i > 0$ for all $i \in 1, \dots, k$. Since it holds that $y(b^y) : \mathbf{B} : [\{x_i : N_i | i \in 1, \dots, k\}] \in \{y(b^y) : \mathbf{B} : [\{x_i : N_i | i \in 1, \dots, k\}]\} \uplus \mathbf{C}$, then by Definition 3.3.1 we have that $y \in \tilde{y}$, and that there exist K' and a value v of type b^y ($\gamma(v) = b^y$) such that $K \xrightarrow{y!(v)} K'$. We need to prove that $K' \Downarrow T'$. Recall that $K = [\tilde{x}\rangle\,\tilde{y}]\{L\}$, then since $[\tilde{x}\rangle\,\tilde{y}]\{L\} \xrightarrow{y!(v)} [\tilde{x}\rangle\,\tilde{y}]\{L'\}$ by the premise of the Rule [OutBase] we know that $L \xrightarrow{y!(v)} L'$ and $y \in \tilde{y}$, so the Lemma 3.6.1 holds. By the Definition 3.3.1 we conclude that $K' \Downarrow T'$.

We proved the case where K is a base component. Now we apply the induction hypothesis for a composite component.

Induction hypothesis Assume that all the subcomponents of K enjoy the type fidelity property.

Inductive step We prove that the component K enjoys the type fidelity property. We have that $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\}$ where $K \Downarrow T$ and $T \xrightarrow{\lambda(b)} T'$.
Let $T =< X_b; \mathbf{C} >$. Since $K \Downarrow T$ then exists a type T_r such that $K_r \Downarrow T_r$. Let $T_r =< Z_b; \mathbf{C}_r >$ and LP be the local protocol of the component K_r. Since K is well-typed, all of its subcomonent's modified types are conformant with their local protocol (e.g. $\mathcal{T}(F, T_r) \bowtie G \downharpoonright_r = LP$). Let us now divide the proof depending on the label λ.

Case 1. $[\lambda = x?]$, i.e., if we had an input on the port x.

Since $T \xrightarrow{x?(b^x)} T'$ we know by the Rule [T5]) that $x(b) \in X_b$. By the extraction procedure of a composite component $\exists z(b^x) \in Z_b, T'_r \mid F = z \leftarrow x, F' \Rightarrow T_r \xrightarrow{z?(b^x)} T'_r$. By induction hypothesis $\exists K'_r, \gamma, v \mid K_r \xrightarrow{\text{z?(v)}} K'_r \wedge K'_r \Downarrow T'_r \wedge \gamma(v) = b^x$ (since the value is forwarded the value is directly input, i.e., the number of internal moves is zero). Applying the rule [InpComp] we have that then $K \xrightarrow{\text{x?(v)}} K'$. Since $K \Downarrow T$ and $K \xrightarrow{\text{x?(v)}} K'$, applying the [Theorem 3.5.1] and knowing by the definition of the type extraction that T' is unique we have that $K' \Downarrow T'$.

This is true because an external input does not affect the modified types of the subcomponents different from the interfacing one, so they remained conformant to their local protocol. In the case of the interfacing component by the Rule [$\mathcal{T}$5] and the 48 3.5.1, an input on the external port does not affect the modified type.

Case 2. $[\lambda = y!]$

Since $K \Downarrow T$ and $T \xrightarrow{y!(b)} T'$ we know by the definition of the type extraction that $\exists \overline{y} : F = y \leftarrow \overline{y}, F'$.

We have to consider two possible cases:

1 $T_r \xrightarrow{\overline{y}!(b)} T'_r$

2 $T_r \not\xrightarrow{\overline{y}!(b)}$

If the case 1 holds, by induction hypothesis exist K'_r and v such that $K_r \xrightarrow{\overline{y}!(v)} K'_r$, $\gamma(v) = b$ and $\overline{K'} \Downarrow T'_r$ (since the value is forwarded the value is directly output, i.e., the number of internal moves is zero). Then, by the rule [OutComp] we have that $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{\text{y!(v)}} K = [\tilde{x} > \tilde{y}]\{G; r = K'_r, R; D; r[F]\}$. Since G did not move (all the projections remained the same) and the output on the port $\overline{y}$ does not interfere with the conformance, we can conclude that $\mathcal{T}(T'_r)G \downharpoonleft_r = LP$. Since all the modified type of other components remain the same, by Theorem 3.5.1 we can conclude that $K' \Downarrow T'$.

If the case 2 holds, T can output a value but T_r cannot. This means that during the type extraction we capture values that are flowing. Since the port y can output, this means that all its dependencies are satisfied. However, since $F = y \leftarrow \overline{y}, F'$, but $\overline{y}$ still has some unsatisfied dependencies, the only possible case is that $\overline{y}$ still needs to receive the values from the ports in $\mathit{fp}(LP)$:

Assume that there is one input port, e.g., $z' \in \mathit{fp}(LP)$ (without a loss of generality) such that $\overline{y}$ depends on it, and on which does not have the dependency satisfied.

Let $LP = G \downharpoonleft_r$ and $T_r =< Z_b; \mathbf{C}_r >$ where

$$\mathbf{C}_r = \{\overline{y}(b) : \mathbf{B}_r : [\mathbf{D}_r]\} \uplus \mathbf{C}'_r.$$

We have the case where $\mathbf{D}_r = \{z' : M\} \uplus \mathbf{D}'_r \wedge (M = 0 \vee M = \Omega)$.

Since $\mathcal{T}(T_r) \bowtie LP$ and $z' \in \mathit{fp}(LP)$ then we can write that $\mathcal{T}(F, T_r) \bowtie \mathcal{C}[z'(b').LP']$. This implies that $LP' = G' \downarrow_r$ where $G \to \cdots \xrightarrow{r\ell(v')} G'$. Since $K \Downarrow T$ by induction hypothesis all the subcomponents enjoy the type fidelity property, by the ?THM? **??** exists a trace such that $K \xrightarrow{\tau} \cdots \xrightarrow{\tau} = [\tilde{x} > \tilde{y}]\{G'; r = K'_r, R; D; r[F]\} = K''$ and we have that exists some T'_r such that $K'_r \Downarrow T'_r$. Applying the Rule [$\mathit{InpConf}$] (possibly multiple times, together with the Rule [$\mathit{OutConf}$]) we have that $\mathcal{T}(F, T'_r) \bowtie G' \downharpoonleft_r$ which implies that we had an input on z'. Since all the dependencies of $\overline{y}$ are satisfied now, we conclude based on the case 1) that exists K' such that $K'' \xrightarrow{\text{y!(v)}} K'$. Then we have $K \xRightarrow{\text{y!(v)}}$ and the Theorem 3.5.1 (applied multiple times) exists T' such that $K' \Downarrow T'$.

$\square$

Chapter 4

The IC type language

In this chapter we introduce the IC type language that characterises the reactive behaviour of components modelled in (full) GC language. First we intuitively introduce our type language through a motivating example and then we formalise the language.

4.1 Informal introduction of IC type language

In this section we provide an overview of our type language and of our type extraction procedure through an example. In particular, we start from base components and show how they can be assembled together with a protocol resulting in a new component. For each component we give a brief description of its type and describe how it is extracted from the code.

4.1.1 A passport renewal system

Consider a person who wants to set up an appointment for passport renewal. Assume that the person can start the renewal procedure by using an online *Passport renewal* system made of different interacting services. Hereafter, for simplicity, we focus only on two of them: a *Request handler* and an *Administration* service. The person uploads the *Passport number*, the *Name*, the *Renewal reason* and the person's *Available date* to the system, which handle them through to *Request handler* service. When the first three pieces of information are collected, they are joined together and sent to other system services, e.g., for logging the request, which we do not model. Moreover, some of the uploaded data are further elaborated. As soon as the *Passport number* is uploaded to *Request handler*, the expiration date of the passport is compared with the current one to check if the renewal request can be accepted or not. The result of this check is sent to *Administration*. If the request is accepted, *Request handler* sends the previously uploaded person's *Available date* to *Administration*, after which it sends also the message containing the *Available dates* that are self-generated (working days, excluding weekends, holidays, etc.). However, if the renewal request is denied, *Request handler* ends the interaction with *Administration*, and the person does not get the appointment. The *Administration* service computes the best date for the appointment based on the received information. It sends them to *Request handler* that finally forwards the final result to the person.

4.1.2 Modeling the system in GC language

We model the system presented above as a composite component in the GC language. In particular, we implement *Request handler* and *Administration* through the two base components K_{rh} and K_a, and the whole *Passport renewal* system through the composite component K_{pr}. Figure 5 shows an architectural representation of K_{pr} highlighting how the components are wired together, whereas Figure 6 shows an activity-like diagram describing the workflow of the behaviour of the components.

As described above, crucial to the behaviour of K_{pr} is the interaction between K_{rh} and K_a that must obey to the following protocol G:

$$G = RH \xrightarrow{a/d} ADM(RH \xrightarrow{date} ADM; RH \xrightarrow{Av.d.} ADM; ADM \xrightarrow{App.d.} RH, \mathbf{end})$$

where RH stands for *Request handler*, ADM for *Administration*, *a/d* for *Approval/ Disapproval*, *Av.d.* for *Available date* and *App.date* for *appointment date*. (Note that we use a different font for the roles in order to distinguish them from the values.) The protocol G requires that if *Request handler* sends the *Approval* to *Administration*, it also needs to send *Date* (referring to person's available date) and then the computed *Available dates*; once received these pieces of information *Administration* replays with the generated *Appointment date*. Otherwise, if *Request handler* sends the disapproval to *Administration*, ends the communication.

Finally, note that *Request handler* communicates with both internal (*Administration*) and external environment (*Person*), thus, K_{rh} will be the

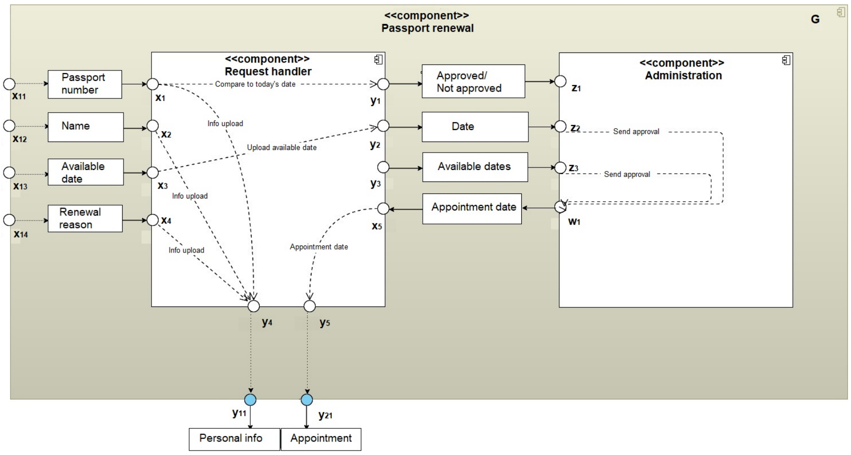

Figure 5: Passport renewal

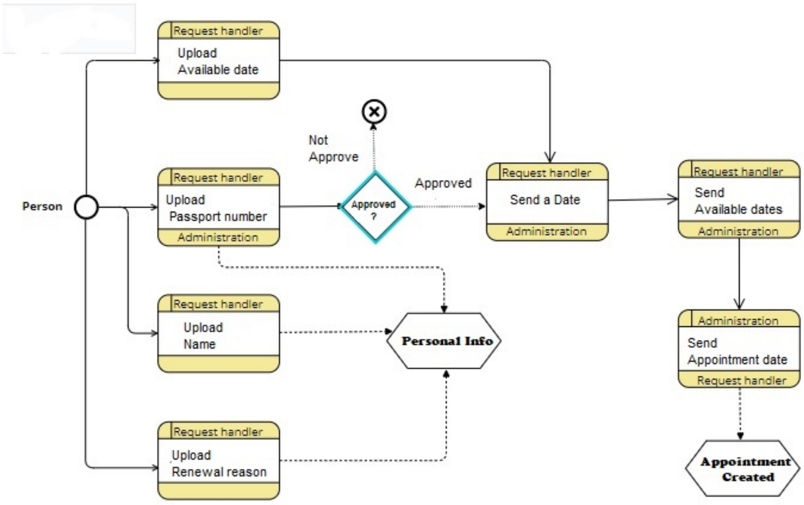

Figure 6: Passport renewal data flow

interfacing component of K_{pr}. Below we first focus on describing the implementation of K_{rh} in GC language (the implementation of K_a is similar), and then we describe how obtaining K_{pr} from K_{rh} and K_a.

The definition of component K_{rh} A graphical representation of K_{rh} is in Figure 5 where to each port is attached a description of the received and produced values. The component K_{rh} has five input ports x_1, x_2,x_3, x_4 and x_5 and five output ports y_1, y_2, y_3, y_4 and y_5. In order to provide a *Personal info* via port y_4, K_{rh} needs *Passport number*, *Name* and *Renewal reason* from x_1,x_2 and x_4, respectively. We say that port y_4 depends on x_1,x_2 and x_4. Also, port y_1 depends on port x_1 because the component needs to receive *Passport number* before emitting the approval or disapproval. The component can output *Date* from port y_2 only when a person's *Available date* is received on port x_3. Similarly, to produce a *Appointment* from port y_5 an *Appointment date* on port x_5 is needed. However, K_{rh} can always produce a list of *Available dates* for the appointment on port y_3 because this value does not depend on any input.

Using GC language, the implementation of component K_{rh} is the following:

$$K_{rh} = [I]\{W\}$$

- ◇ $I = x_1, x_2, x_3, x_4.x_5\rangle y_1, y_2, y_3, y_4, y_5$
- ◇ $W = y_1 = f_{cd}(x_1), y_2 = f_{uad}(x_3), y_3 = f(), y_4 = f_{iu}(x_1, x_2, x_4), y_5 = f_{ad}(x_5)$

where the interface I of K_{rh} specifies the input and output ports described above; the implementation W lists of the local binders defining how compute the produced values; and f_{cd} stands for function *Compare to today's date*, f_{uad} for *Upload available date*, f_{iu} for *Info upload* and f_{ad} for *Appointment date*. For example, the local binder $y_1 = f_{cd}(x_1)$ attaches the function f_{cd} to decide the approval of the request based on the value of x_1. By inspecting the implementation of a base component it is immediate to derive the direct dependencies among ports. The definition of K_a is similar and we do not show it for brevity.

The definition of K_{pr} Component K_{pr} has its own interface exposing the input ports x_{11}, x_{12}, x_{13} and x_{14} and the output ports y_{11} and y_{21}. As we said, these values are directly forwarded to/from the interfacing component K_{rh}. The internal communication between the components K_{rh} and K_a is governed by the previously described protocol G.

Using GC language, K_{pr} can be implemented as:

$$K_{pr} = [x_{11}, x_{12}, x_{13}, x_{14} \rangle\, y_{11}, y_{21}]\{G; R; D; r[F]\}$$

- ◇ $R = RH = K_{rh}, ADM = K_a$ (Role assignments)
- ◇ $D = ADM.z_1 \xleftarrow{a/d} RH.y_1, ADM.z_2 \xleftarrow{date} RH.y_2, ADM.z_3 \xleftarrow{Av.d} RH.y_3$,
  $RH.x_5 \xleftarrow{App.d} ADM.w_1$ (Distribution binders)
- ◇ $F = x_1 \leftarrow x_{11}$, $x_2 \leftarrow x_{12}$, $x_3 \leftarrow x_{13}$, $x_4 \leftarrow x_{14}$, $y_{11} \leftarrow y_4$ and $y_{21} \leftarrow y_5$ (Forwarders)

the role assignments R assigns the roles RH and ADM to K_{rh} and K_A, respectively; the distribution binders D specify the connections between the ports of K_{rh} and K_a, e.g., the input port z_1 of K_a is connected to the output port y_1 of K_{rh}; the forwarders F specify the connections between K_{pr} and the interfacing component.

4.1.3 Extracting types of components

We use types to capture finitely the reactive behaviour of a component. In particular, we provide a type extraction procedure that infers the type of a component from its definition in GC language. The extracted type is enough to obtain the relevant information and reason about the component's behaviour without looking at its code anymore.

Below, we show first how to extract the type of the interfacing component K_{rh}, and then of the whole system K_{pr}.

Type extraction for Request handler Our types are the concatenation of six matrices: the first and next-to-last matrices specify the component interface; the second, the third, and the fourth matrices describe the dependencies of the output ports of the component on the input ones; the last matrix specifies the choice port (immaterial for base components).

The type of component K_{rh} is the following:

$$T_{rh} = X_b^{rh}.\mathbb{L}^{rh}.\mathbb{R}^{rh}.\mathbb{F}^{rh}.Y_b^{rh}[\mathtt{ch}^{rh}]$$

1. $X_b^{rh} = \begin{bmatrix} x_1(number) & x_2(name) & x_3(date) & x_4(reason) & x_5(date) \end{bmatrix}$

2. $\mathbb{L}^{rh} = \{0\}_{5x5}$

3. $\mathbb{R}^{rh} = \{0\}_{5x5}$

4. $\mathbb{F}^{rh} = \begin{bmatrix} N_{\mathbb{F}11}:0 & 0 & 0 & 0 & 0 \\ 0 & 0 & N_{\mathbb{F}23}:0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ N_{\mathbb{F}41}:0 & N_{\mathbb{F}42}:0 & 0 & N_{\mathbb{F}44}:0 & 0 \\ 0 & 0 & 0 & 0 & N_{\mathbb{F}55}:0 \end{bmatrix}$

5. $Y_b^{rh} = \begin{bmatrix} y_1(y/n):\infty \\ y_2(date):\infty \\ y_3(date):\infty \\ y_4(info):\infty \\ y_5(date):\infty \end{bmatrix}$

6. $\left[\mathrm{ch}^{rh}\right] = \left[0\right]$

The matrix X_b^{rh} consists of a single row whose length is equal to the number of input ports. Each element of this matrix stores the association between the corresponding input port and the type of values that can be received on it. For example, the element $x_1(number)$ denotes the fact that the input port x_1 can receive a numeric value.

The matrices $\mathbb{L}^{rh}$ and $\mathbb{R}^{rh}$ are always zero matrices for base components (they are used to describe dependencies built upon the choice).

The matrix $\mathbb{F}^{rh}$ contains the dependencies that exist only observing the interfacing component, where none of the input values needed for computing the output value goes through the protocol.

Since K_{rh} is a base component, the dependencies either do not exist (0 element of the matrix) or are "per-each-value" kind ($N_{\mathbb{F}ij} : 0$). This kind of dependency encodes the situation where a component needs to receive values from all the input ports it depends on before producing a value on some output port y.

More precisely, an element $N_{\mathbb{F}ij} : 0$ represents a dependency for i^{th} element of the matrix Y_b^{rh} from the j^{th} element of the matrix X_b. For example, the element $N_{\mathbb{F}23}:0$ above encode the fact that y_2 depends x_3.

The zero next to $N_{\mathbb{F}ij}$ represents the number of the so-called *possible values* (always 0 for base components). These values are relevant for the types of composite components when the protocol includes choices. In this case, while extracting the type, we cannot ensure that some value will arrive for computing the output value, but we know that depending on a choice, it can happen.

Note that we are typing our components statically. For this reason, both $N_{\alpha ij}$ and $p_{\alpha ij}$ are equal to zero while extracting the type. However, both represent the number of values received for computing the corresponding output value.

The first one $N_{\alpha ij}$ expresses the number of received values that for sure will eventually be used for computing an output value. Whereas, $p_{\alpha ij}$ counts the received values that might be used for computing the output depending on a choice made by the component. Both counters provide us with the number of values available at run-time for a given port and they help us in proving the safety of our language.

The matrix Y_b^{rh} gives information on the types of the values produced by the output ports and on the maximum number of values that can be output. For example, the element $y_2(date)\!:\!\infty$ informs that from the port y_2 the component can produce values of the type $date$ unlimited number of times.

