A type language for message passing component-based systems

Component-based development is challenging in a distributed setting, for starters considering programming a task may involve the assembly of loosely-coupled remote components. In order for the task to be fulfilled, the supporting interaction among components should follow a well-defined protocol. In this paper we address 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 reactive to messages in a flexible way. Our contribution is at the level of the type language that allows to capture component behaviour so as to check its compatibility with a protocol. We show the correspondence of component and type behaviours, which entails a progress property for components.

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 Development (CBD) [11], where the idea is to build systems relying on the composition of loosely-coupled and independent units called components. A component is a software element that is replaceable and reusable, and that offers a well-defined functionality via an external interface which encapsulates the implementation (e.g., components range from software packages that aggregate a set of related functions to web services, just to mention a few).

The motivations behind CBD are, on the one hand, to increase development efficiency and lower the costs (by building a system from pre-existing 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., [7]) 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.

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 [14] or the choreography diagrams of BPMN [13]. A component should be able to carry out a certain sequence of input/output actions in order to fulfil its role in the protocol. One way to accomplish this is to implement a component in a way that prescribes a strict sequence of I/O actions, that should precisely match 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 communication 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 that requires the classification is sent multiple times?

In contrast, a more flexible design choice inspired in reactive programming is to design components so that they can respond to external stimulus without any specific I/O sequence. The reactive programming principle for building such components is to consider 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, there is the 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?

Submitted to:
ICE 2020

Carbone, Montesi and Vieira [6] proposed a language that supports the development of distributed systems by combining the notions of reactive components with choreographic specifications of communication protocols [12]. 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, by abstracting from the number of supported reactions, e.g., if a component can (always) perform a computation reacting to some data inputs, then it can be used in different protocols that require such computation to be carried out a different number of times; by abstracting from message ordering, e.g., 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 shouldn't be necessary to inspect the inner workings in order to asses if it is usable in a determined context for the purpose of *off-the-shelf* substitution or 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. [6] 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 time that the component is used "off-the-shelf" we need to check if the reactions supported by the component are enough to implement its part in the protocol.

In this work we consider a different approach, avoiding the implementation check each time a component is to be used. Firstly, we introduce a type language that characterises the reactive behaviour of components. 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".

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 to be emitted always (for each output) require a specific set of inputs (dubbed *per each value* dependencies). 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 extraction procedures are sound with respect to the semantics of the Governed Components (GC) language [6], considering here the choice-free subset of the GC language and leaving a full account of the language to future work. 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.

The rest of the paper is organised as follows. We first present the GC language in Section 2. Then in Section 3 we intuitively introduce our type language through a motivating example based on AWS Lambda [2] where we point out different scenarios that might occur while composing components and how our types allow to describe certain behavioural patterns. Section 4 introduces the syntax and the semantics of our types. Then, we define the type extraction for base components in Section 4.1, whereas the type extraction for composite components in Section 4.2. In Section 4.3 we present our results. Section 5 concludes, discusses related work and gives some future issues. Finally, in order to support the reviewing, we present the proof sketches in Appendix.

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

Table 1: Syntax of Governed Components

2 Background: Governed Components Language

In this section we briefly introduce the GC language, focusing on the main points that allow to grasp the essence of the model and to support a self-contained understanding of this paper. We refer the interested reader to [6] for a full account of the language. The syntax of the (protocol choice-free fragment of the) GC language is given in Table 1. 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.

In the case of a base component the implementation is given by the 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 (input) ports $\tilde{x}$. So, we say that component's ability to output a value may depend on the ones received, 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 runtime (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 in σ_1 the oldest values received are stored, in σ_2 the second-oldest values, and so on and so forth up to σ_k).

The implementation of a composite component, represented by $\{G;R;D;r[F]\}$, 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., $\mathsf{r} = K$ denotes 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 (e.g., $\mathsf{p}.x \xleftarrow{\ell} \mathsf{q}.y$ states that a message with a label ℓ is emitted on port y of the component assigned to role q, and to be received on port x of the component assigned to role p). Ports are uniquely associated to message labels (ℓ) in such way that each communication step in the protocol has a precise mapping to a port, i.e., 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. Finally, subterm $r[F]$ is used to specify the externally-observable behaviour: the (only interfacing) subcomponent responsible for the interaction with the external environment is identified (by its role r) and the respective connection between ports is provided by forwarders (F). The idea 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 (x and y are the ports of the composite component).