The last matrix declares the port in charge of sending or receiving the choice (for base components is always 0).

Once we have the type of the interfacing component, we can extract the type of the component K_{pr} (Figure 5) representing the whole system.

The type extraction for Passport renewal Since the input ports can always receive values, we focus on which conditions a composite component can output a value in our type extraction procedure. We extract the type of a component statically when values are yet to arrive. Consider first the output port y_{11}. From the graphical representation of component K_{pr} in Figure 5, we have that the values received on the ports x_{11}, x_{12} and x_{14} are needed for computing the output for port y_{11} (per-each-value dependencies). We also note that these values are forwarded to/from the interfacing component from/to an external environment without passing through the protocol. Thus, we store the information about these dependencies in the matrix $\mathbb{F}$ whose first row regards y_{11} (we discuss below the second row regarding y_{21}):

$$\mathbb{F} = \begin{bmatrix} \boxed{0{:}0} & \boxed{0{:}0} & 0 & \boxed{0{:}0} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The "box-emphasised" entries $(0:0)$ announce the per-each-value dependencies, where the values are still yet to be received. Instead, zero (0) entries announce the absence of dependencies.

Now consider the output port y_{21}. From the definition of K_{rh}, we have that the values produced by this port come from y_5 of K_{rh}. Hence, we actually need to understand under which condition y_5 produces values.

For this aim, recall the definition of the protocol G:

$$G = RH \xrightarrow{a/d} ADM(RH \xrightarrow{date} ADM; RH \xrightarrow{Av.d.} ADM; ADM \xrightarrow{App.d.} RH, \mathbf{end})$$

and the matrices making up the type of component K_{rh}:

$$T_{rh} = X_b^{rh}.\mathbb{L}^{rh}.\mathbb{R}^{rh}.\mathbb{F}^{rh}.Y_b^{rh}[\mathtt{ch}].$$

Consider also the local protocol LP returned by the protocol projection operator and representing the behaviour of K_{rh} driven by protocol G:

$$LP = y_1!(a/d)(y_2!(date).y_3!(Av.d.).x_5?(App.d.).\mathbf{end}, \mathbf{end}).$$

From the matrix $\mathbb{F}^{rh}$ of T_{rh} (the relevant entries are in red in the PDF), we have the dependency $N_{55} : 0$, meaning that the output port y_5 needs values received on x_5. Inspecting the definition of the local protocol LP from right to left, we observe that K_{rh} receives values on port x_5 after others interactions with K_a. These interactions may introduce further dependencies. In particular:

- the input on x_5 is preceded by an output on y_3; from the matrix $\mathbb{F}^{rh}$ of T_{rh}, this output does not depend on any inputs;
- the output on y_3 is preceded by an output on y_2; from the matrix $\mathbb{F}^{rh}$ of T_{rh}, we know that y_2 depends on external input port x_3;
- the output on y_2 is preceded by output on y_1; from the matrix $\mathbb{F}^{rh}$ of T_{rh} we have that port y_1 depends on the external port x_1.

Thus, these interactions create transitive dependencies of y_{21} on ports x_{13} and x_{11}. The steps to compute the dependencies described above are summarised in Table 13.

Need	Requirements	Observation
$y_{21}!$ (forwarded from y_5)	$x_5?$	According to T_{rh}
$x_5?$	$y_3!$	According to LP
$y_3!$	$y_2!$	According to LP
$y_2!$	$x_{13}?$ (forwards to x_3)	According to T_{rh}
	$y_1!$	According to LP
$x_{13}?$	no requirements	
$y_1!$	$x_{11}?$ (forwards to $x_1?$)	According to T_{rh}
x_{11}	right choice (*Approval*)	According to LP

Table 13: Creating transitive dependencies-summary

Moreover, from local protocol LP we have that an output on y_1 is followed by a choice between the termination of interaction (**end**) or the continuation of communication through the sub-protocol ($y_2!(date).y_3!(Av.d.).x_5?(App.d.).\mathbf{end}$). Note also that a reception of a value on port x_5 is only possible if K_{pr} receives the approval of the passport renewal and the right side of the protocol is executed.

The branch of the protocol to choose depends on the values produced by y_1 which result from the computation of function f_{cd} on values from port x_1. In turn, x_1 receives values from port x_{11} (via forwarders), so we mark it as a choice port because it determines the branch taken by LP.

Therefore, all the dependencies of y_{21} (obtained via y_5) are created observing LP. Since selecting the "left side" of LP we do not have further communications, the matrix $\mathbb{L}$ storing the left dependencies is zero. Instead, when we select the "right side" of the protocol, y_5 depends on ports x_{13} and x_{11}. Thus, we store this information in the matrix $\mathbb{R}$. Moreover, note that our protocol G is a one-shot protocol (once the communication is established, there are no further communications), thus, the obtained dependencies are initial ones. This means that the dependency is dropped once an output port receives a value from the input port on which it initially depends. Moreover, since x_5 can receive only a value according to LP, the maximum number of values that y_{21} can emit is 1, and we record this bound in the matrix Y_b.

Summing up, we "box-emphasise" the non-zero dependencies in matrices $\mathbb{R}$ and $\mathbb{F}$ and the type of component K_{pr} is the following:

$$T_{pr} = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}]$$

where

1. $X_b = \begin{bmatrix} x_{11}(number) & x_{12}(name) & x_{13}(date) & x_{14}(reason) \end{bmatrix}$
2. $\mathbb{L} = \{0\}_{2x4}$
3. $\mathbb{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \boxed{\boxed{\Omega{:}0}} & 0 & \boxed{\boxed{\Omega{:}0}} & 0 \end{bmatrix}$
4. $\mathbb{F} = \begin{bmatrix} \boxed{0{:}0} & \boxed{0{:}0} & 0 & \boxed{0{:}0} \\ 0 & 0 & 0 & 0 \end{bmatrix}$
5. $Y_b = \begin{bmatrix} y_{11}(\mathit{info}){:}\infty \\ y_{21}(date){:}1 \end{bmatrix}$
6. $\begin{bmatrix}\mathtt{ch}\end{bmatrix} = \begin{bmatrix} x_{11} \end{bmatrix}$

In the matrices above we used red and green colors to distinguish the non-zero dependencies of port y_{11} and port y_{21}, respectively. In the matrix $\mathbb{R}$ we use the notation $\Omega : p_{ij}$ to represent an initial dependency, where $p_{ij} = 0$ is the number of values received on the j^{th} input port. However, these values will be used to compute the output value for the i^{th} output port, if the component choices the right branch of the protocol. Moreover, since the arrival of values for the computation of the output port y_{11} is not restricted by protocol, y_{11} can produce an unbounded number of values (denoted with the bound ∞ in the matrix Y_b). Note that the basic types of the interface ports of K_{pr} match those of the interface ports of K_{rh} to/from whom values are forwarded (e.g. $x_1 \leftarrow x_{11}$).

Note that component K_a can have further interactions with other components, but our focus here is on the interfacing one that interacts with internal and external environment (K_{rh} in this case). Moreover, K_{rh} and K_a can interact in a different way from the one we described. For example, we can define a protocol that excludes the choice and lets a person get an appointment regardless of approval/disapproval message. The protocol we considered captures the most relevant scenario for introducing our proposal.

Extending the passport system

Here, we extend the scenario presented above to the case where we want to assemble components using a recursive protocol.

Passports are usually valid for ten years, and a person can lose her passport more than once during this period. We define a new component that uses the Passport renewal system above to book an appointment when a person loses her passport. In this case, the person gives only the *Name* and the *Renewal reason* to the *Renewal passport* service, whereas the new *Passport number* and the available date are provided by another component, called *Official database*. The *Official database* sends the *Available date* when the employee in charge of paperwork is at disposal. In the end, the new component produces as output the *Appointment* for the person, together with the *Personal info* collected if needed for further communication.

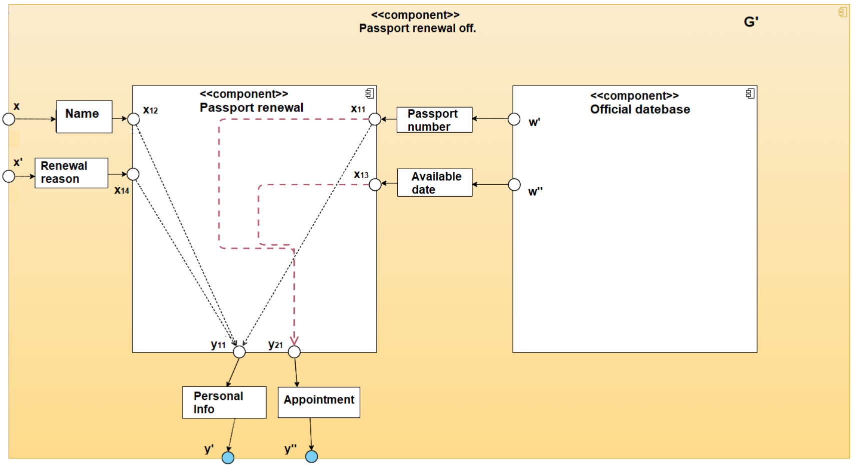

Figure 7: Passport renewal off.

Definition of the new component Below we assume to have a base component K_{od} for *Official database* and we compose it with K_{pr} introduced above to obtain a new composite component K_{pro} (see Figure 7). The communication between *Passport renewal* and *Official database* is the following: each time a new person requests a new passport, $Official\ database$ sends the *Passport number* to $Passport\ renewal$, after which sends the *Available date*. Formally, this protocol can be written as:

$$G' = \mu\mathbf{X}.OD \xrightarrow{pass.num} PR; OD \xrightarrow{Av.d} PR.\mathbf{X}$$

where OD stands for $Official\ database$ and PR for $Passport\ renewal$. We assign the component K_{pr} to the role $Passport\ renewal$, and K_{od} to the role *Official database*.

The definition of K_{pro} in GC language is the following:

$$K_{pro} = [x, x' \rangle\, y', y''] \{G', R', D', r[F']\}$$

where

◇ $R' = PR = K_{pr}, OD = K_{od}$ (Role assignments)

◇ $D' = PR.x_{11} \xleftarrow{pass.num} OD.w', PR.x_{13} \xleftarrow{Av.d.} OD.w''$ (Distribution binders)

◇ $F' = x_{12} \leftarrow x, x_{14} \leftarrow x', y' \leftarrow y_{11}$ and $y'' \leftarrow y_{21}$ (Forwarders)

The component K_{pro} has two input ports x and x' and two output ports y and y'; the input port forward values to ports x_{12} and x_{14} of K_{pr}; the output ports y and y' receive values from y_{11} and y_{21}, respectively.

Type extraction for K_{pro} Extracting the type of a composite component is quite challenging, especially in the case where we compose composite components with others. The problem comes when we have an interfacing component whose type has a choice port different from zero. In those cases, we need to restrict how to compose such components. The reason behind this restriction is to ensure that we can extract the choice port from the new-built component for further composition.

Consider the type T_{pr} of the interfacing component K_{pr} described above, and the local protocol LP' obtained projecting G' on K_{pr}:

$$LP' = \mu.\mathbf{X}.x_{11}?(num).x_{13}?(date).\mathbf{X}$$

The choice port x_{11} of component K_{pr} is the part of the description of LP'. This means that x_{11} is the port in charge of the internal communication. In this case, we need to have an external port that announces the choice made internally, and we do it by checking the following:

- There exists some external output port that depends on that input choice port (the external output port y_{i1} depends on x_{11} according to T_{pr});

- If such a port exists, it must be the case that their dependency does not depend on internal choices of component K_{pr} (formally, $\forall_{1\leq k\leq 4} : d^{\mathbb{L}}_{ik} = 0 \wedge d^{\mathbb{R}}_{ik} = 0$, see Section 4.2). This is true according to T_{pr}.

Now we can say that in the extracted type of component K_{pro} we find the port y_{11} as the choice port. The set of the interfacing input ports is $\{x_{12}, x_{14}\}$, where x_{12} receives values from port x and x_{14} from x'.

To extract the type, we need to consider under which conditions the output ports y' and y'' can produce values. First, we focus on port y'. As prescribed by forwarders F, this port gets values from port y_{11} of K_{pr}. In turn, from the matrix $\mathbb{F}$ of T_{pr} we have that port y_{11} depends on both x_{12} and x_{14}, and consequently on the two external ports x and x'.

Since the protocol G' is not branching, we have that each time a value is received on x_{12} (from x) and on x_{14} (from x'), it used for computing the value of y_{11} (and of y' consequently). We observe only the dependencies on the external ports, so we can say that y' depends on x and x', where each time a value is received on both ports, y' can perform an output. Moreover, since the protocol is recursive, each time a value is received on x_{11}, it is used for computing y_{11} (y').

Since there is no bound on the number of values that can be received (the external environment can continually produce new ones, and the recursive behaviour of the protocol enables reception of new values in each iteration), the component is able produce unlimited number of values from port y'.

Consider now port y'', which receives values from port y_{21}. According to the type of K_{pr} (matrices $\mathbb{R}$ and $[\texttt{ch}]$), y_{21} depends on x_{11} and x_{13} and receives values from them only if the internal protocol of K_{pr} makes the adequate choice. Both ports x_{11} and x_{13} are used for internal communication between components K_{pr} and K_{od}. Since local protocol LP' is recursive, we get a value on x_{11} and x_{13} for computing a value for y_{21} in each iteration. However, the dependency between x_{11} (x_{13}) and y_{21} in matrix $\mathbb{R}$ of T_{pr} is an initial one: it tells us that if the component makes the right choice the dependency is dropped, after receiving one value. Moreover, from T_{pr} we know that y_{21} can produce only one value, it does not depend on any interfacing port, that we capture in our types, but depends on a choice.

In this case, we make a per-each-value dependency on the interfacing input ports for y_{21}, giving the unbounded number of possible values, where when the right choice is made once, one of the possible values becomes the actual value used for computing the output on y_{21}.

Summing up, the type of component K_{pro} is then:

$$T_{pr} = X'_b.\mathbb{L}'.\mathbb{R}'.\mathbb{F}'.Y'_b.[\mathtt{ch}']$$

1. $X'_b = \begin{bmatrix} x(name) & x'(reason) \end{bmatrix}$
2. $\mathbb{L}' = \{0\}_{2x2}$
3. $\mathbb{R}' = \begin{bmatrix} 0 & 0 \\ 0{:}\infty & 0{:}\infty \end{bmatrix}$
4. $\mathbb{F}' = \begin{bmatrix} 0{:}0 & 0{:}0 \\ 0 & 0 \end{bmatrix}$
5. $Y'_b = \begin{bmatrix} y'(info){:}\infty \\ y''(date){:}1 \end{bmatrix}$
6. $\begin{bmatrix} \mathtt{ch}' \end{bmatrix} = \begin{bmatrix} y' \end{bmatrix}$

Matrix $\mathbb{R}'$ contains the abstract dependencies $0:\infty$, announcing that the port y'' does not actually depend on ports x and x', but on the choice, and that it is going to be able to produce a value if the component makes a right choice.

Remark. Components from the example of Section 3.1 can be described in IC type language, where components from this section cannot be described in EC type language, because of the lack of the information in the type description. We can say that the amount of components that can be typed in EC language is a subset of those that can be typed in IC type language.

We can "translate" the first type language into the second one.

Example 4.1.1. *Let us recall the type from Subsection 3.1*

$$\begin{aligned}
&T_{Portal} =< X_b; \boldsymbol{C} > \\
&X_b = \{x_p(image), x'_p(class)\} \\
&\boldsymbol{C} = \{C_1, C_2, C_3\} \\
&C_1 = y_p(image){:}\infty{:}[\{x_p{:}N_p\}] \\
&C_2 = y'_p(class){:}\infty{:}[\{x'_p{:}N'_p\}] \\
&C_3 = y''_p(version){:}\infty{:}[\emptyset]
\end{aligned}$$

This type in IC language is translated into:
$T_{Portal} = X_b.[0].[0].\mathbb{F}.Y_b.[0]$

$$X_b = \{x_p(image), x'_p(class)\}$$

$$\mathbb{F} = \begin{bmatrix} N_p\!:\!0 & 0\!:\!0 \\ 0\!:\!0 & N'_p\!:\!0 \\ 0\!:\!0 & 0\!:\!0 \end{bmatrix}$$

$$Y_b = \begin{bmatrix} y_p(image)\!:\!\infty \\ y'_p(class)\!:\!\infty \\ y''_p(version)\!:\!\infty \end{bmatrix}$$

4.2 Formal introduction of IC type language

In this section we present the syntax and the semantics of the language. Then, in the next sections we present two procedures that define how to extract the type of a component. The first procedure is for base, and the second one is for composite components.

4.2.1 Syntax of IC type language

In this section, we formally present our type language to describe the behaviour of GC components. We first introduce the syntax and then, in the following section, the operational semantics of our types.

The syntax of our types is in Table 14. The type of a component is a sequence of five matrices and a *choice port*. The first X_b and the last matrix Y_b represent the interface of a component. The matrix X_b is a row matrix that has m elements representing the input ports of the component and the basic types b_{1j} of the values that can be received. Similarly, Y_b is a column matrix of size n and contains the output ports with the basic types b_{i1} of values they produce and their capabilities $\mathbf{B}_{i1}$. A capability represents the maximum number of values that an output port can produce at run-time. We denote with N a finite bound and with ∞ an unbounded capability.

The middle three matrices ranged over $\alpha = \{\mathbb{L}, \mathbb{R}, \mathbb{F}\}$ encode the dependencies among ports. In the case of composite components, matrices $\mathbb{L}$ and $\mathbb{R}$ record the dependencies on the choices made by a protocol or an internal component. The existence of one entails the existence of the other one. If both are zero, emitting values from any output port does not depend on a choice. The matrix $\mathbb{F}$ instead represents dependencies that exist regardless of any choice. Note that these matrices are always zero for base components but not for composite ones.

All three matrices are of the same size $n \times m$, where m is the number of columns of X_b, and n is the number of rows of Y_b. The underlying idea is that k^{th} column refers to a k^{th} element of X_b, and that the q^{th} row refers to the q^{th} element of Y_b. In this way, an element $d^{\alpha}_{i,j}$ in the matrix α represents the dependency of the output port y_{i1} on the input port x_{1j}.

The dependencies can be of three kinds: a *zero dependency* denoted by 0 means the absence of dependency; a *initial one*, in symbols $\Omega : p_{\alpha ij}$, says that the port y_{i1} needs only one *actual* value received on x_j for producing its first value, after which the dependency is dropped (observing the matrix α); and *per-each-value* dependency, in symbols $N_{\alpha ij} : p_{\alpha ij}$, says that each value emitted by y_i needs one *actual* value received on x_j. When we type a component at a static time, $N_{\alpha ij}$ is always zero, but at run-time, it tracks the number of values available on x_{1j}. This tracking helps prove the safety of our language.

A value $p_{\alpha ij}$ appearing in a dependency is called *possible value*, and counts the number of values received on x_{1j} that are possibly used for y_{i1}. More precisely, $p_{\alpha ij}$ represents the values that could reach y_{i1} depending on the choices made by an internal component or by the protocol. Like $N_{\alpha ij}$, $p_{\alpha ij}$ is zero at static time, and it is incremented at run-time.

The last syntactic element of a type is the so-called *choice* port `ch` that says from which port we get the information about the choice made internally by a component. Note that this port is one belonging to the interface of the component or it is zero when there is no choice.

4.2.2 Semantics of IC type language

We now describe the operational semantics of our type language needed to show that our types truly capture component behaviour. The semantics is given by means of a LTS whose labels λ are defined by the following grammar:

$$\lambda = x_{1j}? \mid x_{1j}(b_{1j})? \mid y_{i1}(b_{i1})! \mid \tau.$$

The label $x_{1j}?$ denotes an input on port x_{1j} where we do not record the type of the value; $x_{1j}(b_{1j})?$ represents an input of a value of type b_{1j}; label $y_{i1}!$ is an output from the port y_{i1} of any type and $y_{i1}(b_{i1})!$ represents an output of a value of type b; finally, τ captures an internal step.