Protocol specifications prescribe the interaction among a set of parties identified by roles. Communication term $\mathsf{p} \xrightarrow{\ell} \tilde{\mathsf{q}}; G$ specifies that role p sends the message labelled ℓ to the (nonempty) set of roles $\tilde{\mathsf{q}}$, after which the protocol continues as specified by G. The difference between this work and [6] is the absence of branching. Then we have terms $\mu\mathbf{X}.G$ and X for specifying recursive protocols. Finally, term **end** defines the termination of the protocol.

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 $y!v$ captures an output on port y of a value v, and label τ denotes an internal move.

The rules that describe the behaviour of components are presented in two parts, addressing base and composite components separately. We present only the main rules, the full semantics can be found in [6].

$$\frac{L \xrightarrow{y!v} L' \quad y \in \tilde{y}}{[\tilde{x}\rangle \tilde{y}]\{L\} \xrightarrow{y!v} [\tilde{x}\rangle \tilde{y}]\{L'\}}\ \mathsf{OutBase} \qquad \frac{L \xrightarrow{x?v} L' \quad x \in \tilde{x}}{[\tilde{x}\rangle \tilde{y}]\{L\} \xrightarrow{x?v} [\tilde{x}\rangle \tilde{y}]\{L'\}}\ \mathsf{InpBase}$$

Table 2: Semantics of base components

Rules OutBase and InpBase that are given in Table 2 capture base component behaviour, and are defined relying on transitions exhibited by local binders, denoted $L \xrightarrow{\lambda} L'$. Rule OutBase states that if local binders L can exhibit an output of value v on port y, where 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. We omit here the rules for local binders (see [6]) and provide only an informal account for them. Essentially, a (runtime) 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 ($x \notin \tilde{x}$) then 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. For this to happen, the oldest mapping in queue $\tilde{\sigma}$ must be complete (i.e., assign values to all of $\tilde{x}$) at which point function f may be computed, the result which 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 3. 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 $\mathsf{r} = K$ captures the relation between component K and role r, which is specified as the interfacing role ($\mathsf{r}[F]$). Rule OutComp allows for the interfacing component K to send a value v to the external environment via one of the ports of the composite component (y). Notice that the connection between the port of the interfacing component (z) and the port of the composite component (y) 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 $\mathsf{p} = K$, 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 $D = \mathsf{q}.z \xleftarrow{\ell} \mathsf{p}.u, D'$ 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 $G \xrightarrow{\mathsf{p}!\ell\langle v\rangle} G'$ realises the component governing by 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. The rules

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

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

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

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

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

Table 3: Semantics of composite components

for protocol transitions are given in [6]. We do not present here the rules for the semantics of protocols as the key idea is that *there is* a well-defined semantics of protocols. Naturally which 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, by saying that receiving component K can exhibit the respective input transition, that the distribution binder specifies the relation of message label ℓ with receiver role q and port z, and that protocol G prescribes the input of a value.

3 Motivating example: microservices for Image Recognition System

In order to further motivate GC and also to introduce our typing approach, we now informally discuss an example inspired on a microservices scenario [2] 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 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 calculus 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. Interaction between these two components is governed by global protocol G, that can be described in the following way:

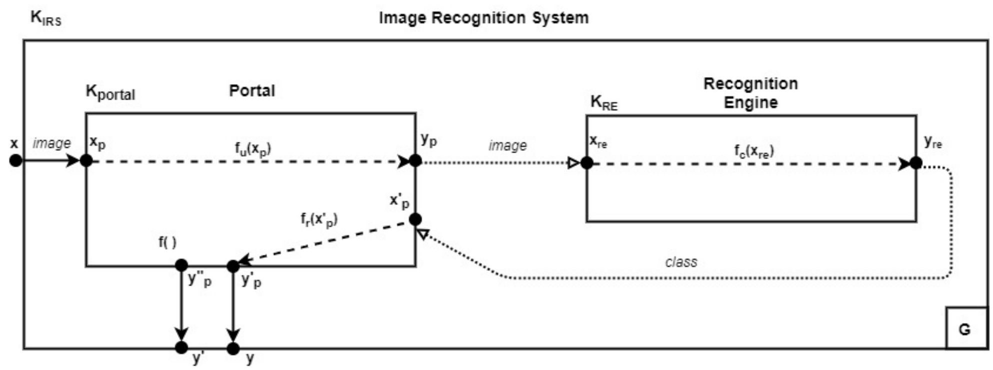
$$\mathsf{Portal} \xrightarrow{image} \mathsf{RE}; \mathsf{RE} \xrightarrow{class} \mathsf{Portal}$$

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

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

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. Below, we show 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 regarding their behavioural capabilities. In particular, for each output port there are constraints which comprise three pieces of information: what type of values are emitted; what is the maximum number of values that can be emitted; and 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 in y_p the component can emit *image*s 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:

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

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:

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

where G is the already described one-shot protocol

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

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 not 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_{RS}:

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

Let us now consider a recursive version of protocol

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

is used instead (i.e., $K'_{IRS} = [x\rangle\, y, y']\{G'; \mathsf{Portal} = K_{Portal}, \mathsf{RE} = K_{RE}; D; \mathsf{Portal}[F]\}$). The idea now is that for each *image* received a *class* is produced. So, *class* may be emitted on 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 now that a component K'_{Portal} (is 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

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

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}

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

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

Table 4: Type Syntax

4 A type language for the components

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

Syntax The syntax of types is presented in Table 4 and some explanations follow. A type T consists in 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 $\mathbf{C}$, 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 in ports, so as to ensure that no unexpected values arise. Each constraint in $\mathbf{C}$ contains a triple of the form $y(b) : \mathbf{B} : [\mathbf{D}]$, which describes the type (b) of values sent via y, the capability ($\mathbf{B}$) of y and the dependencies ($\mathbf{D}$) of y on the input ports. Capability $\mathbf{B}$ 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.

Semantics 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 5. 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.

We briefly describe the rules shown in Table 5. 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 $\mathbf{D}$ ($dom(\mathbf{D}) = \{x \mid \mathbf{D} = \{x : M\} \uplus \mathbf{D}'\}$), 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 [T4, T5, T6] 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 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

$$\frac{x \notin dom[\mathbf{D}]}{y(b) : \mathbf{B} : [\mathbf{D}] \xrightarrow{x?} y(b) : \mathbf{B} : [\mathbf{D}]} \; [T1] \qquad \frac{}{y(b) : \mathbf{B} : [\{x : \Omega\} \uplus \mathbf{D}] \xrightarrow{x?} y(b) : \mathbf{B} : [\mathbf{D}]} \; [T2]$$

$$\frac{}{y(b) : \mathbf{B} : [\{x{:}N\} \uplus \mathbf{D}] \xrightarrow{x?} y(b) : \mathbf{B} : [\{x{:}N+1\} \uplus \mathbf{D}]} \; [T3] \qquad \frac{}{T \xrightarrow{\tau} T} \; [T4]$$

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

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

Table 5: Type Semantics

dependencies present (which are dropped once satisfied), only per each value dependencies.

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

Example 4.1. *We revisit the type of component K_{Portal} shown in Section 3*

$$< \{x_p(i), x'_p(c)\}; \{y_p(i) : \infty : [\{x_p{:}N_1\}], y'_p(c) : \infty : [\{x'_p{:}N_2\}], y''_p(v) : \infty : [\emptyset]\} >$$

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:

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

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.

4.1 Type extraction for base components

We describe the procedure that allows to (automatically) extract the type of a component, focusing first on the case of base components. 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. First, we assume that from each function $f(\tilde{x})$ used in a local binder, we can infer the respective function type. 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 to obtain the list of respective types (e.g., $\gamma(1, \texttt{hello}) = integer, string$). 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:

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

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. We may now define our type extraction procedure for base components:

Definition 4.1 (Type Extraction for a Base Component)**.** *Let*

$$[\tilde{x} > \tilde{y}]\{y_1 = f_{y_1}(\tilde{x}^{y_1}) < \tilde{\sigma}^{y_1}, \ldots, y_k = f_{y_k}(\tilde{x}^{y_k}) < \tilde{\sigma}^{y_k}\}$$

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 $\gamma(f_{y_i}) = \tilde{b}^{y_i} \to b'_i$ for any $i \in 1, \ldots, k$ and that $\tilde{b}^{y_i} = \gamma(\tilde{x}^{y_i})$ for any $i \in 1, \ldots, k$ 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

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

In Definition 4.1 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 the 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 4.1 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 $\mathbf{C}$, 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 $count(x, \sigma^{y_i})$.