Type	$T \triangleq X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\texttt{ch}]$
Input interfaces	$X_b \triangleq \{x_{1j}(b_{1j})\}_{1\times m}$
Output interfaces	$Y_b \triangleq \{y_{i1}(b_{i1}) : \mathbf{B}_{i1}\}_{n\times 1}$
Choice-related dependencies	$\mathbb{L}.\mathbb{R} \triangleq \{d^{\mathbb{L}}_{ij}\}_{n\times m}.\{d^{\mathbb{R}}_{ij}\}_{n\times m}$
Choice-free dependencies	$\mathbb{F} = \{d^{\mathbb{F}}_{ij}\}_{n\times m}$
Dependencies	$d^{\alpha}_{ij} ::= 0 \mid \Omega : p_{\alpha ij} \mid N_{\alpha ij} : p_{\alpha ij}$
Choice port	$\texttt{ch} ::= 0 \mid x_{1j}(b_{1j}) \in X_b \mid y_{i1}(b_{i1}) : \mathbf{B}_{i1} \in Y_b$
Boundary	$\mathbf{B}_{i1}, p_{\alpha ij} ::= N \mid \infty$
Additional variables	$N, N_{\alpha ij} \in \mathbb{N}_0 \quad \alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\}$
	$i \in \{1, ..., n\}, j \in \{1, ..., m\}; m, n \in \mathbb{N}$

Table 14: Type syntax (IC type language)

Let $A = \{a_{ij}\}_{n\times n}$ be a matrix, we denote with $[a_{kj}]$ and with (a_{ik}) the k^{th} row and the k^{th} column of A, respectively.

We now introduce an auxiliary operation that we use in the semantic rules to decrement the number of values available in a per-each-values dependency. Let $T = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\texttt{ch}]$ be a type, where $Y_b = (y_{k1}(b_{k1}) : \mathbf{B}_{k1}), 1 \leq k \leq n$. Let k be the index of the row containing a per-each-value dependencies $(N_{\alpha kj} : p_{\alpha kj})$ or zeros for all $\alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\}$; let $N_{\alpha kj} > 0$ for all $j \in \{1, ..., m\}$ and $\mathbf{B}_{k1} > 0$. We define $shrink(\mathbb{L}, \mathbb{R}, \mathbb{F}, Y_b, k)$ on the elements of a type as follow (we overload the $shrink$ to work also on matrices and its elements):

$$shrink(\mathbb{L}, \mathbb{R}, \mathbb{F}, Y_b, k) \triangleq$$

$$shrink(\mathbb{L}, k).shrink(\mathbb{R}, k).shrink(\mathbb{F}, k).shrink(Y_b, k)$$

$$shrink(N_{\alpha kj} : p_{\alpha kj}, k) \triangleq \quad N_{\alpha kj} - 1 : p_{\alpha kj}$$

$$shrink(N_{\alpha kj} : p_{\alpha kj}, s) \triangleq \quad N_{\alpha kj} : p_{\alpha kj} \qquad s \neq k$$

$$shrink(0, k) \triangleq \quad 0$$

$$shrink([d^{\alpha}_{kj}], k) \triangleq \quad [shrink(d^{\alpha}_{kj}, k)]$$

$$shrink(\alpha, k) \triangleq \begin{bmatrix} [d^{\alpha}_{1j}] \\ \vdots \\ shrink([d^{\alpha}_{kj}], k) \\ \vdots \\ [d^{\alpha}_{nj}] \end{bmatrix}$$

$$shrink(Y_b, k) \triangleq \begin{bmatrix} y_{11}(b_{11}) : \mathbf{B}_{11} \\ \vdots \\ y_{k1}(b_{k1}) : \mathbf{B}_{k1} - 1 \\ \vdots \\ y_{n1}(b_{n1}) : \mathbf{B}_{n1} \end{bmatrix}$$

Intuitively, the operation defined above goes inside the structure of each matrix, except the matrix X_b. If a dependency is found in any of the matrices $\mathbb{L}, \mathbb{R}$ or $\mathbb{F}$ is zero, the *shrink* operation returns the zero as it was. Otherwise, this operation decrements two counters: the number of values received from the per-each-value dependencies and the number of values that can be produces from the input port, all of them of the row index by k.

Table 15 shows the rules of the type semantics. We have two levels of rules: those that describe how the entry of a matrix evolves and those that describe the behaviour of types. A description of the rules of the first level is in order. Rule [`InpZero`] says that if the element in the j^{th} column of a matrix $\mathbb{L}, \mathbb{R}$ or $\mathbb{F}$ is zero, it remains zero after an input of a j^{th} element of a matrix X_b. In other words, if y_{i1} had no dependencies on x_{1j}, the input on that port does not create a new one. Rule [`InpDisc`] states that the input of the k^{th} element from the matrix X_b does not have an impact on the dependencies found in a column different from k in the dependency matrices. This comes from labelling the dependencies (d_{ij}) in such a way that the column index (j) points to the input port on which the output port pointed with the row index (j) depends on. Rules [`InpIntF`] and [`InpPevF`] describe dependencies that exist regardless of a choice. If the dependency is initial (rule [`InpIntF`]), after an input on the corresponding port, the dependency is dropped. Instead, if the dependency (rule [`InpPevF`]) is of kind per-each-value, the input increments by one the number of values available for the port y_{ij}.

$$\frac{d^{\alpha}_{ij} = 0}{d^{\alpha}_{ij} \xrightarrow{x_{1j}?} d^{\alpha}_{ij}}\ [\texttt{InpZero}] \qquad \frac{k \neq j}{d^{\alpha}_{ij} \xrightarrow{x_{1k}?} d^{\alpha}_{ij}}\ [\texttt{InpDisc}] \qquad \frac{d^{\mathbb{F}}_{ij} = \Omega : p_{\mathbb{F}ij}}{\Omega : p_{\mathbb{F}ij} \xrightarrow{x_{1j}?} 0}\ [\texttt{InpIntF}]$$

$$\frac{d^{\mathbb{F}}_{ij} = N_{\mathbb{F}ij} : p_{\mathbb{F}ij}}{N_{\mathbb{F}ij} : p_{\mathbb{F}ij} \xrightarrow{x_{1j}?} N_{\mathbb{F}ij} + 1 : p_{\mathbb{F}ij}}\ [\texttt{InpPevF}] \qquad \frac{}{T \xrightarrow{\tau} T}\ [\texttt{InpInter}]$$

$$\frac{d^{\beta}_{ij} = \Omega : p_{\beta ij} \quad \beta \in \{\mathbb{L}, \mathbb{R}\}}{\Omega : p_{\beta ij} \xrightarrow{x_{1j}?} \Omega : p_{\beta ij} + 1}\ [\texttt{InpIntLR}]$$

$$\frac{d^{\beta}_{ij} = N_{\beta ij} : p_{\beta ij} \quad \beta \in \{\mathbb{L}, \mathbb{R}\}}{N_{\beta ij} : p_{\beta ij} \xrightarrow{x_{1j}?} N_{\beta ij} : p_{\beta ij} + 1}\ [\texttt{InpPevLR}]$$

$$\frac{\begin{array}{c}\forall i \in \{1, ..., n\},\ \forall j \in \{1, \ldots, m\}\ \ \alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\} \\ \alpha = \{d^{\alpha}_{ij}\}_{n \times m}\ \ d^{\alpha}_{ij} \xrightarrow{x_{1p}?} d'^{\alpha}_{ij}\ \wedge\ x_{1p}(b_{1p}) \in X_b\end{array}}{X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\texttt{ch}] \xrightarrow{x_{1p}(b_{1j})?} X_b.\{(d^{\mathbb{L}}_{ij})'\}.\{(d^{\mathbb{R}}_{ij})'\}.\{(d^{\mathbb{F}}_{ij})'\}.Y_b[\texttt{ch}]}\ [\texttt{InpT}]$$

$$\frac{\begin{array}{c}\forall j \in \{1, ..., m\}\ \ d^{\alpha}_{ij} = N_{\alpha ij} : p_{\alpha ij} \vee d^{\alpha}_{ij} = 0\ \ \mathbf{B}_{i1}, N_{\alpha ij} > 0 \\ i \in \{1, ..., n\} \quad \wedge \quad y_{i1}(b_{i1}) \ : \ B_{i1} \in Y_b\end{array}}{X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\texttt{ch}] \xrightarrow{y_{i1}(b_{i1})!} X_b.shrink(\mathbb{L}, \mathbb{R}, \mathbb{F}, Y_b, i)[\texttt{ch}]}\ [\texttt{OutT}]$$

Table 15: Type semantics (IC type language)

Instead, when we input on a port x_{1j} and the dependencies in matrices $\mathbb{L}$ and $\mathbb{R}$ are different than zero, they evolve by increasing the number of possible values by one for each entry in their j^{th} column. Note that this increment occurs both when the entry represents a per-each-value dependency (rule [`InpPevLR`]), and the initial one (rule [`InpIntLR`]).

Then, we have the rules that capture the type behaviour. Rule [`InpInter`] always allows a type to perform an internal step τ and to remain unchanged. Rule [`InpT`] deals with input actions: if all the entries of the matrices $\mathbb{L}, \mathbb{R}$ and $\mathbb{F}$ can perform an input on x_{1j}, then the whole type can input on that port a value of type b_{1j} as indicated in the interface. Rule [`OutT`] allows a type to carry out an output on port y_{i1} when the following conditions are met: (i) the $i^t h$ rows of the matrices $\mathbb{L}, \mathbb{R}$ and $\mathbb{F}$ contain only 0 or per-each-value dependencies; (ii) the number of values available for the computation of the i^{th} output port is greater than zero; and (iii) the constraint for the i^{th} output port is greater than zero. After the output the type evolves by updating the entries of matrices $\mathbb{L}, \mathbb{R}, \mathbb{F}$ and Y_b using the $shrink$ operation.

Example 4.2.1. *Let us recall the type of component K_{pr} of Section 4.1*

$$T_{pr} = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}]$$

where

$$X_b = \begin{bmatrix} x_{11}(number) & x_{12}(name) & x_{13}(date) & x_{14}(reason) \end{bmatrix}$$

$$\mathbb{L} = \{0\}_{2x4} \quad \mathbb{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \Omega{:}0 & 0 & \Omega{:}0 & 0 \end{bmatrix} \quad \mathbb{F} = \begin{bmatrix} 0{:}0 & 0{:}0 & 0 & 0{:}0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$Y_b = \begin{bmatrix} y_{11}(info){:}\infty \\ y_{21}(date){:}1 \end{bmatrix} \quad \begin{bmatrix}\mathtt{ch}\end{bmatrix} = \begin{bmatrix} x_{11} \end{bmatrix}$$

This type may evolve upon the reception on the port x_{11} as follows:

$$\frac{d^{\mathbb{L}}_{11} = 0}{d^{\mathbb{L}}_{11} \xrightarrow{x_{11}?} 0}\ [\mathtt{InpZero}] \quad \frac{d^{\mathbb{R}}_{21} = 0}{d^{\mathbb{L}}_{21} \xrightarrow{x_{11}?} 0}\ [\mathtt{InpZero}]$$

$$\frac{d^{\mathbb{R}}_{11} = 0}{d^{\mathbb{R}}_{11} \xrightarrow{x_{11}?} d^{\mathbb{R}}_{11}}\ [\mathtt{InpZero}] \quad \frac{d^{\mathbb{R}}_{21} = \Omega : 0}{d^{\mathbb{R}}_{21} \xrightarrow{x_{11}?} \Omega : 1}\ [\mathtt{InpIntLR}] \quad \frac{d^{\mathbb{F}}_{11} = 0 : 0}{d^{\mathbb{F}}_{11} \xrightarrow{x_{11}?} 1 : 0}\ [\mathtt{InpPevF}]$$

For the rest of dependencies we apply the following rule:

$$\frac{}{d^{\alpha}_{ij} \xrightarrow{x_{1k}?} d^{\alpha}_{ij}}\ [\mathtt{InpDisc}]$$

We now apply the last rule:

$$\frac{\begin{array}{c}\forall i \in \{1,2\},\ \forall j \in \{1,2,3,4,\}\ \forall \alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\} \\ \alpha = \{d^{\alpha}_{ij}\}_{2\times 4}\ \ d^{\alpha}_{ij} \xrightarrow{x_{11}?} d'^{\alpha}_{ij}\ \wedge\ x_{11}(number) \in X_b\end{array}}{X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[x_{11}] \xrightarrow{x_{11}(number)?} X_b.\mathbb{L}'.\mathbb{R}'.\mathbb{F}'.Y_b[x_{11}]}\ [\mathtt{InpT}]$$

where

$$\mathbb{L}' = \{0\}_{2x4} \quad \mathbb{R}' = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \Omega{:}1 & 0 & \Omega{:}0 & 0 \end{bmatrix} \quad \mathbb{F}' = \begin{bmatrix} 1{:}0 & 0{:}0 & 0 & 0{:}0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

4.3 IC type extraction for base components

The *type extraction* is a procedure that given a component in GC language infers its type. Here, we focus on base components keeping in mind their reactive behaviour. Next section describes the procedure for composite components.

The extraction procedure for base components starts by inferring the types of values associated to the communication ports, then it focuses on their dependencies, while checking that their usage is consistent throughout. Hereafter, we assume that every output port is associated with a local binder and that each local binder is associated with a port of the component interface.

In the definition of type extraction we use the auxiliary functions $\gamma(\cdot)$ and *count*$(x, \tilde{\sigma})$. The function $\gamma(\cdot)$ maps values, ports, functions and local binders to their respective types. Hereafter, we also extend γ for lists in which case we obtain the list of respective types, e.g., $\gamma(1, \texttt{hello}) = \mathit{integer}, \mathit{string}$.

The function *count*$(x, \tilde{\sigma})$, given a local binder $y = f(\tilde{x}) < \tilde{\sigma}$, returns the number of values available to y for each of the ports in $\tilde{x}$. The returned value corresponds to the number of elements in $\tilde{\sigma}$ that have a binding for x. Formally, let X be the set of ports and Σ the set of lists of mappings from ports to values. Then, the function *count*: $X \times \Sigma \to \mathbb{N}_0$ is defined as follows:

$$\mathit{count}(x, \tilde{\sigma}) = \begin{cases} j & \text{if } \tilde{\sigma} = \sigma_1, \ldots, \sigma_j, \sigma_{j+1}, \ldots, \sigma_l \ \wedge \\ & \quad x \in \bigcap_{1 \le i \le j} \mathsf{dom}(\sigma_i) \ \wedge \ x \notin \bigcup_{j+1 \le i \le l} \mathsf{dom}(\sigma_i) \\ 0 & \text{otherwise} \end{cases}$$

Note that the mappings in $\tilde{\sigma}$ are handled according to a FIFO discipline, so the first (oldest) mappings are the ones that we need to account for.

Now we can define type extraction for base components:

Definition 4.3.1 (Type Extraction for a Base Component)**.**

Let $C = [\tilde{x} > \tilde{y}]\{y_1 = f_{y_1}(\tilde{x}^{y_1}) < \tilde{\sigma}^{y_1}, \ldots, y_k = f_{y_k}(\tilde{x}^{y_k}) < \tilde{\sigma}^{y_k}\}$ *be a base component, where* $\tilde{y} = y_1, y_2, \ldots, y_k$*. If there exists* γ *such that* $\gamma(\tilde{x}) = \tilde{b}$ *and* $\gamma(y_1) = b'_1, \ldots, \gamma(y_k) = b'_k$ *and provided that* $\gamma(f_{y_i}) = \tilde{b}^{y_i} \to b'_i$ *and that* $\tilde{b}^{y_i} = \gamma(\tilde{x}^{y_i})$ *for any* $i \in 1, \ldots, k$*, then the type of* C *is*

$$X_b.\{0\}_{n \times m}.\{0\}_{n \times m}.\mathbb{F}.Y_b.[0]$$

where

- ◇ $X_b = \{x_{1j}(b_{1j})\}_{1\times m}$ *for* $x_{1j} \in \tilde{x}$ *and* $b_{1j} = \gamma(x_{1j})$;
- ◇ $\mathbb{F} = \{d^{\mathbb{F}}_{ij}\}_{n\times m}$ *where* $d^{\mathbb{F}}_{ij} = \begin{cases} \mathit{count}(x_{1j}, \tilde{\sigma}^{y_{i1}}) : 0 & \text{if } x_{1j} \in \tilde{x}^{y_{i1}} \\ 0 & \text{otherwise} \end{cases}$
- ◇ $Y_b = \{y_{i1}(b_{i1}) : \infty\}_{n\times 1}$ *for* $y_{i1} \in \tilde{y}$ *and* $b_{i1} = \gamma(y_{i1})$.

In the definition above the list of local binders is specified in such a way that each function (f_{y_i}), its parameters ($\tilde{x}^{y_i}$) and the list of mappings ($\tilde{\sigma}^{y_i}$) are indexed with the relevant output port (y_i). We assume γ to provide the list of basic types for the input ports, ($\gamma(\tilde{x}) = \tilde{b}$), for the output ports ($\gamma(y_1) = b'_1, \ldots, \gamma(y_k) = b'_k$), and for each function f_{y_i}. Then, we require that the return type of each f_{y_i} matches the one of y_i (i.e., b'_i); and that the types of the parameters of f_{y_i} ($\tilde{b}^{y_i}$) match the ones of input port parameters ($\gamma(\tilde{x}^{y_i})$).

The type obtained by the extraction procedure has matrices $\mathbb{L}$ and $\mathbb{R}$ equal to zero as well as the choice port. This is always the case for base components since there are no protocols in their implementation. The matrix X_b contains a row for each input ports occurring in the component interface together with the basic type returned by the function $\gamma(\cdot)$. The elements of the matrix Y_b are obtained in a similar way but they also include the maximum number of values that can be produced by each port, that for a base component is always unbounded (local binders can potentially perform computations indefinitely).

The elements of the matrix $\mathbb{F}$ are per-each-value dependencies $\mathit{count}(x_{1j}, \tilde{\sigma}^{y_{i1}}) : 0$ if x_{1j} is in the list of arguments of function $f_{y_{i1}}$, otherwise they are zeros. Note that in a per-each-value dependency $\mathit{count}(x_{1j},)$ is the number of actual values received on input port x_{1j}. The number of possible values is always zero in the case of base components.

From an operational perspective, the type extraction procedure of Definition 4.3.1 can be implemented in the same way as is Section 3.3.