From an operation perspective, Definition 4.1 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 4.2. *Consider our running example from Section 3, 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 then $\boldsymbol{C} = \{y_p(i) : \infty : [\{x_p : 0\}], y'_p(c) : \infty : [\{x'_p : 0\}], y''_p(v) : \infty : [\emptyset]\}$ and the extracted type of the component K_{Portal} is $< X_b; \boldsymbol{C} >$.

4.2 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 presented in [6], 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 inaction (**end**). Our local protocols differ from the ones used in [6] 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 4.3. *Let G be the (one-shot) protocol $G = \mathsf{Portal} \xrightarrow{image} \mathsf{RE}; \mathsf{RE} \xrightarrow{class} \mathsf{Portal}; \mathbf{end}$ from Section 3 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 $G\downarrow_{\mathsf{Portal}}$ 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 $\mathsf{RE}.x_{re} \xleftarrow{image} \mathsf{Portal}.y_p, \mathsf{Portal}.x'_p \xleftarrow{class} \mathsf{RE}.y_{re}$. 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 $\mathscr{C}$, so as to abstract from the entire local protocol and focus on specific parts and we define it as: $\mathscr{C}[\,\cdot\,] ::= x?:b.\mathscr{C}[\,\cdot\,] \mid y!:b.\mathscr{C}[\,\cdot\,] \mid \cdot$. 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, 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 $\mathscr{C}[\mu\mathbf{X}.\mathscr{C}'[y!:b'.\mathscr{C}''[x?:b.LP']]]$. 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 $\mathscr{C}'''[x?:b.LP'']$ where $x,y \notin fp(\mathscr{C}'''[\cdot])$. We denote by $\mathbf{vf}(LP,x,y)$ that there is a value flowing from y to x in LP, in which case $\mathbf{vf}(LP,x,y)=1$, otherwise $\mathbf{vf}(LP,x,y)=0$. We will return to this notion in the context of the extraction of the dependencies of the output ports, discussed next.

Kinds of dependencies Composite components comprise two kinds of dependencies between output ports and input ports, illustrated in Figure 1 and Figure 2, which are dubbed direct and transitive, respectively.

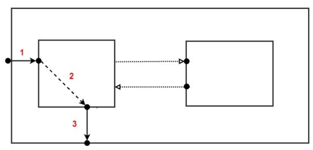


Figure 1: Direct Dependency

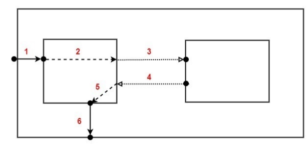


Figure 2: Transitive Dependency

We gather the set of direct dependencies, i.e., when external output ports directly depend on external input ports (see Figure 1), 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 $\mathbf{C}$ whenever both ports are external and preserving the kind of dependency M so as to lift it to the outer interface.

For transitive dependencies (see Figure 2) 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 $x',y' \in fp(LP)$ and where $i \in \{1,2,3\}$ as follows: $y' \diamond_1^{LP} x'$ if $LP = \mathscr{C}[y'!:b^{y'}.\mathscr{C}'[x'?:b^{x'}.LP']]$ and $x',y' \notin rep(LP)$; $y' \diamond_2^{LP} x'$ if $LP = \mathscr{C}[y'!:b^{y'}.\mathscr{C}'[\mu\mathbf{X}.\mathscr{C}''[x'?:b^{x'}.LP']]]$ and $y' \notin rep(LP)$; $y' \diamond_3^{LP} x'$ if $LP = \mathscr{C}[\mu\mathbf{X}.\mathscr{C}'[y'!:b^{y'}.\mathscr{C}''[x'?:b^{x'}.LP']]]$. 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 $[\tilde{x}' \rangle \tilde{y}']\{G;r=K,R;D;\mathrm{r}[F]\}$ be the 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:

$$\begin{array}{lcl}
\eta & = & \mathbf{C} = \{y(b_1):\mathbf{B}:[\{x':M'\} \uplus \mathbf{D}'], y'(b_2):\mathbf{B}':[\{x:M\} \uplus \mathbf{D}]\} \uplus \mathbf{C}' \\
 & & \wedge\, x \in F^i \wedge y \in F^o \wedge y' \diamond_i^{LP} x' \\
 & & \\
\mathbf{D}_t(\mathbf{C},F,LP,y) & \triangleq & \{x:\Omega \mid \eta \wedge i \in \{1,2\} \wedge M \not\succ 0\} \\
 & & \bigcup \\
 & & \{x:\Omega \mid \eta \wedge i = 3 \wedge (M = \Omega \vee (M' = \Omega \wedge M = 0 \wedge \mathbf{vf}(LP,x',y') = 0))\} \\
 & & \bigcup \\
 & & \{x:(N+N'+\mathbf{vf}(LP,x',y')) \mid \eta \wedge i = 3 \wedge M = N \wedge M' = N'\}
\end{array}$$

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 $\mathbf{C}$ 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 ($\mathbf{vf}(LP,x',y') = 0$) 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 $\mathbf{vf}(LP,x',y') = 1$ 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 $\mathbf{pr}(\ ,\ )$ that gives priority to per each value dependencies (with respect to "initial"). The definition of priority follows expected lines (see Appendix B) and builds on the property (cf. Proposition $B.1$) 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:

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

Boundaries 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 (an 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:

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

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

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

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 an 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.

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

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 4.2 (Type Extraction for a Composite Component)**.** *Let $[\tilde{x}\rangle\,\tilde{y}]\{G; r = K, R; D; \mathrm{r}[F]\}$ 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 4.4. *Let us extract the type of component K_{IRS} from Section 3 considering protocol* $G = \mathsf{Portal} \xrightarrow{image} \mathsf{RE}; \mathsf{RE} \xrightarrow{class} \mathsf{Portal}; \mathbf{end}$. *The type of interfacing component K_{Portal} is*

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

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 $\boldsymbol{ren}(F, x_p) = x$, $\boldsymbol{ren}(F, y'_p) = y$ and $\boldsymbol{ren}(F, y''_p) = y'$. 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 $\boldsymbol{ren}(y''_p(v) : \infty : [\emptyset])$, 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

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

4.3 Type Safety

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 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 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 $\mathscr{T}$ (see Appendix D). 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. By $\mathscr{T}(F, T)$ we denote the modified type that results from abstracting such external dependencies in T, relying on forwarders F, namely by considering per each value dependencies for external input ports are unbounded (∞) and dropping initial dependencies. The rules for the semantics of modified types are the same as the ones shown in Section 4, the only implicit difference for modified types is that decrementing an unbounded dependency has no effect.

The definition of the conformance relation (see Appendix E) is given by induction of the structure of the local protocol and it is characterised by judgments of the form $\Gamma \vdash \mathscr{T} \bowtie LP$, where Γ is a type environment that handles protocol recursion (i.e., Γ maps recursion variables to modified types). We report and comment here only on the rules for input and output:

$$\frac{\mathscr{T} \xrightarrow{x?(b)} \mathscr{T}' \quad \Gamma \vdash \mathscr{T}' \bowtie LP}{\Gamma \vdash \mathscr{T} \bowtie x?:b.LP}\ [InpConf] \qquad \frac{\mathscr{T} \xrightarrow{y!(b)} \mathscr{T}' \quad \Gamma \vdash \mathscr{T}' \bowtie LP}{\Gamma \vdash \mathscr{T} \bowtie y!:b.LP}\ [OutConf]$$

Rule $[InpConf]$ states that a modified type $\mathscr{T}$ is conformant with protocol $x?:b.LP$, if $\mathscr{T}$ can input a value of type b on port x and if continuation $\mathscr{T}'$ is conformant with the continuation of protocol LP. Rule $[OutConf]$ is similar but deals with the output of a value on port y.