Example 4.3.1. *Consider our running example from Section 4.1, in particular, component K_{rh} specified as*

$$K_{rh} = [I]\{W\}$$

- ◇ $I = x_1, x_2, x_3, x_4.x_5\rangle y_1, y_2, y_3, y_4, y_5$
- ◇ $W = y_1 = f_{cd}(x_1), y_2 = f_{uad}(x_3), y_3 = f(), y_4 = f_{iu}(x_1, x_2, x_4), y_5 = f_{ad}(x_5)$

Let us take γ such that $\gamma(x_1, x_2, x_3, x_4, x_5) = number,\ name,\ date,\ reason,\ date$ and $\gamma(y_1) = y/n$, $\gamma(y_2) = date$, $\gamma(y_3) = date$, $\gamma(y_4) = info$ and $\gamma(y_5) = date$. We know that function f_{cd} takes a number and gives an y/n in return, hence $\gamma(f_{cd}) = number \rightarrow y/n$. Similarly, we also know that function f_{uad} is typed as $\gamma(f_{uad}) = date \rightarrow date$; function f_{iu} is typed as $\gamma(f_{iu}) = number, name, reason \rightarrow info$; and function f_{ad} is typed as $f_{ad} = date \rightarrow date$. Function f does not have any parameters hence $\gamma(f) = () \rightarrow date$. The extracted matrix X_b^{rh} of input ports with their types is

$$\begin{bmatrix} x_1(number) & x_2(name) & x_3(date) & x_4(reason) & x_5(date) \end{bmatrix}$$

and the extracted matrix Y_b^{rh} of output ports with their types and boundaries is

$$\begin{bmatrix} y_1(y/n):\infty & y_2(date):\infty & y_3(date):\infty & y_4(info):\infty & y_5(date):\infty \end{bmatrix}^T$$

Assume that the component is in the initial (static) state, so the queues of lists of mappings are empty (i.e., $\tilde{\sigma}^{y_1} = \tilde{\sigma}^{y_2} = \tilde{\sigma}^{y_3} = \tilde{\sigma}^{y_4} = \tilde{\sigma}^{y_5} = \cdot$). Hence, we have that $count(x_1, \tilde{\sigma}^{y_1}) = count(x_3, \tilde{\sigma}^{y_2}) = count(x_1, \tilde{\sigma}^{y_4}) = count(x_2, \tilde{\sigma}^{y_4}) = count(x_4, \tilde{\sigma}^{y_4}) = count(x_5, \tilde{\sigma}^{y_5}) = 0$. The extracted matrix of dependencies $\mathbb{F}^{rf}$ is then

$$\mathbb{F}^{rh} = \begin{bmatrix} 0:0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0:0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0:0 & 0:0 & 0 & 0:0 & 0 \\ 0 & 0 & 0 & 0 & 0:0 \end{bmatrix}$$

and the extracted type of the component is

$$T_{rh} = X_b^{rh}.\{0\}_{5\times5}.\{0\}_{5\times5}.\mathbb{F}^{rh}.Y_b^{rh}[0]$$

.

4.4 IC type extraction for composite components

Capturing the behaviour of a composite component is quite challenging due to the presence of choices in protocols. The type extraction for the composite components can be done in five steps. Below, we explain each of them separately.

4.4.1 Step 1: Local protocol computation

The extraction procedure for composite components targets the interfacing component, which interacts with the external world via forwarders and with other internal components via the protocol.

This step of the algorithm mainly aims at identifying precisely how the interfacing component should interact with the other ones according to the global protocol. To do achieve that, we introduce the notion of local protocols LP to represent how components interact through a given global protocol.

These protocols result from the projection of a (global) protocol on the specific role of a component [5]. This step outputs the local protocol of the interfacing component.

The syntax of local protocols LP is:

$$\begin{aligned} LP ::= {} & x?(b).LP \mid y!(b).LP \mid \mu\mathbf{X}.LP \mid \mathbf{X} \mid \mathbf{end} \mid \\ & \mid x?(b)\,\&(LP_{\mathtt{inl}}, LP_{\mathtt{inr}}) \mid y!(b) \oplus (LP_{\mathtt{inl}}, LP_{\mathtt{inr}}). \end{aligned}$$

A term $x?(b).LP$ denotes a reception of a value of type b on port x, upon which the interaction continues according to the continuation LP. Similarly, a term $y!(b).LP$ represents an output on port y. Then, we have standard constructs for recursion, recursion variable and protocol termination (**end**). The term $x?(b)\,\&(LP_{\mathtt{inl}}, LP_{\mathtt{inr}})$ denotes the input of a choice on the port x of a value of type b, after which the communication continues according to $LP_{\mathtt{inl}}$ or $LP_{\mathtt{inr}}$. The term $y!(b) \oplus (LP_{\mathtt{inl}}, LP_{\mathtt{inr}})$ is similar, but for the output of a choice. To simplify the formal development, hereafter we restrict ourselves to consider only global protocols that have at most one recursion and one branching (selection). Consequently, also the projected local protocols respect these restrictions. We also assume that message labels can appear at most once in a global protocol specification (up to unfolding of recursion), so that ports occur only once in projected local protocols (also up to unfolding).

Note that we do not lose generality because we can always transform a generic protocol description by renaming the labels so as to satisfy the previous assumption.

We introduce some notation useful for the definition of the type extraction. The first one is the notion of contexts for local protocols (excluding recursion), in symbols $\mathcal{C}[\cdot]$. Intuitively, a context is a local protocol with a hole that is filled in by the term we want to highlight. Formally, a context $\mathcal{C}$ is defined as follow:

$$\mathcal{C}[\,\cdot\,] ::= x? : b.\mathcal{C}[\,\cdot\,] \mid y! : b.\mathcal{C}[\,\cdot\,] \mid \cdot\,.$$

Below we use contexts to abstract from the entire local protocol and focus on specific parts. Finally, we denote with $\mathit{fp}(LP)$ the ports appearing in a local protocol, and with $\mathit{rep}(LP)$ the ports that occur in a recursion ($\mu\mathbf{X}.LP$), formally

$$\begin{aligned}\mathit{fp}(LP) \triangleq \{z | LP = \mathcal{C}[z?.LP'] \vee LP = \mathcal{C}[z!.LP'] \vee LP = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[z?.LP']]\\ \vee LP = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[z!.LP']] \vee \mathcal{C}[\,\&(LP', LP'')] \vee \mathcal{C}[\oplus(LP', LP'')]\}\\ \mathit{rep}(LP) \triangleq \{z | LP = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[z?.LP']] \vee LP = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[z!.LP']]\}\end{aligned}$$

4.4.2 Step 2: Checking composition

Base components can always take part in a composition, as long as they are able to carry out the prescribed protocol that governs the interaction. However, when we compose composite components multiple times, we must pay some attention. Indeed, after the first composition, new issues may arise: encapsulating the new-composed components might lead to loosing track of the source of a choice. This may require that to correctly track the source of a choice we need to type check a component more than once. However, tracking all the choices may require revealing some implementation details of a component that may hinder the encapsulation properties that we want to ensure. For this reason, we decide to detect the components that suffer from this problem and reject them during the type extraction. The goal of this step of algorithm is exactly to detect and reject such ''bad" components.

Below, we first discuss the conditions that a component and a composition need to fulfil in order to be typed.

Note that extracting a component's type is challenging also with our restrictions. Moreover, the second or the third level of the composition might make the internal choice invisible to the external environment, hence, making hard describing the component's behaviour.

Let $K' = [\tilde{x}\,\rangle\,\tilde{y}]\{G; R; D; r[F]\}$ be the composite component that we are about to type. Let K be its interfacing component with the following type T

$$T = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}],$$

and let LP be the local protocol of component K (the projection of G to role associated to K).

In the following, $LP^{\cancel{ch}}$ denotes a local protocol LP that includes no branching and no selection term. Formally, $LP^{\cancel{ch}}$ is such that

$$LP \neq \mathcal{C}[z_{1j}?(b)\,\&(LP_1, LP_2)] \wedge LP \neq \mathcal{C}[y_{i1}!(b) \oplus (LP_1, LP_2)] \wedge LP \neq$$

$$\mathcal{C}[\mu\mathbf{X}.z_{1j}?(b)\,\&(LP_1, LP_2)] \wedge LP \neq \mathcal{C}[\mu\mathbf{X}.y_{i1}!(b) \oplus (LP_1, LP_2)].$$

We have 5 different cases to determine whether K' can be typed depending on the choice port `ch`:

[Case 1.] $\texttt{ch} = 0$. In this case we know that there is no branching in the implementation of the interfacing component K, hence,

$$T = X_b.\{0\}.\{0\}.\mathbb{F}.Y_b.[0].$$

We identify other 5 cases depending on the shape of the local protocol LP:

[Case 1.(a)] $LP = LP^{\cancel{ch}}$. It is enough to ensure that the component K can carry-out the protocol LP by observing the conformance relation (see below).

[Case 1.(b)] $LP = \mathcal{C}[x_{1k}\,\&(LP_1, LP_2)]$. Since K is the interfacing component, it has to expose its choice to the external environment. We require the existence of an external output port y_{i1} of K which informs us about the internal choice (whose output does not depend on any choice), and which depends on x_{1k}, formally,

$$\exists y_{i1}(b_{i1}) : \mathbf{B}_{i1} \in Y_b \mid F = y'_{i1} \leftarrow y_{i1}, F' \wedge d^{\mathbb{L}}_{ik} \neq 0 \wedge \forall m(d^{\mathbb{R}}_{im}, d^{\mathbb{L}}_{im} = 0)$$

Note: y'_{i1} becomes a choice port of K'.

[Case 1.(c)] $LP = \mathcal{C}[y_{i1} \oplus (LP_1, LP_2)]$. If the protocol requires component K to output its choice, then the output port y_{i1} must depend on some external input port. Formally,

$$\exists x_{1k}(b_{1k}) \in X_b \mid F = x_{1k} \leftarrow x'_{1k}, F' \wedge d^{\mathbb{F}}_{ik} \neq 0 \;\wedge\; \forall m(d^{\mathbb{R}}_{im}, d^{\mathbb{L}}_{im} = 0)$$

Note that x'_{ik} becomes a choice port of K'.

[Case 1.(d)] $LP = \mathcal{C}[\mu\mathbf{X}.x_{1k}\,\&(LP_1, LP_2)]$. The conditions are the same as in Case 1.(b), but in addition we require the existence of a per-each-value dependency between y_{i1} and x_{1k}, formally:

$$\exists y_{i1}(b_{i1})\!:\!\mathbf{B}_{i1}\!\in\! Y_b \mid F = y'_{i1} \leftarrow y_{i1}, F' \wedge\ d^{\mathbb{F}}_{ik}\!=\!N_{ik}\!:\!p_{ik} \wedge$$
$$\forall m(d^{\mathbb{R}}_{im}, d^{\mathbb{L}}_{im}\!=\!0)$$

Note that y'_{i1} becomes a choice port of K'.

[Case 1.(e)] $LP = \mathcal{C}[\mu\mathbf{X}.y_{i1} \oplus (LP_1, LP_2)]$. The conditions are the same as in Case 1.(c), but we also require a per-each-value dependency of y_{i1} on some external input port x_{1k}:

$$\exists x_{1k}(b_{1k}) \in X_b \mid F = x_{1k} \leftarrow x'_{1k}, F' \wedge d^{\mathbb{F}}_{ik} = N_{ik} : p_{ik} \wedge$$
$$\forall m(d^{\mathbb{R}}_{im}, d^{\mathbb{L}}_{im} = 0)$$

Note that x'_{1k} becomes a choice port of K'.

[Case 2.] `ch` is an input port x_{ik}. In this case, we know that K internally performs a choice, but the information about it is provided to K by another subcomponent of K'. Then, the only cases we consider are the ones where LP is described as in Case 1.(b) and Case 1.(d) but where we now have that `ch` $= x_{ik}$. The reasoning and the conditions are the same with the difference that now we do not create a choice port, but we "replace it" with y'_{i1}, for the new type.

[Case 3.] `ch` is an output communication port y_{i1}. The only cases we consider are the ones where LP is described as in Case 1.(c) and Case 1.(e) but where we now have that `ch` $= y_{i1}$. Thus, we "replace" choice port y_{i1} with x'_{1k}, for the new type.

The following two cases refer to the choice port of the type T (the type of the interfacing component) that forward their values together with a choice to/from the external environment.

[Case 4.] Let `ch` be an interfacing input port such that `ch` $= x_{1k}$ and $F = x_{1k} \leftarrow x'_{1k}, F'$. We have three different cases depending on the local protocol LP:

[Case 4.(a)] $LP = LP^{\cancel{ch}}$. We check the compatibility using the conformance relation. For the new composite component, the choice port is x'_{1k}.

[Case 4.(b)] $LP = \mathcal{C}[x_{1j} \,\&(LP_1, LP_2)]$ or $LP = \mathcal{C}[\mu\mathbf{X}.x_{1k} \,\&(LP_1, LP_2)]$. We do not compose K, since the component has to choose between the choice that comes internally and externally.

[Case 4.(c)] $LP = \mathcal{C}[y_{i1} \oplus (LP_1, LP_2)]$. It must be that y_{i1} depends on a choice port x_{1k}. Moreover we require that there are no ports on which y_{i1} depends that in turn depend on the internal choice of K. Formally,

$$d^{\mathbb{F}}_{ik} \neq 0 \;\wedge\; \forall m(d^{\mathbb{R}}_{im} = 0 \wedge d^{\mathbb{L}}_{im} = 0)$$

[Case 4.(d)] $LP = \mathcal{C}[\mu\mathbf{X}.y_{i1} \oplus (LP_1, LP_2)]$. If the protocol requires component to output its choice, then the output port must depend on some external input port. So, also require a per-each-value dependency of y_{i1} on external input port x_{1k}:

$$d^{\mathbb{F}}_{ik} = N_{ik} : p_{ik} \;\wedge\; \forall m(d^{\mathbb{R}}_{im} = 0 \wedge d^{\mathbb{L}}_{im} = 0)$$

[Case 5.] Let `ch` be an interfacing output port such that $\mathtt{ch} = y_{i1}$ and $F = y'_{i1} \leftarrow y_{i1}$. We have the following four cases:

[Case 5.(a)] $LP = \mathcal{C}[y_{i1} \oplus (LP_1, LP_2)]$ or $LP = \mathcal{C}[\mu.\mathbf{X}y_{i1} \oplus (LP_1, LP_2)]$. In this case we do not compose the component further, since it might happen that it has to chose between choice made internally and externally.

[Case 5.(b)] $LP = LP^{\cancel{\mathtt{ch}}}$, then it must exist an input port x_{1k} such that

$$d^{\mathbb{L}}_{ik} \neq 0 \;\wedge\; \forall m(d^{\mathbb{R}}_{im} = 0 \wedge d^{\mathbb{L}}_{im} = 0)$$

[Case 5.(c)] $LP = \mathcal{C}[x_{1k} \,\&(LP_1, LP_2)]$. In this case we have that y_{i1} depends on x_{1k}, and must not depend on any port whose values are received depending on the choice the component K makes. Formally,

$$d^{\mathbb{L}}_{ik} \neq 0 \;\wedge\; \forall m(d^{\mathbb{R}}_{im} = 0 \wedge d^{\mathbb{L}}_{im} = 0)$$

[Case 5.(d)] $LP \neq \mathcal{C}[\mu\mathbf{X}.x_{1k} \,\&(LP_1, LP_2)]$. In this case, y_{i1} has a per-each-value dependency on x_{1k}, and must not depend on any port whose values are received depending on the choice the component K makes. Formally,

$$d^{\mathbb{L}}_{ik} = N_{ik} : p_{ik} \;\wedge\; \forall m(d^{\mathbb{R}}_{im} = 0 \wedge d^{\mathbb{L}}_{im} = 0)$$

The choice port of the composite component is y'_{i1}.

Example 4.4.1. *Consider component*

$$K_{pr} = [x_{11}, x_{12}, x_{13}, x_{14} \rangle\, y_{11}, y_{21}]\{G; R; ; r[F]\}$$

$R = RH = K_{rh}, ADM = K_a$

$D = ADM.z_1 \xleftarrow{a/d} RH.y_1, ADM.z_2 \xleftarrow{date} RH.y_2, ADM.z_3 \xleftarrow{Av.d} RH.y_3,$
$RH.x_5 \xleftarrow{App.d} ADM.w_1$

$F = x_1 \leftarrow x_{11}$*,* $x_2 \leftarrow x_{12}$*,* $x_3 \leftarrow x_{13}$*,* $x_4 \leftarrow x_{14}$*,* $y_{11} \leftarrow y_4$ *and* $y_{21} \leftarrow y_5$

introduced in Section 4.1 and its type

$$T_{pr} = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}]$$

where

1. $X_b = \begin{bmatrix} x_{11}(number) & x_{12}(name) & x_{13}(date) & x_{14}(reason) \end{bmatrix}$
2. $\mathbb{L} = \{0\}_{2x4}$
3. $\mathbb{R} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \boxed{\Omega:0} & 0 & \boxed{\Omega:0} & 0 \end{bmatrix}$
4. $\mathbb{F} = \begin{bmatrix} \boxed{0:0} & \boxed{0:0} & 0 & \boxed{0:0} \\ 0 & 0 & 0 & 0 \end{bmatrix}$
5. $Y_b = \begin{bmatrix} y_{11}(info):\infty \\ y_{21}(date):1 \end{bmatrix}$
6. $\begin{bmatrix}\mathtt{ch}\end{bmatrix} = \begin{bmatrix}x_{11}\end{bmatrix}$

Since it uses x_{11} *as a choice port, it cannot be in a composition governed by a protocol* $G = K \xrightarrow{name} PR(G_1, G_2)$*, because* K_{pr} *would have to choose between an internal and an external choice.*

4.4.3 Step 3: Checking the conformance with the protocol

This step checks whether the interfacing component is able to fulfill its task, namely it is *compliant* with the protocol. Intuitively, a component is compliant when it can perform the actions specified by its local protocol. Below, we formalise this fact by introducing the notion of *conformance relation*. In practice, given a component and a local protocol this step of the algorithm verifies that the conformance relation holds.

However, there is the following technical issue we need to address. The interfacing component, beside interacting with the others, also interacts with the external environment; and, given the reactive nature of our components, the interfacing component can receive in any moment values that are external input. To take into account this ability, we modify our types as if the external values are already received unlimited number of times. So, to check the conformance between the interfacing component and the protocol, we observe its modified type and its local protocol instead of the actual type. Below, we first introduce the notion of modified types, and then we formalise when a type is complaint to a local protocol.

Modified types

We modify the type of a (interfacing) component to receive inputs from the external environment without any constraints. The modified version of type T for a list of forwarders F is denoted by $\mathcal{T}(F, T)$. In the modified type $\mathcal{T}(F, T)$, each dependency on the external input ports, if any, is a per-each-value dependency, and the number of values available is unbounded. Also, we assume that a value is received whenever available on the external input ports. Formally,

Definition 4.4.1. *Let $[\tilde{x} > \tilde{y}]\{G; r = K, R; D; r[F]\}$ be a composite component, and let $T = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}]$ be the type of the interfacing subcomponent K, then $\mathcal{T}(F, T)$ is the T-modified type of T defined as follows for*

$\alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\}$:

$$\begin{array}{lcll}
\mathcal{T}(F, X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\texttt{ch}]) & \triangleq & X_b.\mathcal{T}(F, \mathbb{L}.\mathbb{R}.\mathbb{F}).Y_b.[\texttt{ch}] & \\
\mathcal{T}(F, \mathbb{L}.\mathbb{R}.\mathbb{F}) & \triangleq & \mathcal{T}(F, \mathbb{L}).\mathcal{T}(F, \mathbb{R}).\mathcal{T}(F, \mathbb{F}) & \\
\mathcal{T}(F, \{d_{ij}^{\alpha}\}) & \triangleq & \{\mathcal{T}(F, \{d_{ij}^{\alpha}\})\} & \\
\mathcal{T}(F, d_{ij}^{\alpha}) & \triangleq & d_{ij}^{\alpha} & \textit{if } x_{1j} \notin F^i \\
\mathcal{T}(F, N_{\alpha ij} : p_{\alpha ij}) & \triangleq & \infty : p_{\alpha ij} & \textit{if } x_{1j} \in F^i \\
\mathcal{T}(F, \Omega_{\alpha ij} : p_{\alpha ij}) & \triangleq & 0 & \textit{if } x_{1j} \in F^i
\end{array}$$

$\mathcal{T}$-Type syntax The syntax of $\mathcal{T}$-types is similar to the one presented of Section 4.2, with a difference in the number of values received, because now they can be unbounded (infinite).