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

Definition 4.3. *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* 4.1.
2. *If $K = [\tilde{x} > \tilde{y}]\{G; \mathsf{r}_1 = K_1, \dots, \mathsf{r}_\mathsf{k} = K_k; D; \mathsf{r}_1[F]\}$ then $K \Downarrow T$ when*
   - $\exists T_{r_i} \mid K_i \Downarrow T_{r_i}$,*for* $i = 1, 2, \dots, k$;
   - *T is extracted type from T_1 and $G \lfloor_{r_1}$ by Definition* 4.2*;*
   - $\mathscr{T}(F, T_{r_i}) \bowtie G \lfloor_{r_i}$ *for* $i = 1, 2, \dots, k$;

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

We can now our type safety results given in terms of Subject Reduction and Progress, 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 4.1 (Subject Reduction)**.** *If $K \Downarrow T$ and $K \xrightarrow{\lambda(v)} K'$ and v has type b then $T \xrightarrow{\lambda(b)} T'$ and $K' \Downarrow T'$.*

Proof. By induction on the derivation of $K \xrightarrow{\lambda(v)} K'$ (see Appendix F). □

Theorem 4.1 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 4.1 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 Progress 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 Progress 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 4.2 (Progress)**.** *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. By induction on the structure of K (see Appendix G). □

Theorem 4.2 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 4.2 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 4.1, to attest that types faithfully capture component behaviour.

5 Concluding Remarks

In this paper we introduce a type language for the choice-free subset of the GC language [6] that characterises the reactive behaviour of components and allows to capture "what components can do". In particular, 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 Progress theorems. Typing descriptions such as ours are crucial to promote component reusability, since to use a component we should only need to analyse its type and not its implementation (like in [6]). For instance, for the sake of ensuring the behaviour of a component is compatible with a governing communication protocol, where such compatibility is attested in our case by the conformance of the type to the (local) protocol.

We place our approach in the behavioural types setting (cf. [9]) since our types evolve in order to explain component behaviour (cf. Theorem 4.1), in contrast with classic subject reduction results where the type is preserved. In the realm of behavioural types, we distinguish Multiparty Asynchronous Session Types [8] which actually lay the basis for the protocol language of our target model [6]. 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 [5, 12].

We discuss some closely related work, starting by Open Multiparty Sessions [4] which to some extent shares the same goals and the same background (cf. [8]). The approach in [4] 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 [1]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 [3]), where however an assembly of services cannot be provided as a unit of reuse (like our composite components). We distinguish our type-based approach that aims at abstracting from the implementation and providing more general support for component substitution and reuse.

We believe the ideas reported in this paper can contribute to the theoretical basis for providing support for component-based development in distributed systems. Immediate directions for future work include the support for protocols with branching, and providing a characterisation of the substitution principle [10] based in our types. Further challenges remain at the level of conveying the theoretical model to concrete applications, in particular regarding component deployment and the support for their persistent reuse.

References

[1] CHOReVOLUTION project. `http://www.chorevolution.eu`.

[2] Amazon Web Services, Inc. Aws lambda: Developer guide, 2019. URL `https://docs.aws.amazon.com/en_pv/lambda/latest/dg/lambda-dg.pdf#welcome`.

[3] M. Autili, A. D. Salle, F. Gallo, C. Pompilio, and M. Tivoli. CHOReVOLUTION: Automating the Realization of Highly–Collaborative Distributed Applications. In H. R. Nielson and E. Tuosto, editors, *21th International Conference on Coordination Languages and Models (COORDINATION)*, volume LNCS-11533 of *Coordination Models and Languages*, pages 92–108, Kongens Lyngby, Denmark, June 2019. Springer International Publishing. doi: 10.1007/978-3-030-22397-7_6. URL `https://hal.inria.fr/hal-02365501`. Part 2: Tools (1).

[4] F. Barbanera and M. Dezani-Ciancaglini. Open multiparty sessions. In M. Bartoletti, L. Henrio, A. Mavridou, and A. Scalas, editors, *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, volume 304 of *EPTCS*, pages 77–96, 2019. doi: 10.4204/EPTCS.304.6. URL `https://doi.org/10.4204/EPTCS.304.6`.