Type	$\mathcal{T} \triangleq X_b.\mathcal{L}.\mathcal{R}.\mathcal{F}.Y_b[\texttt{ch}]$
Input interfaces	$X_b \triangleq \{x_{1j}(b_{1j})\}_{1\times m}$
Output interfaces	$Y_b \triangleq \{y_{i1}(b_{i1}) : \mathbf{B}_{i1}\}_{n\times 1}$
Choice-related dependencies	$\mathbb{L}.\mathbb{R} \triangleq \{d_{ij}^{\mathbb{L}}\}_{n\times m}.\{d_{ij}^{\mathbb{R}}\}_{n\times m}$
Choice-free dependencies	$\mathbb{F} = \{d_{ij}^{\mathbb{F}}\}_{n\times m}$
Dependencies	$d_{ij}^{\alpha} ::= 0 \mid \Omega : p_{\alpha ij} \mid N_{\alpha ij} : p_{\alpha ij}$
Choice port	$\texttt{ch} ::= 0 \mid x_{1j}(b_{1j}) \in X_b \mid y_{i1}(b_{i1}) : \mathbf{B}_{i1} \in Y_b$
Boundary	$\mathbf{B}_{i1}, p_{\alpha ij}, \mathcal{N}_{\alpha ij} ::= N \mid \infty$
Additional variables	$N \in \mathbb{N}_0 \quad \alpha \in \{\mathcal{L}, \mathcal{R}, \mathcal{F}\}$
	$i \in \{1, ..., n\}, j \in \{1, ..., m\}; m, n \in \mathbb{N}$

Table 16: $\mathcal{T}$-Type syntax (IC type language)

$\mathcal{T}$-Type semantics (IC type language) The rules defining the semantics of modified types are the same as the ones shown in Table 15.

$$\frac{d^{\alpha}_{ij} = 0}{d^{\alpha}_{ij} \xrightarrow{x_{1j}?} d^{\alpha}_{ij}}\ [\texttt{InpZero}] \qquad \frac{}{d^{\alpha}_{ij} \xrightarrow{x_{1k}?} d^{\alpha}_{ij}}\ [\texttt{InpDisc}] \qquad \frac{d^{\mathbb{F}}_{ij} = \Omega : p_{\mathbb{F}ij}}{\Omega : p_{\mathbb{F}ij} \xrightarrow{x_{1j}?} 0}\ [\texttt{InpIntF}]$$

$$\frac{d^{\mathbb{F}}_{ij} = \mathcal{N}_{\mathbb{F}ij} : p_{\mathbb{F}ij}}{\mathcal{N}_{\mathbb{F}ij} : p_{\mathbb{F}ij} \xrightarrow{x_{1j}?} \mathcal{N}_{\mathbb{F}ij} + 1 : p_{\mathbb{F}ij}}\ [\texttt{InpPevF}] \qquad \frac{}{T \xrightarrow{\tau} T}\ [\texttt{InpInter}]$$

$$\frac{d^{\beta}_{ij} = \Omega : p_{\beta ij} \quad \beta \in \{\mathbb{L}, \mathbb{R}\}}{\Omega : p_{\beta ij} \xrightarrow{x_{1j}?} \Omega : p_{\beta ij} + 1}\ [\texttt{InpIntLR}]$$

$$\frac{d^{\beta}_{ij} = \mathcal{N}_{\beta ij} : p_{\beta ij} \quad \beta \in \{\mathbb{L}, \mathbb{R}\}}{\mathcal{N}_{\beta ij} : p_{\beta ij} \xrightarrow{x_{1j}?} \mathcal{N}_{\beta ij} : p_{\beta ij} + 1}\ [\texttt{InpPevLR}]$$

$$\frac{\overset{\forall i \in \{1,\dots,n\},\ \forall j \in \{1,\dots,m\},\ \forall \alpha \in \{\mathbb{L},\mathbb{R},\mathbb{F}\}}{d^{\alpha}_{ij} \xrightarrow{x_{1j}?} d'^{\alpha}_{ij} \ \wedge\ x_{1j}(b_{1j}) \in X_b}}{X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\texttt{ch}] \xrightarrow{x_{1j}(b_{1j})?} X_b.\{(d^{\mathbb{L}}_{ij})'\}.\{(d^{\mathbb{R}}_{ij})'\}.\{(d^{\mathbb{F}}_{ij})'\}.Y_b[\texttt{ch}]}\ [\texttt{InpT}]$$

$$\frac{\overset{\forall j \in \{1,\dots,m\} \quad d^{\alpha}_{ij} = \mathcal{N}_{\alpha ij} : p_{\alpha ij} \vee d^{\alpha}_{ij} = 0 \quad \mathbf{B}_{i1}, \mathcal{N}_{\alpha ij} > 0}{i \in \{1,\dots,n\} \ \wedge\ y_{i1}(b_{i1}) : B_{i1} \in Y_b}}{X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\texttt{ch}] \xrightarrow{y_{i1}(b_{i1})!} X_b.shrink(\mathbb{L}, \mathbb{R}, \mathbb{F}, Y_b, i)[\texttt{ch}]}\ [\texttt{OutT}]$$

Table 17: $\mathcal{T}$-Type semantics (IC type language)

Note that given a composite component K, the types of all the other subcomponents are unmodified (formally, $K_1 \Downarrow T_{r_2}, \dots, K_n \Downarrow T_{r_n}$) so, we have that

$$\mathcal{T}(F, T_{r_2}) = T_{r_2}, \dots, \mathcal{T}(F, T_{r_n}) = T_{r_n}$$

since the only component that forwards the values from/to external environment is component K_1. Note that given a composite component, the types of all the other subcomponents are unmodified, since the only component that forwards the values from/to external environment is interfacing one.

We also introduce a subtyping relation between modified types, defined as:

Definition 4.4.2. *$\mathcal{T}' \leq \mathcal{T}$ if exists a (possibly empty) set of typed input ports $\{x_1(b_1), x_2(b_2), \ldots, x_k(b_k)\}$ such that $\mathcal{T}' \xrightarrow{x_1?(b_1)} \cdots \xrightarrow{y!(b)} \cdots \xrightarrow{x_k?(b_k)} \mathcal{T}$.*

Since, by the Definition 4.4.1 we have that dependencies on the external ports are either zero or there is an unlimited number of values received, this definition raises after observing the changes and the reduction of the modified type after receiving values that come from the protocol.

From the semantics and the definition of modified types, we can deduce that the only possible difference between types $\mathcal{T}'$ and $\mathcal{T}$ is that some initial dependencies might be dropped or that the number of values available on some input ports for some outputs might increase.

Example 4.4.2. *Consider the interfacing component*

$$K_{rh} = [I]\{W\}$$

- ◇ $I = x_1, x_2, x_3, x_4.x_5\rangle y_1, y_2, y_3, y_4, y_5$
- ◇ $W = y_1 = f_{cd}(x_1), y_2 = f_{uad}(x_3), y_3 = f(), y_4 = f_{iu}(x_1, x_2, x_4), y_5 = f_{ad}(x_5)$

of component K_{pr} of Section 4.1, the list of forwarders

$$F = x_1 \leftarrow x_{11}, x_2 \leftarrow x_{12}, x_3 \leftarrow x_{13}, x_4 \leftarrow x_{14}, y_{11} \leftarrow y_4, y_{21} \leftarrow y_5$$

and type :

$$T_{rh} = X_b^{rh}.\mathbb{L}^{rh}.\mathbb{R}^{rh}.\mathbb{F}^{rh}.Y_b^{rh}.[\mathtt{ch}^{rh}]$$

$$X_b^{rh} = \begin{bmatrix} x_1(number) & x_2(name) & x_3(date) & x_4(reason) & x_5(date) \end{bmatrix}$$

$$\mathbb{L}^{rh} = \{0\}_{5x5} \quad \mathbb{R}^{rh} = \{0\}_{5x5}$$

$$\mathbb{F}^{rh} = \begin{bmatrix} N_{\mathbb{F}11}\!:\!0 & 0 & 0 & 0 & 0 \\ 0 & 0 & N_{\mathbb{F}23}\!:\!0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ N_{\mathbb{F}41}\!:\!0 & N_{\mathbb{F}42}\!:\!0 & 0 & N_{\mathbb{F}44}\!:\!0 & 0 \\ 0 & 0 & 0 & 0 & N_{\mathbb{F}55}\!:\!0 \end{bmatrix}$$

$$Y_b^{rh} = \begin{bmatrix} y_1(y/n):\infty \\ y_2(date):\infty \\ y_3(date):\infty \\ y_4(info):\infty \\ y_5(date):\infty \end{bmatrix}$$

$$[\mathtt{ch}^{rh}] = [0]$$

Then, following the Definition 3.5.1, we have the following modified type:

$$\mathcal{T}(F, T_{rh}) = X_b^{rh}.\mathbb{L}^{rh}.\mathbb{R}^{rh}.\mathbb{F}^{rh^*}.Y_b^{rh}[\mathtt{ch}^{rh}]$$

$$\mathbb{F}^{rh^*} = \begin{bmatrix} \infty:0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \infty:0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \infty:0 & \infty:0 & 0 & \infty:0 & 0 \\ 0 & 0 & 0 & 0 & N_{\mathbb{F}_55}:0 \end{bmatrix} \check{z}$$

where all the other dependency matrices remain the same (they are zero matrices).

To formalise the ability of a component to carry-out its local protocol, we introduce a conformance relation, denoted by $\bowtie$. This relation is inductively defined by the inference rules of Table 12, where Γ is a type environment mapping recursion variables to modified types. We now briefly comment on these rules.

Notice that all the rules, except the last two, are the same as the Rules from the Subsection 3.5.1, hence, we did not rename it. Rules $[BranchConf]$ and $[SelectConf]$ are for the protocols with the choices, and are similar to the Rules $[InpConf]$ and $[OutConf]$, respecively, but require that the resulting type is conformant with both branches.

Example 4.4.3. *Let us take modified type $\mathcal{T}(F, T_{rh})$ from the previous example and the local protocol $LP = y_1!(app).y_2!(date).\mathbf{end}$.*

We apply the Rule $[OutConf]$: all the dependencies of y_1 are either 0 or have an unlimited number of values available for the computation of its value (only one dependency), and moreover, the boundary is unlimited, the type can output a value from the port y_1. The new-evolved type is the $\mathcal{T}(F, T_{rh})$ itself (since $\infty - 1 = \infty$ for both number of values available and the boundary, see the Table 17). It is conformant to the protocol $y_2!(date).\mathbf{end}$. and we can apply the same rule and the same reasoning again, and finally, apply the rule $[EndConf]$.

$$\frac{\mathcal{T} \xrightarrow{x?(b)} \mathcal{T}' \quad \Gamma \vdash \mathcal{T}' \bowtie LP}{\Gamma \vdash \mathcal{T} \bowtie x?(b).LP} \; [InpConf]$$

$$\frac{\mathcal{T} \xrightarrow{y!(b)} \mathcal{T}' \quad \Gamma \vdash \mathcal{T}' \bowtie LP}{\Gamma \vdash \mathcal{T} \bowtie y!(b).LP} \; [OutConf]$$

$$\frac{}{\Gamma \vdash \mathcal{T} \bowtie \mathbf{end}} \; [EndConf] \qquad \frac{\mathcal{T}' \leq \mathcal{T}}{\Gamma, X : \mathcal{T}' \vdash \mathcal{T} \bowtie X} \; [VarConf]$$

$$\frac{\Gamma, X : \mathcal{T} \vdash \mathcal{T} \bowtie LP}{\Gamma \vdash \mathcal{T} \bowtie \mu\mathbf{X}.LP} \; [RecConf]$$

$$\frac{\mathcal{T} \xrightarrow{x?(b)} \mathcal{T}' \quad \Gamma \vdash \mathcal{T}' \bowtie LP_1 \quad \Gamma \vdash \mathcal{T}' \bowtie LP_2}{\Gamma \vdash \mathcal{T} \bowtie x?(b)\,\&(LP_1, LP_2)} \; [BranchConf]$$

$$\frac{\mathcal{T} \xrightarrow{y!(b)} \mathcal{T}' \quad \Gamma \vdash \mathcal{T}' \bowtie LP_1 \quad \Gamma \vdash \mathcal{T}' \bowtie LP_2}{\Gamma \vdash \mathcal{T} \bowtie y!(b) \oplus (LP_1, LP_2)} \; [SelectConf]$$

Table 18: Conformance relation (IC type language)

4.4.4 Step 4: Dependencies extraction

The goal of this step is to determine the dependencies between the input and output ports of a composite components. We identify two kinds of dependencies of output ports on the input ports: the *direct* and the *transitive* ones. Intuitively, we say that an output port y directly depends on an input port x, when x is one of the argument of the function attached to y. Whereas we say that an output port y transitively depends on an input port x when y depends on the values produced by another internal component that in turn depends on x.

Computing such dependencies is the most challenging part of our type extraction procedure. For this task we rely on the following observation. We know that the values received on the composite component ports are directly forwarded to the ones of interfacing component. Symmetrically, the values produced by the output ports of a composite component come from its interfacing component. Thus, to extract the dependencies we can focus on the local protocol of the interfacing component, its ports and the forwarders attached to them.

Below, we provide a formal characterization of the dependencies from which it is direct derive an algorithm to actually compute them. In our formal treatment we denote with external input ports (F^i) and with external output ports (F^o) the ports of the interfacing components which interact with the external environment. We define them as follows:

$$F^i \triangleq \{x \mid \exists x' \quad F = x \leftarrow x', F'\}$$
$$F^o \triangleq \{y \mid \exists y' \quad F = y' \leftarrow y, F'\}$$

We can identify two kinds of dependencies of output ports on the input ports: the *direct* and the *transitive* ones. Below, we characterise both of them and to graphical representation we reuse the figures from from Subsection 3.4.1, where we put (*) next to the name of reused figure.

Direct dependencies

Given a composite component K with interfacing component K_p, intuitively, a direct dependency exists between an external input port x and an external output y when both the following conditions are met (see Figure 3): (i) y is linked to an output port y_p of K_p which depends solely on input ports of K_p; (ii) port x is linked to an input port x_p of K_p which y_p depends on.

Below, we formalise this intuition. Let K be a composite component with interfacing component K_p and forwarders F; and, let $T_p = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\mathtt{ch}]$ be the type of K_p. Let F^i be the set of (internal) input ports of F and F^o be the set of (internal) output ports of F (e.g., if $F = x_p \leftarrow x$ then $F^i = \{x_p\}$).

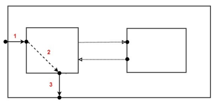

Figure 8: Direct Dependency (*)

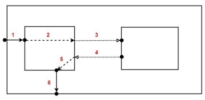

Figure 9: Transitive Dependency (*)

For each output port y_{i1}, we gather the set of direct dependencies $\mathbf{D}(\mathbb{L}, \mathbb{R}, \mathbb{F}, y_{i1})$ defined as follows:

$$\mathbf{D}(\mathbb{L}, \mathbb{R}, \mathbb{F}, y_{i1}) \triangleq \{d^\alpha_{ij}, \alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\} \mid d^\alpha_{ij} \neq 0 \wedge x_{1j} \in F^i \wedge y_{i1} \in F^o\}$$

Example 4.4.4. *All the non-zero dependencies of base components are the direct dependencies, since none of them is created through the protocol.*

The transitive dependencies

Intuitively, given a composite component K we have a transitive dependency between an external input port x and an external output y, when y is linked to an output port y_p of the interfacing component K_p, which depends solely on a value output by another internal component, that in turn depends on x (see Figure 9).

More precisely, for a transitive dependency to exist there are three necessary conditions. The first one is to have in the local protocol of K_p at least one output action, say on port y', that precedes at least one input action, say on port x'. The second condition is that such output port y' depends on some external input port x of K. The third condition is that there exists some external output port y of K that depends on port x'. In such cases, we say that y depends on x in a transitive way.

We first introduce a relation $\diamond^{\cdot}_{\cdot}$ to capture the first condition above. Let LP be the local protocol of K_p, two ports x and y are in relation $\diamond^{LP}_{r}$ for LP and $r \in \{1,2,3,4,5,6\}$ if $x, y \in \mathit{fp}(LP)$ and one of the following condition hods:

1. $y \diamond^{LP}_{1} x$ if $LP = \mathcal{C}[y!(b).\mathcal{C}'[x?(b).LP']]$ and $x, y \notin \mathit{rep}(LP)$;
2. $y \diamond^{LP}_{2} x$ if $LP = \mathcal{C}[y'!(b).\mathcal{C}'[\mu\mathbf{X}.\mathcal{C}''[x?(b').LP]]]$ and $y \notin \mathit{rep}(LP)$;
3. $y \diamond^{LP}_{3} x$ if $LP = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[y!(b).\mathcal{C}''[x?(b).LP']]]$.
4. $y \diamond^{LP}_{4} x$ if $LP = \mathcal{C}[y!(b).\mathcal{C}'[(\mathcal{C}''[x?(b').LP']), LP'']]$ and $x, y \notin \mathit{rep}(LP)$;
5. $y \diamond^{LP}_{5} x$ if $LP = \mathcal{C}[y!(b).\mathcal{C}'[(\mathcal{C}''[\mu\mathbf{X}.\mathcal{C}''[x?(b').LP']]), LP'']]$ and $y \notin \mathit{rep}(LP)$;
6. $y \diamond^{LP}_{6} x$ if $LP = \mathcal{C}[\mu\mathbf{X}.\mathcal{C}'[y!(b).\mathcal{C}''[(\mathcal{C}'''[x?(b').LP']), LP'']]]$.

First note that in all conditions we find an output on y before the reception on x. The first relation captures the case where both x and y are not in the recursive (if exists) part of the protocol; The second one is for the case where y is not in the repetitive part of the protocol, but x is; The next one serves for describing the relation where both of them are in the body of the recursion of the protocol; The fourth and the fifth one are the same as one and two, respectively, but they consider the protocol with the choices, where y comes before, and x after the choice in the protocol description; The last relation takes into account the case where both reception on x and the output on y are found in the recursive part of the protocol, after the choice is made.

We now formally characterise the transitive dependencies. Let $[\tilde{x}' \rangle \tilde{y}']\{G; r = K_p, R; D; r[F]\}$ be a composite component, $T_p = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\mathtt{ch}]$ be the type of the interfacing component K_p and LP its local protocol.

Given an output port $y_{i1} \in Y_b$ of K, the set of its transitive dependencies denoted by $\mathbf{D}(\mathbb{R}, \mathbb{L}, \mathbb{F}, F, y_{i1}, LP)$ relies on the predicate η^{α_1/α_2}, defined as:

$$\eta^{\alpha_1/\alpha_2}(LP, F) \triangleq d^{\alpha_1}_{kj} \neq 0 \wedge d^{\alpha_2}_{im} \neq 0 \wedge x_{1j} \in F^i \wedge y_{i1} \in F^o \wedge y_{k1} \diamond^{LP}_r x_{1m}$$

and for abbreviating the following definitions we have that:

If $LP = \mathcal{C}[\,\&(LP', LP'')] \vee \mathcal{C}[\oplus(LP', LP'')]$ then $x \in \mathbb{L}$ if $x \in \mathit{fp}(LP')$ and $x \in \mathbb{R}$ if $x \in \mathit{fp}(LP'')$, otherwise $x \in \mathbb{F}$.

The set of transitive dependencies $\mathbf{D}(\mathbb{R}, \mathbb{L}, \mathbb{F}, F, y_{i1}, LP)$ is defined below as the union of six sets:

$$\mathbf{D}(\mathbb{R}, \mathbb{L}, \mathbb{F}, F, y_{i1}, LP) \triangleq A(\alpha, F, y_{i1}, LP) \cup B(\alpha, F, y_{i1}, LP) \cup C(\alpha, F, y_{i1}, LP) \cup$$
$$D(\alpha, F, y_{i1}, LP) \cup E(\alpha, F, y_{i1}, LP) \cup Q(\alpha, F, y_{i1}, LP)$$

where the sets A and B contain the initial dependencies, while C contains the per-each-value dependencies obtained transitively, formally:

$$A(\alpha, F, y_{i1}, LP) = \big\{\Omega : p_{\mathbb{F}ij} \;\mid\; \eta^{\mathbb{F}/\mathbb{F}}(F, LP) \wedge (d^{\mathbb{F}}_{im} = \Omega : 0 \vee d^{F}_{im} = 0 : 0) \wedge$$
$$d^{\mathbb{R}}_{kj}, d^{\mathbb{R}}_{i}m, d^{\mathbb{L}}_{kj}, d^{\mathbb{L}}_{im} = 0 \wedge r \in \{1, 2\}\big\}$$

$$B(\alpha, F, y_{i1}, LP) = \big\{\Omega : p_{\mathbb{F}ij} \;\mid\; \eta^{\mathbb{F}/\mathbb{F}}(F, LP) \wedge r = 3 \wedge ((d^{\mathbb{F}}_{im} = \Omega : 0) \vee$$
$$(d^{F}_{kj} = \Omega : 0 \wedge d^{\mathbb{F}}_{im} = 0 : 0 \wedge \mathit{vf}(LP, x_{1m}, y_{k1} = 0))) \wedge d^{\mathbb{R}}_{kj}, d^{\mathbb{R}}_{i}m, d^{\mathbb{L}}_{kj}, d^{\mathbb{L}}_{im} = 0\}$$

$$C(\alpha, F, y_{i1}, LP) = \big\{[N_{\mathbb{F},kj} + N_{\mathbb{F}im} + \mathit{vf}(LP, x_{1m}, y_{k1})] : 0)) \;\mid\; \eta^{\mathbb{F}/\mathbb{F}}(F, LP)$$
$$\wedge r = 3 \wedge d^{\mathbb{F}}_{im} = N_{\mathbb{F}im} : 0 \wedge d^{\mathbb{F}}_{kj} = N_{\mathbb{F}_kj} : 0 \wedge d^{\mathbb{R}}_{kj}, d^{\mathbb{R}}_{i}m, d^{\mathbb{L}}_{kj}, d^{\mathbb{L}}_{im} = 0\big\}$$

$$D(\alpha, F, y_{i1}, LP) = \big\{\Omega : p_{\beta ij}, \beta \in \{\mathbb{L}, \mathbb{R}\} \;\mid\; \eta^{\mathbb{F}/\mathbb{F}}(F, LP) \wedge x_{ij} \in \beta \wedge$$
$$(d^{\mathbb{F}}_{im} = \Omega : 0 \vee d^{F}_{im} = 0 : 0) \wedge d^{\mathbb{R}}_{kj}, d^{\mathbb{R}}_{i}m, d^{\mathbb{L}}_{kj}, d^{\mathbb{L}}_{im} = 0 \wedge r \in \{4, 5\}\big\}$$

$$E(\alpha, F, y_{i1}, LP) = \big\{\Omega : p_{\beta ij}, \beta \in \{\mathbb{L}, \mathbb{R}\} \mid \eta^{\mathbb{F}/\mathbb{F}}(F, LP) \wedge r = 6 \wedge x_{ij} \in \beta \\ ((d^{\mathbb{F}}_{im} = \Omega : 0) \vee (d^{F}_{kj} = \Omega : 0 \wedge d^{\mathbb{F}}_{im} = 0 : 0 \wedge \mathit{vf}(LP, x_{1m}, y_{k1} = 0))) \wedge \\ d^{\mathbb{R}}_{kj}, d^{\mathbb{R}}_{i}m, d^{\mathbb{L}}_{kj}, d^{\mathbb{L}}_{im} = 0\}$$

$$Q(\alpha, F, y_{i1}, LP) = \big\{N_{\beta im} : [N_{\beta,kj} + \mathit{vf}(LP, x_{1m}, y_{k1})], \beta \in \{\mathbb{L}, \mathbb{R}\} \mid \\ x_{ij} \in \beta\eta^{\mathbb{F}/\mathbb{F}}(F, LP) \wedge i = 6 \wedge d^{\mathbb{F}}_{im} = N_{\mathbb{F}im} : 0 \wedge d^{\mathbb{F}}_{kj} = N_{\mathbb{F}_kj} : 0 \wedge \\ d^{\mathbb{R}}_{kj}, d^{\mathbb{R}}_{i}m, d^{\mathbb{L}}_{kj}, d^{\mathbb{L}}_{im} = 0\}$$

Example 4.4.5. *As an example of extraction of transitive dependencies consider the process described in Table 13 of Section 4.1.*

4.4.5 Step 5: Boundaries extraction

Sometimes the availability of input values needed for computing some output might be limited due to the protocol. For example, if y has a dependency on some input port obtained transitively, and we have a "one-shot" protocol that allows only one value for computing y, the values produced by y are limited, where the boundary turns to be 1.

This step aims to compute these boundaries. Intuitively, the boundary associated to an output port depends on the kind of dependency. In the case of per-each-value dependency the boundary is computed as the minimum of internal boundaries. In case of initial dependency the boundary is zero. Below, we formally characterise this computation. Again, it is direct to translate our formal characterisation into an algorithm.

Let $K = [\tilde{x}\rangle\, \tilde{y}]\{G; R; D; r[F]\}$ be a composite component, and let $T_r = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}]$ be the type of the interfacing component $r = K_r$, and LP its local protocol ($LP \downarrow_r$). If $y_{i1}(b) : \mathbf{B}$ is an entry of the matrix Y_b and $y_{i1} \in F^o$, we distinguish three cases for three possible limitations:

$$\begin{aligned}
\mathbf{B}_1(\alpha, F, LP, y_{i1}) &\triangleq \{N_{\alpha ij} \mid d^{\alpha}_{ij} = N_{\alpha ij} : p_{\alpha ij} \wedge x_{1j} \notin (\mathit{fp}(LP) \bigcup F^i)\} \\
\mathbf{B}_2(\alpha, F, LP, y_{i1}) &\triangleq \{0 \mid d^{\alpha}_{ij} = \Omega_{\alpha ij} : p_{\alpha ij} \wedge x_{1j} \notin (\mathit{fp}(LP) \bigcup F^i)\} \\
\mathbf{B}_3(\alpha, F, LP, y_{i1}) &\triangleq \{N_{\alpha ij} + 1 \mid d^{\alpha}_{ij} = N_{\alpha ij} : p_{\alpha ij} \wedge x_{1j} \in \mathit{fp}(LP) \\
&\qquad x_{1j} \notin (\mathit{rep}(LP) \cup F^i)\}
\end{aligned}$$

In $\mathbf{B}_1$ and $\mathbf{B}_2$ we capture the case when there is a dependency on a port that is not used in the protocol ($x_{1j} \notin \mathit{fp}(LP)$) nor linked externally ($x_{1j} \notin F^i$), where the difference is in the kind of dependency. For per-each-value dependencies (if any), the minimum of the internally available values is identified as the potential boundary, while for initial dependencies (if present) the potential boundary is zero (or the empty set). In B_3 we capture a case where we have a per-each-value dependency on some port, and where the port is used in the protocol but in a non-repetitive way, hence only one (further) value can be provided together with internally available values.

Next, we get the minimum number among the internal boundaries for each matrix separately:

$$\mathbf{B}^{\mathbb{L}} = min\{t|t \in (\mathbf{B}_1(\mathbb{L}, F, LP, y_{i1}) \cup \mathbf{B}_2(\mathbb{L}, F, LP, y_{i1}) \cup \mathbf{B}_3(\mathbb{L}, F, LP, y_{i1}))\}$$

$$\mathbf{B}^{\mathbb{R}} = min\{t|t \in (\mathbf{B}_1(\mathbb{R}, F, LP, y_{i1}) \cup \mathbf{B}_2(\mathbb{R}, F, LP, y_{i1}) \cup \mathbf{B}_3(\mathbb{R}, F, LP, y_{i1}))\}$$

$$\mathbf{B}^{\mathbb{F}} = min\{t|t \in (\mathbf{B}_1(\mathbb{F}, F, LP, y_{i1}) \cup \mathbf{B}_2(\mathbb{F}, F, LP, y_{i1}) \cup \mathbf{B}_3(\mathbb{F}, F, LP, y_{i1}))\}$$

We compute the greater number between $\mathbf{B}^{\mathbb{R}}$ and $\mathbf{B}^{\mathbb{L}}$, since those numbers are computed from the choice matrices, and we do not want the case where the component can produce more values due to its choice, than the port is available (if choosing the left choice we get 0, and the right 5, we want the output port to be able to produce 5 values if the component makes the right choice).

$$\mathbf{B}^{*} = max\{\mathbf{B}^{\mathbb{L}}, \mathbf{B}^{\mathbb{R}}\}$$

The final boundary determined for y_{i1}, denoted by $\mathbf{B}(\alpha, F, LP, y_{i1})$, is the minimum number among the internal boundary of y_{i1} (i.e., $\mathbf{B}$) and $\mathbf{B}*$ and $\mathbf{B}^{\mathbb{F}}$ described above:

$$\mathbf{B}(\alpha, F, LP, y_{i1}) = min\{\mathbf{B}^{*}, \mathbf{B}, \mathbf{B}^{\mathbb{F}}\}$$

Example 4.4.6. *Recall the component K_{pr} of Section 4.1. Let us focus on extracting the boundary of port y_{21}. The values are forwarded from the port y_5 of the interfacing component K_{rh}. From type T_{rh} we know that the internal of y_5 is unbounded (∞). Moreover, port y_5 has only a per-each-value dependency on x_5 in matrix $\mathbb{F}$ ($N_{\mathbb{F}55}:0$). The interfacing component is governed by the protocol*

$$G = RH \xrightarrow{a/d} ADM(RH \xrightarrow{date} ADM; RH \xrightarrow{Av.d.} ADM; ADM \xrightarrow{App.d.} RH, \mathbf{end}),$$

that is a "one-shot" protocol. So, with only one dependency, we can generate the set: $\boldsymbol{B}_3 = \{N_{\mathbb{F}55} + 1 : 0\} = \boldsymbol{B}^{\mathbb{F}}$.

Thus, the boundary of port y_{21} is $\boldsymbol{B} = min\{\infty, N_{\mathbb{F}55}+1\} = N_{\mathbb{F}55}+1$. Since we are extracting the type at static time, at the beginning of the type extraction $N_{\mathbb{F}55} = 0$, so the boundary for y_{21} is 1.

4.4.6 Type extraction

We now present our type extraction procedure for a composite components. Our definition relies on a renaming operation **ren**(,) that allows us to single out the ports that are linked via forwarders to the external environment, since we introduced the dependencies and the boundaries via the names of the interfacing component ports. Formally, the renaming operation **ren**(,) is defined as follows:

$$\mathbf{ren}(F, \{x_{1j}(b_{1j})\}_{1\times n}) \triangleq \{x'_{1j}(b_{1j})\}_{1\times n}$$

$$\text{with } F = x_{11} \leftarrow x'_{11}, \ldots, x_{1n} \leftarrow x'_{1n}, F'$$

$$\mathbf{ren}(F, \{y_{i1}(b_{i1})\!:\!B_{i1}\}_{m\times 1}) \triangleq \{y'_{i1}(b_{i1})\!:\!B_{i1}\}_{m\times 1}$$

$$\text{with } F = y'_{11} \leftarrow y_{11}, \ldots, y'_{m1} \leftarrow y_{m1}, F'$$

Now we have all the ingredients for defining the type extraction for composite components:

Definition 4.4.3 (Type Extraction for a Composite Component)**.** *Let* $[\tilde{x}\rangle\,\tilde{y}]\{G; r = K, R; D; r[F]\}$ *be a composite component and* $LP = G \downharpoonright_r$ *the local protocol for interfacing component* K*. If* $T_r = X'_b.\mathbb{L}'.\mathbb{R}'.\mathbb{F}'.Y'_b.[\mathtt{ch}']$ *is the type of component* K*, then the extracted type of the whole component is*

$$T(LP, T_r, F) = \boldsymbol{ren}(F, X_b).\mathbb{L}.\mathbb{R}.\mathbb{F}.\boldsymbol{ren}(F, Y_b).[\mathtt{ch}])$$

where

$$\begin{aligned}
X_b =& \{x_{1j}(b_{1j})\}_{1\times m} \;\; \textit{where} \;\; x_{1j}(b_{1j}) \in X'_b \wedge x_{1j} \in F^i \\
\alpha =& \begin{cases} \{d^{\alpha}_{ij}\}_{n\times m} & \textit{if } d^{\alpha}_{ij} \in \boldsymbol{D}(T_r, F, LP) \\ 0 & \textit{otherwise} \end{cases} \quad \textit{where} \;\; \alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\} \\
Y_b =& \{y_{i1}(b_{i1})\!:\!\boldsymbol{B}(\alpha, F, LP, y_{i1})\}_{n\times m} \textit{where } y_{i1} \in F^o
\end{aligned}$$

From Definition 4.4.3 is evident that it suffices knowing the type and the local protocol of the interfacing component and the list of forwarders to carry out the type extraction.

Example 4.4.7. *We present a technical example which illustrates how our type extraction procedure works in the case of composite components. Consider component* K *specified as* $[\tilde{x}\rangle\,\tilde{y}]\{G; R; D; r[F]\}$*, where*

$$G = r \xrightarrow{\ell_1} p(p \xrightarrow{\ell_2} r.\mathbf{end}, \mathbf{end}), R = r = K_1, p = K_2$$

$$D = p.x_1' \xleftarrow{\ell_1} r.y_1, r.x_2 \xleftarrow{\ell_2} p.y_1', F = x_1 \leftarrow x, y \leftarrow y_2.$$

Let the interfacing component K_1 have type

$$T_1 = \begin{bmatrix} x_1(b_1) & x_2(b_2) \end{bmatrix}.\{0\}.\{0\}.\begin{bmatrix} 0:0 & 0 \\ 0 & 0:0 \end{bmatrix}.\begin{bmatrix} y_1(b^1):\infty \\ y_2(b^2):\infty \end{bmatrix}.\begin{bmatrix} 0 \end{bmatrix}$$

and component K_2 type

$$T_2 = \begin{bmatrix} x_1'(b_1') \end{bmatrix}.\{0\}.\{0\}.\begin{bmatrix} 0:0 \end{bmatrix}.\begin{bmatrix} y_1'(b_1'):\infty \end{bmatrix}.\begin{bmatrix} 0 \end{bmatrix}$$

Step 1. Local protocol. *The local protocol of the interfacing component K_1 is*

$$LP = G \downarrow_r = y!(b^1) \oplus (x_2?(b_2).\mathbf{end}, \mathbf{end})$$

Step 2. Restriction on the composition *This is the case $1.(c)$ ($LP = \mathcal{C}[y_{i1} \oplus (LP_1, LP_2)]$) that says: If the protocol requires component K_1 to output its choice, then the output port must depend on some external input port, and cannot have the dependencies in the choice matrices. From type T_1 we can read that this condition is fulfilled, since $F = x_1 \leftarrow x, F'$. The choice port of the composite component becomes x.*
Step 3. Modified type and conformance relation. *The modified type of the type T_1 is*

$$\mathcal{T}(F, T_1) = \begin{bmatrix} x_1(b_1) & x_2(b_2) \end{bmatrix}.\{0\}.\{0\}.\begin{bmatrix} \infty:0 & 0 \\ 0 & 0:0 \end{bmatrix}.\begin{bmatrix} y_1(b^1):\infty \\ y_2(b^2):\infty \end{bmatrix}.\begin{bmatrix} 0 \end{bmatrix}$$

Applying the rules [$SelectConf$] and then [$InpConf$] and/or [$EndConf$], we have that $\mathcal{T}(F, T_1) \bowtie LP$. We take the next steps observing only interfacing output port y_2.
Step 4. The dependencies extraction. *We have the following information: $y_2 \in F^o$ and $x_1 \in F^i$ ($F = x_1 \leftarrow x, y \leftarrow y_2$). From the fourth matrix of the type T_1 we have that y_1 has a per-each-value dependency on x_1, and y_2 has a per-each-value dependency on x_2, where by the local protocol description (LP) we get that $y_1, x_2 \in fp(LP)$. Moreover, $y_1 \diamond_4^{LP} x_2$, and we conclude the transitive relation between ports x_1 and y_2. The transitive relation that we obtain is initial dependency of y_2 on x_1 in the left choice matrix ($\Omega : 0$) that comes from the set D from transitive dependencies extraction where*

$$D(\alpha, F, y_{i1}, LP) = \big\{\Omega : p_{\beta ij}, \beta \in \{\mathbb{L}, \mathbb{R}\} \mid \eta^{\mathbb{F}/\mathbb{F}}(F, LP) \wedge x_{ij} \in \beta \wedge$$
$$(d_{im}^{\mathbb{F}} = \Omega : 0 \vee d_{im}^{F} = 0 : 0) \wedge d_{kj}^{\mathbb{R}}, d_i^{\mathbb{R}}m, d_{kj}^{\mathbb{L}}, d_{im}^{\mathbb{L}} = 0 \wedge r \in \{4, 5\}\big\}$$

Step 5. Boundaries extraction. *In our example, we have that the boundary of y_2 reading the T_1 is ∞, and we have that the conditions from the set $\boldsymbol{B}_3$ defined as*

$$\boldsymbol{B}_3(\mathbb{F}, F, LP, y_{i1}) \triangleq$$

$$\{N_{\mathbb{F}ij}+1 \mid d^{\mathbb{F}}_{ij} = N_{\mathbb{F}ij} : p_{\mathbb{F}ij} \wedge x_{1j} \in fp(LP) x_{1j} \notin (rep(LP) \cup F^i)\}$$

fulfilled, where in our case if we have that $x_2 = x_1 j$ and $y_2 = y_{i1}$ the boundary $\boldsymbol{B}_3$ is $0+1=1$, and the final boundary of y_2 is $min\{\infty, 1\} = 1$

*Now we rename our ports: **ren**$(F, x_1) = x$ and **ren**$(F, y_2) = y$.*

The extracted type of our component is:

$$T = \left[x(b_1)\right] . \left[\Omega : 0\right] . \{0\}.\{0\}. \left[y(b^2) : 1\right] . \left[x\right]$$

We can now formally define when a component K has type T, in which case we say K is well-typed.

Definition 4.4.4. *Let K be a component, we say that K is well typed and has a type T, in symbols $K \Downarrow T$:*

1. *If K is a base component, and T is obtained by Definition 4.3.1;*

2. *If $K = [\tilde{x} > \tilde{y}]\{G; r_1 = K_1, \ldots, r_k = K_k; D; r_1[F]\}$ is a composite component, and the following hold*

   - *Each sub-component K_i has a type T_{r_i}, i.e., $K_i \Downarrow T_{r_i}$, for $i = 1, \ldots, k$;*
   - *T is extracted from the type T_1 and $G \downharpoonright_{r_1}$ by Definition 4.4.3;*
   - *Each sub-component is conformance to its local protocol, i.e.,*

$$\mathcal{T}(F, T_{r_i}) \bowtie G \downharpoonright_{r_i} \text{ for } i = 1, 2, \ldots, k;$$

Notice that the definition above relies on modified types for ensuring the conformance to the local protocols of sub-components. However, for each type T not associated with the interfacing component we have that $\mathcal{T}(F, T) = T$ since there are no links to external ports (assuming that all ports have different identifiers).

Example 4.4.8. *One example of well typed component is component K_{pr} from Section 4.1.*

4.5 Type safety (IC type language)

In this section we state Subject Reduction and Type fidelity theorems that ensure a tight correspondence between the behaviours of components and of their extracted types. These results provide a tight correspondence between the behaviours of well-typed components and their types. This enables us to use types, e.g., for verifying properties of a component. In the statements below we denote with $\lambda(v)$ the actions $x?(v)$, $y!(v)$ or τ on values; whereas we denote with $\lambda(b)$ the actions $x?(b)$, $y!(b)$ or τ on types.

Theorem 4.5.1 (Subject Reduction). *If $K \Downarrow T$ and $K \xrightarrow{\lambda(v)} K'$ and v has type b then $T \xrightarrow{\lambda(b)} T'$ and $K' \Downarrow T''$ where $T'' \subseteq T'$.*

For stating our Type fidelity theorem, we introduce the following abbreviation which denotes the transition of a component whose input or output action is preceded and succeeded by some number (possible zero) of internal actions.