[5] M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi: 10.1145/2429069.2429101. URL `https://doi.org/10.1145/2429069.2429101`.

[6] M. Carbone, F. Montesi, and H. T. Vieira. Choreographies for reactive programming. *CoRR*, abs/1801.08107, 2018. URL `http://arxiv.org/abs/1801.08107`.

[7] N. D. S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, today, and tomorrow. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017. URL: `https://doi.org/10.1007/978-3-319-67425-4_12,doi:10.1007/978-3-319-67425-4_12`.

[8] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1): 9:1–9:67, 2016. doi: 10.1145/2827695. URL `https://doi.org/10.1145/2827695`.

[9] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi: 10.1145/2873052. URL `https://doi.org/10.1145/2873052`.

[10] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16 (6):1811–1841, 1994. doi: 10.1145/197320.197383. URL `https://doi.org/10.1145/197320.197383`.

[11] M. D. Mcllroy. Mass produced software components. In *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, pages 138–155. Garmisch, 1969.

[12] F. Montesi. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, 2013. URL `http://fabriziomontesi.com/files/choreographic_programming.pdf`.

[13] 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/`.

[14] 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/`.

A Auxiliary definitions for Components

As shown in definitions for the extraction of a components (Definition 4.1 and Definition 4.2) both base and composite, the extraction of a set of constraints is obtained in such a way that the constraint for each output port is obtained separately. For this reason we first introduce the next definition, that is used in propositions required for proving the theorems.

Definition A.1. *We define the restriction of a component K to one output port denoted by $K(y)$ as*

1. *If $K = [\tilde{x}\rangle\,\tilde{y}]\{L\}$, then $K(y) = [\tilde{x}\rangle\, y]\{L\}$, where $y \in \tilde{y}$.*
2. *If $K = [\tilde{x}\rangle\,\tilde{y}]\{G;R;D;\mathsf{r}[F]\}$, then $K(y) = [\tilde{x}\rangle\, y]\{G;R;D;\mathsf{r}[F]\}$, where $y \in \tilde{y}$.*

In the Definition 4.1 we mainly focus on the local binders. For this reason we introduce some of their properties that are used in proofs.

Proposition A.1. *If $[\tilde{x}\rangle\,\tilde{y}]\{L\} \xrightarrow{x?v} [\tilde{x}\rangle\,\tilde{y}]\{L'\}$ then:*

1. *If $L = f(\tilde{x}) < \tilde{\sigma}, L_1 \wedge x \in \tilde{x}$. Then we have that $L' = f(\tilde{x}) < \tilde{\sigma}', L_1'$ where $count(x,\tilde{\sigma}') = count(x,\tilde{\sigma}) + 1$;*
2. *If $L = f(\tilde{x}) < \tilde{\sigma}, L_1 \ \wedge\ x \notin \tilde{x}$. Then we have that $L' = f(\tilde{x}) < \tilde{\sigma}, L_1'$.*

Proof. Proof by induction on the derivation of $L \xrightarrow{x?v} L'$.

[LInpDisc] $y = f(\tilde{x}) < \tilde{\sigma} \xrightarrow{\mathrm{x?v}} y = f(\tilde{x}) < \tilde{\sigma}$, by inversion we know that $x \notin \tilde{x}$ so the property holds.

[LInpNew] $y = f(\tilde{x}) < \tilde{\sigma} \xrightarrow{\mathrm{x?v}} y = f(\tilde{x}) < \tilde{\sigma}, \{x \to v\}$, by inversion we know that $x \in \cap_{\sigma_i \in \tilde{\sigma}}\ dom(\sigma_i)$, $i \in \{1,2,\ldots,n\}$. After input on x, the number of mappings for x increases by 1. So, the property holds.

[LInpUpd] $y = f(\tilde{x}) < \tilde{\sigma}_1, \sigma, \tilde{\sigma}_2 \xrightarrow{\mathrm{x?v}} y = f(\tilde{x}) < \tilde{\sigma}_1, \sigma[x \to v], \sigma_2$. 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 increases by 1. So, the property holds.