Definition 4.5.1. $K \xRightarrow{\lambda(v)} K' \triangleq K \rightarrow \overset{\tau}{\cdots} K'' \xrightarrow{\lambda(v)} K''' \rightarrow \overset{\tau}{\cdots} K'$

Now we are ready to state and prove our Type fidelity theorem:

Theorem 4.5.2 (Type fidelity). *If $K \Downarrow T$ and $T \xrightarrow{\lambda(b)} T'$ and $\lambda(b) \neq \tau$ then b is the type of a value v and $K \xRightarrow{\lambda(v)} K'$ and $K' \Downarrow T''$ where $T'' \subseteq T'$.*

4.6 Proof of type (IC type language)

In the Section 3.6 (Chapter 3) we have stated and proved some propositions and lemmas (Proposition 3.6.2, Proposition 3.6.1, Lemma 3.6.1) that refer to the properties of the local binder and base components in GC language. Those results are used also in this section for the sake of proofs.

When the protocol that governs the internal interaction among components includes choices, this internal action is not supposed to be visible in the type, i.e. due to the type semantics, the type remains the same after the internal action. However, the type of the interfacing component (of the composite component) might change after the internal action. These types (before and after the internal action) are in a specific relation that we are now about to give a definition of.

The following definition and the proposition refer to a relation between types comparing the number of the received input and the number of values that the port is able to output.

Definition 4.6.1. *Let* $T' = X_b.\mathbb{L}'.\mathbb{R}'.\mathbb{F}'.Y'_b.[\mathtt{ch}]$ *and* $T = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}]$, $\alpha' \in \{\mathbb{L}', \mathbb{R}', \mathbb{F}'\}$ *and* $\alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\}$.

$$T' \subseteq T \textit{ if}$$

1. $d^{\alpha}_{ij} = N_{\alpha ij} : p_{\alpha ij}$ *then* $d^{\alpha'}_{ij} = N'_{\alpha' ij} : p_{\alpha' ij}$ *and* $N_{\alpha' ij} \geq N_{\alpha ij}$,
2. $d^{\alpha}_{ij} = \Omega_{\alpha, ij} : p_{\alpha ij}$ *than* $d^{\alpha'}_{ij} = \Omega : p_{\alpha' ij}$ *or* $d^{\alpha'}_{ij} = 0$.
3. $y_{i1}(b_{i1}) : \boldsymbol{B}_{i1} \in Y_b$ *then* $y_{i1}(b_{i1}) : \boldsymbol{B}'_{i1} \in Y'_b$ *and* $\boldsymbol{B}'_{i1} \leq \boldsymbol{B}_{i1}$.

The following propositions are crucial for proving the Subject Reduction theorem:

Proposition 4.6.1. *If* $T \subseteq T'$ *and* $T' \subseteq T''$ *then* $T \subseteq T''$

Proof. The relation "$\leq$" between numbers ($N_{\alpha ij}$) is a transitive relation. Assume that $\Omega_{\alpha ij} : p_{\alpha ij}$ is a dependency in T'. By the definition of the relation "$\subseteq$" it is the dependency in T'' and in T, or in T it is 0. If in T' some dependency is 0, also it is 0 in T, and in T'' can be either 0 or initial dependency. □

Proposition 4.6.2. *If* $T_r \subseteq T'_r$ *then* $T(LP, T_r, F) \subseteq T(LP, T'_r, F)$.

Proof. Immediate from the Definition 4.6.1 and Definition 4.4.3. □

The following proposition helps us with proofs and is a consequence of the rules of the semantics that describe the evolution of dependencies due to an input of a value.

Proposition 4.6.3. *If* $x_{1j} \in X_b$ *then* $X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\mathtt{ch}] \xrightarrow{x_{1j}(b_{1j})?} X_b.\mathbb{L}'.\mathbb{R}'.\mathbb{F}'.Y_b[\mathtt{ch}]$.

Proof. Since the collection rules from the Table 15 that describes the transitions of dependencies with an input, announce that in any possible case whether it is a per-each-value dependency, on initial one, or zero, or even if some port does not depend on a specific input port, the dependency can perform an input. □

Now we are ready to state and prove our Theorem 4.5.1 result:

Theorem 4.6.1 (Subject Reduction). *If $K \Downarrow T$ and $K \xrightarrow{\lambda(v)} K'$ and v has type b then $T \xrightarrow{\lambda(b)} T'$ and $K' \Downarrow T''$ where $T'' \subseteq T'$.*

Proof. By induction on the derivation of $K \xrightarrow{\lambda(v)} K'$ and by cases on the last rule applied.

[InpBase] We know that $K = [\tilde{x} > \tilde{y}]\{L\} \xrightarrow{x?(v)} [\tilde{x} > \tilde{y}]\{L'\}$, by inversion on the rule we have that $x \in \tilde{x}$ and that $L \xrightarrow{x?(v)} L'$, so the Proposition 3.6.1 holds.

By hypothesis $K \Downarrow T$ and since K is a base component, by the Definition 4.3.1

$$T = X_b.\{0\}.\{0\}.\{d^{\mathbb{F}}_{ij}\}.Y_b.[0]$$

where since $x \in \tilde{x}$, then $x(b) \in X_b$ and $b = \gamma(v)$. Moreover, if x is the j^{th} element of the matrix X_b, for every element y_{i1} of the matrix Y_b ($x_{1j} = x$) we have that

$$\mathbb{F} = \begin{cases} \{d^{\mathbb{F}}_{ij}\}_{n\times m} \text{ where } d^{\mathbb{F}}_{ij} = \mathit{count}(x_{1j}, \tilde{\sigma}^{y_{i1}}) : 0 & \text{if } x_{1j} \in \tilde{x}^{y_{i1}} \\ d^{\mathbb{F}}_{ij} = 0. & \text{otherwise} \end{cases}$$

Applying the rules [`InpZero`], [`InpDisc`] and [`InpPevF`] we have that

$$\forall \alpha \in \{\mathbb{L}, \mathbb{R}, \mathbb{F}\} \; d^{\alpha}_{ij} \xrightarrow{x_{1j}?} d'^{\alpha}_{ij}$$

Since the hypothesis of the Rule [`InpT`] holds, we conclude that exists T' such that $T \xrightarrow{x_{1j?}(b)} T'$.

If $x_{1j} \notin \tilde{x}^{y_{i1}}$ then $d^{\mathbb{F}}_{ij} = 0$, and applying the rule [`InpZero`] $d'^{\mathbb{F}}_{ij} = 0$. If $x_{1j} \in \tilde{x}^{y_{i1}}$ then $d^{\mathbb{F}}_{ij} = \mathit{count}(x_{1j}, \tilde{\sigma}^{y_{i1}}) : 0$ and by the rule [`InpPevF`] $d^{\mathbb{F}}_{ij} = \mathit{count}(x_{1j}, \tilde{\sigma}^{y_{i1}}) + 1 : 0$. Other dependencies are 0, so applying the Rule [`InpDisc`], they remain the same. Then $T' = X_b.\{0\}.\{0\}.\{d'^{\mathbb{F}}_{ij}\}.Y_b.[0]$. By the Definition 4.3.1 and Proposition 3.6.1 we conclude $K' \Downarrow T'$ where $T' \subseteq T'$.

[OutBase] We know that $[\tilde{x} > \tilde{y}]\{L\} \xrightarrow{\text{y!(v)}} [\tilde{x} > \tilde{y}]\{L'\}$, by inversion on the rule we have that $y \in \tilde{y}$ and that $L \xrightarrow{y!(v)} L'$, so the

Proposition 3.6.2 holds. Then we have that for all $x_{1j} \in \tilde{x^y}$ holds: *count*$(x_{1j}, \tilde{\sigma}^y) > 0$ (*).

By hypothesis we have that $K \Downarrow T$ and since K is a base component we have by the Definition 4.3.1

$$T = X_b.\{0\}.\{0\}.\{d^{\mathbb{F}}_{ij}\}.Y_b.[0]$$

where since $y \in \tilde{y}$, then $y(b) \in X_b$ and $b = \gamma(v)$. Moreover, if y is the i^{th} element of the matrix Y_b ($y_{i1} = y$) then

$$\mathbb{F} = \begin{cases} \{d^{\mathbb{F}}_{ij}\}_{n\times m} \text{ where } d^{\mathbb{F}}_{ij} = \mathit{count}(x_{1j}, \tilde{\sigma}^{y_{i1}})\!:\!0 & \text{if } x_{1j} \in \tilde{x}^{y_{i1}} \\ d^{\mathbb{F}}_{ij} = 0. & \text{otherwise} \end{cases}$$

and

$$Y_b = \begin{bmatrix} \cdots & y_{i1}(b)\!:\!\infty & \cdots \end{bmatrix}^t.$$

Since (*) holds, and we have that for the base component the boundary is infinite, we have that the hypothesis of the Rule [`OutT`] holds. So, exists T' such that $T \xrightarrow{y_{i1}!(\mathrm{b})} T'$, and $T' = X_b.\mathit{shrink}(\{0\}, \{0\}, \mathbb{F}, Y_b, i)[0]$. Note that $\infty - 1 = \infty$. By the Definition 4.3.1 and Proposition 3.6.2 we conclude that $K' \Downarrow T'$ where $T' \subseteq T'$.

[InpComp] We have that

$$K = [\tilde{x}\,\rangle\,\tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{x?v} \overbrace{[\tilde{x}\,\rangle\,\tilde{y}]\{G; r = K'_r, R; D; r[F]\}}^{=K'}.$$

By inversion on the rule we know that $x \in \tilde{x}$, and that exist K'_r and z such that $K_r \xrightarrow{z?v} K'_r$ and that $F = z \leftarrow x, F'$. Since $K \Downarrow T$, there exists T_r such that $K_r \Downarrow T_r$. By induction hypothesis, there exist T'_r and T''_r such that $T_r \xrightarrow{z?(b)} T'_r$ and $K'_r \Downarrow T''_r$, where $T''_r \subseteq T'_r$. Since the global protocol G does not evolve, neither the local protocols do. Let $LP = G \downharpoonright_r$. Then by the Definition 4.4.3 the extracted type of component K' is $T' = T(LP, T''_r, F)$

Since $x \in \tilde{x}$ and $K \Downarrow T = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b.[\mathtt{ch}]$, then by the Definition 4.4.3, the port x is the j^{th} element of the matrix X_b together with its basic type $\gamma(v) = b$. By the Proposition 4.6.3 we know that then exists T' such that $T \xrightarrow{x_{1j}?(b_x)} T'$. Since $T''_r \subseteq T'_r$ we have that $T(LP, T''_r, F) \subseteq T'$.

[OutComp] We have that

$$K = [\tilde{x}\rangle\, \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{y!v} \overbrace{[\tilde{x}\rangle\, \tilde{y}]\{G; r = K'_r, R; D; r[F]\}}^{=K'}$$

By inversion on the rule we know that $y \in \tilde{y}$, $K_r \xrightarrow{w!v} K'_r$ and that $F = y \leftarrow w, F'$. Since $K \Downarrow T$, there exists T_r such that $K_r \Downarrow T_r$. By induction hypothesis there exists T'_r such that $T_r \xrightarrow{w!(b_w)} T'_r$ and $K'_r \Downarrow T''_r$, $T''_r \subseteq T'_r$. Since G does not evolve, neither the local protocols evolve. Since K can perform an output, by the definition of the type extraction we have that all the dependencies in T for y_{i1} are satisfied, so there is T' such that $T \xrightarrow{y!(b_y)} T'$ where $\gamma(v) = b_w = b_y$ T' is uniquely determined by the rule [`OutT`]. Let $LP = G \downharpoonright_r$, by applying the rule [`OutT`] on T_r, we get the shape of T'_r. By the definition of the type extraction we get $T(LP, T''_r, F)$, with $T(LP, T''_r, F) \subseteq T'$.

[Internal] We know that $K = [\tilde{x}\rangle\, \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{\tau} [\tilde{x}\rangle\, \tilde{y}]\{G; r = K'_r, R; D; r[F]\} = K'$ by inversion on the rule we know that exists K'_r such that $K_r \xrightarrow{\tau} K'_r$. Since $K \Downarrow T$, there is T_r such that $K_r \Downarrow T_r$. By induction hypothesis there exists T'_r such that $T_r \xrightarrow{\tau} T'_r$, according to the rule $[InpInter]$ $T_r \xrightarrow{\tau} T_r$. Moreover, there exists Tr'' such that $T''_r \subseteq T_r$ and $K'_r \Downarrow T''_r$. Global protocol G did not evolve, hence none of the local protocols, so all the modified type of the subcomponents of a composite component stay conformant with their local protocols. Applying the [`InpInter`] we have that $T \xrightarrow{\tau} T$. By the definition of the type extraction from T''_r and LP (projection of G to role associated to K_r) since $T''_r \subseteq T_r$, than $T(LP, T''_r, F) \subseteq T$.

[InpChor] We know that $K = [\tilde{x}\rangle\, \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{\tau} [\tilde{x}\rangle\, \tilde{y}]\{G'; r = K'_r, R; D; r[F]\} = K'$ by inversion on the rule we know that exists K'_r such that $K_r \xrightarrow{x?(v)} K'_r \wedge D = r.z' \leftarrow p.u, D' \wedge G \xrightarrow{r?l<v>} G'$. By the rule [`InpInter`] we have that $T \xrightarrow{\tau} T$.

Let assign the subcomponents to their roles in the following way $R = r = K_r, r_2 = K_2, \ldots, r_m = K_m$ and since $K \Downarrow T$ then $K_r \Downarrow T_r, K_1 \Downarrow T_1, \ldots, K_m \Downarrow T_m$.

We have that $D = r.z' \leftarrow p.u, D' \ \wedge \ G \xrightarrow{r?l<v>} G'$and since K is well-typed, all its subcomponent's modified types are conformant with their local protocols, with input on x as the first action, i.e.

$$\mathcal{T}(T_r) \bowtie x?G' \downharpoonright_r, T_2 \bowtie x?G' \downharpoonright_{r_2}, \ldots, T_m \bowtie x?G' \downharpoonright_{r_m} .$$

Since x is the port of (only) component K_r, $G \downharpoonright r_i = G' \downharpoonright r_i, i = 2, \ldots, m$.

We can conclude that due to the semantic of the (modified) type that $\mathcal{T}(T_r) \xrightarrow{\text{x?}} \mathcal{T}'(T_r), T_2 \xrightarrow{\text{x?}} T_2, \ldots, T_m \xrightarrow{\text{x?}} T_m$.

Applying the rule $[InpConf]$ we have that $\mathcal{T}'(T_r) \bowtie G' \downharpoonright_r, T_2 \bowtie G' \downharpoonright_{r_2} \ldots, T_m \bowtie G' \downharpoonright_{r_m}$.
Note that $LP = G \downharpoonright_r$ and $LP' = G' \downharpoonright_r$.

By induction hypothesis exist T'_r and T''_r such that $T_r \xrightarrow{x?(b_x)} T'_r \wedge K'_r \Downarrow T''_r \wedge T''_r \subseteq T'_r$. Applying the previous proof for the rule $[\texttt{InpComp}]$ we have that $T'_r = T''_r$.

<u>We need prove that $T(LP', T''_r, F) \subseteq T$.</u>

We know that $x \in fp(LP)$ and let x be the the k^{th} element of the matrix X^r_b, where $T_r = X^r_b.\mathbb{L}^r.\mathbb{R}^r.\mathbb{F}^r.Y^r_b.[\texttt{ch}^r]$ $(x = x_{1k})$.

Let us consider the following cases:

1. $\neg\exists \ d^{\alpha^r}_{jk} \neq 0$ where $y_{j1} \in Y^r_b$ and forwards the values to an external environment ($y_{j1} \in F^o$) . By the type composition description $x \neq \texttt{ch}^r$. By the type extraction definition, the input on that port cannot create or discard the new dependencies and none of the dependencies in T were created via that port. By the definition of the type extraction $T(LP', T''_r, F) = T$.

2. $x \neq \texttt{ch}^r$. Since $T_r \xrightarrow{x_{1k}?(b_{1k})} T'_r$, (having $b_{1k} = b_x$). By the rule $[\texttt{InpT}]$, we know the type T'_r, and the values expected to be received by the protocol due to the type extraction are received, we get the same type as type T.

3. $x = \texttt{ch}^r$. With an input on x as a choice port, the protocol becomes restricted to one branch (up to the next unwrapping if it is a recursive protocol). By the type extraction procedure, in

the extracted type from T'_r and LP' we have some initial dependencies dropped or some actual values (not possible values) received for computing the output. By the definition of the relation "$\subseteq$", $T(LP.T''_r, F) \subseteq T$.

[OutChor] Analogous to the previous proof, considering the output and the rule [`OutT`] applied on the type T_r.

$\square$

Now we are ready to state and prove our Type Fidelity theorem:

Theorem 4.6.2 (Type fidelity). *If $K \Downarrow T$ and $T \xrightarrow{\lambda(b)} T'$ and $\lambda(b) \neq \tau$ then b is the type of a value v and $K \overset{\lambda(v)}{\Longrightarrow} K'$ and $K' \Downarrow T''$ where $T'' \subseteq T'$.*

Proof. Proof by induction on the structure of K.

<u>*Base case.*</u> Let K be the base component. If K is a base component of the type T then:

(Proof for the base case by induction on the derivation of $T \xrightarrow{\lambda(b)} T'$.)

[`InpT`]: $X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\mathtt{ch}] \xrightarrow{x_{1j}(b_{1j})?} X_b.\{(d^{\mathbb{L}}_{ij})'\}.\{(d^{\mathbb{R}}_{ij})'\}.\{(d^{\mathbb{F}}_{ij})'\}.Y_b[\mathtt{ch}]$, then by inversion we know that $d^{\alpha}_{ij} \xrightarrow{x_{1j}?} (d^{\alpha}_{ij})'$ and $x_{1j}(b_{1j}) \in X_b$. Since $x_{1j}(b_{1j}) \in X_b \Rightarrow \exists K' \mid K \xrightarrow{x_{1j}?(v)} K'$ (v is some value of the type b_{1j}). We need to prove that $K' \Downarrow T''$, for some $T'' \subseteq T'$.

– By applying the rules [`InpZero`], [`InpDisc`], [`InpIntF`] and [`InpPevF`] and by observing the Lemma 3.6.1 by the Definition 3.3.1 we conclude that $K' \Downarrow T'$, knowing that $T' \subseteq T'$.

[`OutT`]: $X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\mathtt{ch}] \xrightarrow{y_{i1}(b_{i1})!} X_b.shrink(\mathbb{L}, \mathbb{R}, \mathbb{F}, Y_b, i)[\mathtt{ch}]$. By inversion we know that

$\forall j \in \{1, ..., m\}\ \ d^{\alpha}_{ij} = N_{\alpha ij} : p_{\alpha ij} \vee d^{\alpha}_{ij} = 0\ \ \mathbf{B}_{i1}, N_{\alpha ij} > 0\ \ (i \in \{1, ..., n\})\ \wedge\ y_{i1}(b_{i1}) : B_{i1} \in Y_b$.

Since $K \Downarrow T$, the proof is straightforward. Let $K = [\tilde{x} > \tilde{y}]\{L\}$.

– $\forall j \in \{1, ..., m\}\ \ d^{\alpha}_{ij} = N_{\alpha ij} : p_{\alpha ij} \vee d^{\alpha}_{ij} = 0\ \ \mathbf{B}_{i1}, N_{\alpha ij} > 0$, by Lemma 3.6.1 $L \xrightarrow{\mathrm{y!(v)}} L'$.

– By the *Proposition* 3.6.1 and the *Definition* 3.6.1, knowing the form of T' on whose matrices is applied the *shrink* operation, we conclude that $K' \Downarrow T'$ and $T' \subseteq T'$.

Induction hypothesis.(1) Let $K = [\tilde{x} > \tilde{y}]\{G; r = \overline{K}, R; D; r[F]\}$ and let's assume that Theorem 4.5.2 holds for all the components "smaller" that K (subcomponents).

Proof for K. Let K be the composite component $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\}$ where $K \Downarrow T$ and $T \xrightarrow{\lambda(b)} T'$.
Let

$$T = X_b.\mathbb{L}.\mathbb{R}.\mathbb{F}.Y_b[\mathtt{ch}]$$

Since $K \Downarrow T$ then $\exists T_r : K_r \Downarrow T_r$ and let $T_r = X_b^r.\mathbb{L}^r.\mathbb{R}^r.\mathbb{F}^r.Y_b^r[\mathtt{ch}^r]$.
Moreover, since K is well-typed $\mathcal{T}(T_r) \bowtie G\downharpoonright_r = LP$.

[Case 1.] [$\lambda = x_{1j}?$] Since $T \xrightarrow{x_{1j}(b_{1j})} T'$ by [$\mathtt{InpT}$] we know that $x_{1j}(b_{1j}) \in X_b$. By the definition of the type extraction $\exists z_{1k}(b_{1j}) \in X_b^r$ such that $F = z \leftarrow x, F' \Rightarrow T_r \xrightarrow{z_{1k}(b_{1j})} T_r'$. By induction hypothesis there exists K_r' such that $K_r \xRightarrow{\text{z?(v)}} K_r'$. By repeatedly applying the rule [Internal] some number of times and then applying the rule [InpComp], we conclude that there is K' such that $K \xRightarrow{\text{x?(v)}} K'$. Since $K \Downarrow T$ and $K \xRightarrow{\text{x?(v)}} K'$ by repeatedly applying Theorem 3.5.1 (first for internal moves, then for the input), knowing that "$\subseteq$" is the partial order relation, we have that exists T'' such that $T'' \subseteq T'$ and $K' \Downarrow T''$.

[Case 2.] [$\lambda = y!$] Since $K \Downarrow T$ and $T \xrightarrow{y!(b)} T'$ we know by the Definition 3.4.1 that $\exists w : F = y \leftarrow w, F'$.

We have two possible cases:

2.1 $T_r \xrightarrow{w!(b)} T_r'$

2.2 $T_r \not\xrightarrow{w!(b)}$

If the case 2.1 holds, by induction hypothesis (1) there exist K_r' such that $K_r \xRightarrow{w!(v)} K_r'$ and T_r'' such that $K_r' \Downarrow T_r''$ and $T_r'' \subseteq T_r'$.

Repeatedly applying the rule [Internal] and then the rule [OutComp] we have that $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{\text{y!(v)}} K = [\tilde{x} > \tilde{y}]\{G; r = K_r', R; D; r[F]\}$. Since G did not move, all the projections remain the same, and since the output on the port w does not affect the conformance to the protocol, we can conclude that $\mathcal{T}(T_r)G \downharpoonleft_r = LP$. We have also that all the other types of subcomponents remain conformant with their local protocols because their types do not evolve neither does G. Since $K \Downarrow T$ and $K \xRightarrow{\text{y!(v)}} K'$ by repeatedly applying Theorem 3.5.1 (firts for internal moves, then for the output), and since "$\subseteq$" is the partial order relation, there is T'' such that $T'' \subseteq T'$ and $K' \Downarrow T''$.

If the case 2.2 holds, since the type T can output, but T_r cannot, we know that extracting the type we captured the values that are input, but still flowing, or the ones that protocol "promised" to give. Since all the dependencies of the port y are satisfied, and $F = y \leftarrow w, F'$, but w still has some unsatisfied dependencies, the only possible case is that w still needs to receive the values from the ports that are in $\mathit{fp}(LP)$. Now we need to prove that those values will eventually be received, so that T_r can output a value from the port w, and then we will be back on the case 2.1.

Let's take any input port in $\mathit{fp}(LP)$ without loss of generality that still needs values to be input, so that w can output:

$LP = G \downharpoonleft_r$

$T_r = X_b^r.\mathbb{L}^r.\mathbb{R}^r.\mathbb{F}^r.Y_b^r[\mathtt{ch}^r]$

$w = w_{i1}$, as the part of the i^{th} element of the matrix Y_b^r.

$\exists z_{1j} \in \mathit{fp}(LP) : z_{1j}(b^z) \in X_b^r \wedge (d_{ij}^{\alpha} = \Omega : p_{\alpha ij}$ or $d_{ij}^{\alpha} = 0 : p_{\alpha ij})$

Since $K \Downarrow T \Rightarrow \mathcal{T}(T_r) \bowtie LP$, and because $z_{1j} \in \mathit{fp}(LP) \Rightarrow \mathcal{T}(T_r) \bowtie \mathcal{C}[z_{1j}?\!:\!b^z.LP']$.

[*]Now we prove by the induction on the number of evolution steps of G, that G will have an input on z_{1j} and that the conformance relation is still satisfied.

Base case.

$G \xrightarrow{r?l_{z_{1j}}(v)} G'$

If $G \xrightarrow{r?l_{z_{1j}}(v)} G'$ then $LP = z_{1j}? : b^z.LP'$. By induction hypothesis (1) $K_r \xRightarrow{z_{1j}?(v)} K_r'$. So, the dependency is satisfied, hence: $K_r \xrightarrow{x_{1j}?(v)} K_r' \xrightarrow{w_{i1}(v)} K_r''$. Applying the rules [Internal] and [InpChor] we have that $K \xRightarrow{\text{y!(v)}} K'$.

[$\triangle$] All the other subcomponents did not evolve, neither their types, and since the port z_{1j} is the free port of LP, only the protocol projection for the component with the role r changed, and others remained the same. Moreover [InpConf] $\mathcal{T}(T_r') \bowtie LP'$.

Now we need to distinguish two different cases. First one is if $z_{1j} \neq \mathtt{ch}$.

Then by the Definition 3.4.1, [$\triangle$] $\Rightarrow K' \Downarrow T'$ and $T' \subseteq T'$.

Second one is when $z_{1j} = \mathtt{ch}$. Then $G \xrightarrow{r?l_{z_{1j}}(v)} G_1G''$ or $G \xrightarrow{r?l_{z_{1j}}(v)} G_2G''$.

If the choice is G_1G' (the same if it is G_2), by the $Definition$ 3.4.1 all the transitive dependencies created via branch G_1, are dropped (if they are initial dependencies) or have a number of values from an input port available for a specific output increased by one. In other words, one possible value becomes the actual value for the transitive dependencies created via branch G_1.

The transitive dependencies created via branch G_2 become 0, and possibly some boundaries become 0, if it is a "one shot protocol" where $G'' = \mathbf{end}$, or remain the same if $G'' = \mu\mathbf{X}.G$.

Hence, by the $Definition$ 3.4.1, and [$\triangle$] we have that $K \Downarrow T(G'' \downharpoonright_r, T_r', F)$ and $T(G'' \downharpoonright_r, T_r', F) \subseteq T'$.

Induction hypothesis. (2) Lets assume that [*] holds for [k-1] steps.

$G \xrightarrow{pl_1(v)} G_1 \xrightarrow{pl_2(v)} G_2 \cdots \xrightarrow{pl_{k-1}(v)} G_{k-1} \xrightarrow{r?l_{z_{1j}}(v)} G'$.

Let $G \xrightarrow{pl_1(v)} G_1 \xrightarrow{pl_2(v)} G_2 \cdots \xrightarrow{pl_{k-1}(v)} G_{k-1} \xrightarrow{pl_k(v)} G_k \xrightarrow{r?l_{x'}(v)} G'$, where p can be the role of any subcomponent of K, and $p \in \{p?, p!\}$.

Let $G \xrightarrow{pl_1(v)} G_1$.

If $p \neq r$ then by the Rules [BranchConf] ([SelectConf]) and [InpConf] ([OutConf]) the type of the component with the role p

stays conformant with its local protocol. All the other projections of the protocol G_1 did not change, so all the components are conformant with their projections of the protocol G_1. Since $K = [\tilde{x} > \tilde{y}]\{G, r = \overline{K}, R; D; r[F]\}$ and $G \downharpoonright_r = G_1 \downharpoonright_r = LP$ the extracted type for the component K remains the same, and all the subcomponents are conformant with their local protocols, so we can conclude that $K' = [\tilde{x} > \tilde{y}]\{G_1, r = K_r, R; D; r[F]\} \Downarrow T$. By the induction hypothesis (2) K_r will have an input on z_{1j}, hence the dependency of w_{i1} will be satisfied. Now, by applying the previously proved on the component K' with a type T, where $T \xrightarrow{\text{y!(b)}} T'$, we have that exist K'', T'', such that $K \xRightarrow{\text{y!(v)}} K''$, where $K \Downarrow T''$ and $T'' \subseteq T'$.

Let $p = r?$ and $D = r.z_{1k} \xleftarrow{l_1} q, v, D$. By the induction hypothesis, and applying the base case on the component K_r, where $K_r \xrightarrow{z_{1k}\text{?(v)}} K'_r$ and $K_r \Downarrow T_r$, we know that exists T''_r such that $K'_r \Downarrow T''_r$. Next, by the rule [InpChor] we have that $K \xrightarrow{\tau} K'$ where $K' = [\tilde{x} > \tilde{y}]\{G_1, r = K_r, R; D; r[F]\}$ and exists T'' such that $K'_r \Downarrow T''$ and $T'' \subseteq T'$. Moreover, applying the rule [InpConf] we have $\mathcal{T}(T'_r) \bowtie G_1 \downharpoonright_r$. The types of the other components remained the same as did their local protocols. Finally, we now apply the induction hypothesis (2) on K', with a protocol G_1, that has $k-1$ steps up to the input on k_{1j}.

Similar proof for $p = r!$. $\square$

In conclusion following the rules [Internal], [OutChor] and [OutComp], [Definition 3.4.1] we proved that $K \Downarrow T \wedge T \xrightarrow{y!(b)} T' \Rightarrow K \xRightarrow{y!(v)} K' \wedge K' \Downarrow T''$ where $T'' \subseteq T'$.

$\square$

Chapter 5

Concluding remarks

In this chapter we compare the work of this thesis with the relevant literature; then, we present some concluding remarks and discuss some possible future work as the extension of the type languages presented in this thesis.

5.1 Related work

We place our approach in the behavioural types setting ([12]) since our types evolve in order to explain component behaviour (cf. Theorem 3.5.1, Theorem 4.5.1), in contrast with classic subject reduction results where the type is preserved. In the realm of behavioural types, we distinguish Multiparty Asynchronous Session Types [11] which actually lay the basis for the protocol language of our target model [5]. The model builds on the idea that protocols can be used to directly program the interaction, and not only serve as a specification/verification mechanism, following the approach of choreographic programming [4, 19]. Moreover, we need to mention the tool for typechecking protocols StMungo [6] that is based on the integration of session types and typestate which consists of a formal translation of session types for communication channels into typestate specifications for channel objects, the latter defines the order in which the methods of the channel objects can be called. This tool can guide the user in the design and implementation of distributed multiparty communication-based programs with guarantees on communication safety and soundness.

We discuss some closely related work, starting by Open Multiparty Sessions [3] which to some extent shares the same goals and the same background ([11]). The approach in [3] targets the composition of protocols by considering that one of the participants can actually be instantiated by an external environment. Two protocols can then be connected if there is a participant in each that can serve as the interface to the other interaction. So protocols can be viewed as the units of composition instead of components like in our case, and reusing such protocols in other compositions requires compatibility between the I/O actions which are prescribed for the interfacing role. The main difference is therefore that we consider components that are potentially more reusable considering the I/O flexibility provided the reactive flavour.

We also identify the CHOReVOLUTION [13]project where the assembly of services via a choreography is addressed. The I/O flexibility is provided by adapters at assembly time that can solve I/O interface mismatches between service and choreography. We remark that the CHOReVOLUTION approach is at a very mature state (including tool support [2]), where however an assembly of services cannot be provided as a unit of reuse (like our composite components). Differently, our type-based approach aims at abstracting from the implementation and provides more general support for component substitution and reuse. Similar to our research, the model of *service based architectures* (SOA's) [17] exposes components that exchange services between each other via ports. However, the authors do not present the type language, where some ideas we presented for extracting a type of a component could be used for the model they present.

In [14] the authors present a novel type system and type inference algorithm that prevent interconnection and message-handling errors when assembling component-based communication systems. Their type system ensures that typable assemblages do not exhibit the targeted class of errors, where our type system focus on capturing the behaviour of the component in order to avoid any errors.

We may also find the notion of distributed components is the Signal Calculus (SC) [8], where in each component a process is located. The type-based approach presented in [9] addresses the issue of ensuring SC local processes are composed and interact in a way such that they follow a well-defined protocol of interaction. Our model embeds the protocol in the operational semantics so such coordination is ensured by construction. Our types focus on a different purpose of ensuring data dependencies are met in order to ensure components are not stuck in the sense of the type fidelity result.

5.2 Conclusion

We have developed two type languages for components that capture their reactive behaviour together with their capabilities: First one (EC type language [24]) is modelled in a choice-free subset of GC language [5] for describing simpler-composed components, obtaining the easier type extraction procedure; Second one (IC type language) is modelled in a full GC language, obtaining the wider spectrum of possible composition scenarios that can be described with that type language.

Our types describe the ability of components to receive and send values, while tracking different kinds of dependencies (per-each-value and initial ones) and specifying constraints on the boundary of the number of values that a component can emit. We show how types of components can be extracted (inferred) and prove that types faithfully capture component behaviour by means of Subject Reduction and Type fidelity theorems.

5.3 Future work

Immediate directions for future work include providing a characterisation of the substitution principle [15] based in our types. In this work as a *meta data* we introduced a kind of a subtyping relation. However, the subtyping relation and the whole theory behind it remains to be further explored. Further future challenge is to assess the usability of our theoretical model to concrete applications, for specification an the reuse of components. Moreover, our safety results enable to use types as starting point either to apply other verification techniques to check properties of data components exchange [**BodeiDFG17**], to enforce security policies [**DeganoFGM12**], or to extract all the possible ways of communication between two (or more components), which can be the upcoming task. This forthcoming task can be used in making component-based system patterns.

Bibliography

[1] Amazon Web Services, Inc. *AWS Lambda: Developer Guide*. 2019. URL: https://docs.aws.amazon.com/en_pv/lambda/latest/dg/lambda-dg.pdf#welcome.

[2] Marco Autili et al. "CHOReVOLUTION: Automating the Realization of Highly–Collaborative Distributed Applications". In: *21th International Conference on Coordination Languages and Models (COORDINATION)*. Ed. by Hanne Riis Nielson and Emilio Tuosto. Vol. LNCS-11533. Coordination Models and Languages. Part 2: Tools (1). Springer International Publishing, June 2019, pp. 92–108. DOI: 10.1007/978-3-030-22397-7_6.

[3] Franco Barbanera and Mariangiola Dezani-Ciancaglini. "Open Multiparty Sessions". In: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*. Ed. by Massimo Bartoletti et al. Vol. 304. EPTCS. 2019, pp. 77–96. DOI: 10.4204/EPTCS.304.6.

[4] Marco Carbone and Fabrizio Montesi. "Deadlock-freedom-by-design: multiparty asynchronous global programming". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 263–274. DOI: 10.1145/2429069.2429101.

[5] Marco Carbone, Fabrizio Montesi, and Hugo Torres Vieira. *Choreographies for Reactive Programming*. CoRR abs/1801.08107. 2018. URL: http://arxiv.org/abs/1801.08107.

[6] Ornela Dardha et al. "Mungo and StMungo: Tools for Typechecking Protocols in Java". In: *Behavioural Types: from Theory to Tools* (2017), p. 309.

[7] Mariangiola Dezani-Ciancaglini, Luca Padovani, and Jovanka Pantovic. "Session Type Isomorphisms". In: *Electronic Proceedings in Theoretical Computer Science* 155 (June 2014), pp. 61–71. DOI: 10.4204/eptcs.155.9. URL: https://doi.org/10.4204%2Feptcs.155.9.

[8] Gian Luigi Ferrari, Roberto Guanciale, and Daniele Strollo. "JSCL: A Middleware for Service Coordination". In: *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006*. Ed. by Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge. Vol. 4229. Lecture Notes in Computer Science. Springer, 2006, pp. 46–60. DOI: 10.1007/11888116_4.

[9] GianLuigi Ferrari et al. "Coordination Via Types in an Event-Based Framework". In: *Formal Techniques for Networked and Distributed Systems - FORTE 2007, 27th IFIP WG 6.1 International Conference, Tallinn, Estonia, June 27-29, 2007, Proceedings*. Ed. by John Derrick and Jüri Vain. Vol. 4574. Lecture Notes in Computer Science. Springer, 2007, pp. 66–80. DOI: 10.1007/978-3-540-73196-2_5.

[10] Nicola Dragoniand Saverio Giallorenzo et al. "Microservices: Yesterday, today, and tomorrow". In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Springer, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[11] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types". In: *J. ACM* 63.1 (2016), 9:1–9:67. DOI: 10.1145/2827695.

[12] Hans Hüttel et al. "Foundations of Session Types and Behavioural Contracts". In: *ACM Comput. Surv.* 49.1 (2016), 3:1–3:36. DOI: 10.1145/2873052.

[13] S. Keller, M. Autili, and M. Tivoli. *CHOReVOLUTION project*. http://www.chorevolution.eu. 2014.

[14] Michael Lienhardt et al. "Typing Component-Based Communication Systems". In: *Formal Techniques for Distributed Systems*. Springer Berlin Heidelberg, 2009, pp. 167–181. DOI: 10.1007/978-3-642-02138-1_11. URL: https://doi.org/10.1007%2F978-3-642-02138-1_11.

[15] Barbara Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (1994), pp. 1811–1841. DOI: 10.1145/197320.197383.

[16] Theo Lynn et al. "The Internet of Things: Definitions, Key Concepts, and Reference Architectures". In: *The Cloud-to-Thing Continuum: Opportunities and Challenges in Cloud, Fog and Edge Computing*. Ed. by Theo Lynn et al. Cham: Springer International Publishing, 2020, pp. 1–22. ISBN: 978-3-030-41110-7. DOI: 10.1007/978-3-030-41110-7_1. URL: https://doi.org/10.1007/978-3-030-41110-7_1.

[17] A. Malkis and D. Marmsoler. "A Model of Service-Oriented Architectures". In: *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2015, pp. 110–119. DOI: 10.1109/SBCARS.2015.22.

[18] M. Douglas Mcllroy. "Mass Produced Software Components". In: *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Garmisch, 1969, pp. 138–155.

[19] Fabrizio Montesi. "Choreographic Programming". PhD thesis. IT University of Copenhagen, 2013. URL: http://fabriziomontesi.com/files/choreographic_programming.pdf.

[20] Object Management Group, Inc. (OMG). *Business Process Model and Notation, specification version 2.0.2*. 2014. URL: https://www.omg.org/spec/BPMN/2.0.2/.

[21] Joachim Parrow. "Chapter 8. An Introduction to the π-Calculus". In: (Dec. 2001). DOI: 10.1016/B978-044482830-9/50026-6.

[22] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[23] Zorica Savanovic, Letterio Galletta, and Hugo Torres Vieira. "A type language for message passing component-based systems". In: *Proceedings 13th Interaction and Concurrency Experience, ICE 2020, Online, 19 June 2020*. Ed. by Julien Lange et al. Vol. 324. EPTCS. 2020, pp. 3–24. DOI: 10.4204/EPTCS.324.3. URL: https://doi.org/10.4204/EPTCS.324.3.

[24] Zorica Savanović, Letterio Galletta, and Hugo Torres Vieira. "A type language for message passing component-based systems". In: *Electronic Proceedings in Theoretical Computer Science* 324 (Sept. 2020), pp. 3–24. ISSN: 2075-2180. DOI: 10.4204/eptcs.324.3. URL: http://dx.doi.org/10.4204/EPTCS.324.3.

[25] W3C WS-CDL Working Group. *Web Services Choreography Description Language Version 1.0*. 2004. URL: http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.