[LInpList] $L_1, L_2 \xrightarrow{\mathrm{x?v}} L_1', L_2'$. By inversion we know that $L1 \xrightarrow{\mathrm{x?v}} L_1'$, and by i.h. we know that for L_1' the property holds. The same reasoning for $L_2 \xrightarrow{\mathrm{x?v}} L_2'$, so also for L_2' the property holds. Directly we can conclude that the property holds also for L_1', L_2'.

□

Proposition A.2. *If $L \xrightarrow{y!v} L'$ then:*

1. *If $L = y = f(\tilde{x}) < \tilde{\sigma}, L_1 \Rightarrow L' = y = f(\tilde{x}) < \tilde{\sigma}', L_1$ where $\forall x \in \tilde{x} \mid count(x,\tilde{\sigma}') = count(x,\tilde{\sigma}) - 1$ and $count(x,\tilde{\sigma}) \geq 1$;*
2. *If $L = y = f() < \cdot, L_1 \Rightarrow L' = y = f() < \cdot, L_1$.*

Proof. Proof by induction on the derivation of $L \xrightarrow{y!v} L'$.

[LOut] $y = f(\tilde{x}) < \sigma, \tilde{\sigma} \xrightarrow{\mathrm{y!v}} y = f(\tilde{x}) < \tilde{\sigma}$, so $f(\tilde{x})$. By inversion we have that $\{\tilde{x}\}$=$\mathsf{dom}(\sigma)$ and $f(\sigma(\tilde{x})) \downarrow v$. We can directly conclude, from the definition of function *count* that the property holds.

[LConst] $f() < \cdot \xrightarrow{\mathrm{y!v}} f() < \cdot$. We directly conclude that the property holds.

[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'. So, the property holds for L_1', L_2, where $L = L_1, L_2$. □

B Auxiliary definitions for Composite component Type extraction

The set of ports that are found in the description of the body a recursion of some local protocol LP is defined as follows:

$$rep(LP) \stackrel{\triangle}{=} \{z | LP = \mathscr{C}[\mu\mathbf{X}.\mathscr{C}'[z?:b.LP']] \vee LP = \mathscr{C}[\mu\mathbf{X}.\mathscr{C}'[z!:b.LP']]\}$$

The set of ports that are found in the description of some local protocol LP is defined as follows:

$$fp(LP) \stackrel{\triangle}{=} \{z | LP = \mathscr{C}[z?:b.LP'] \vee LP = \mathscr{C}[z!:b.LP'] \vee LP = \mathscr{C}[\mu\mathbf{X}.\mathscr{C}'[z?:b.LP']] \vee LP = \mathscr{C}[\mu\mathbf{X}.\mathscr{C}'[z!:b.LP']]\}$$

For the purpose of making proofs less complex, we introduce the notion of **inc** given by the next definition, that can be the conclusion of observation of our type language semantics when we have the value input.

Definition B.1. *By* $\boldsymbol{inc}(\boldsymbol{C},x)$ *we denote the constraint defined as follows:*

$$\begin{array}{lll} \boldsymbol{inc}(\{y(b'):\boldsymbol{B}:[\{x:N\} \uplus \boldsymbol{D}]\},x) & \triangleq & \{y(b'):\boldsymbol{B}:[\{x:N+1\} \uplus \boldsymbol{D}]\} \\ \boldsymbol{inc}(\{y(b'):\boldsymbol{B}:[\{x:\Omega\} \uplus \boldsymbol{D}]\},x) & \triangleq & \{y(b'):\boldsymbol{B}:[\boldsymbol{D}]\} \\ \boldsymbol{inc}(\{y(b'):\boldsymbol{B}:[\boldsymbol{D}]\},x) & \triangleq & \{y(b'):\boldsymbol{B}:[\boldsymbol{D}]\} \text{ (if } x \notin dom(\boldsymbol{D})\text{).} \end{array}$$

Proposition B.1. *Let* $[\tilde{x}\rangle\, \tilde{y}]\{G; r = K, R; D; \mathsf{r}[F]\}$ *be the composite component,* $T_r = <X_b, \boldsymbol{C}>$ *the type of interfacing component K and LP its local protocol. If* $\boldsymbol{D}_d(\boldsymbol{C},F,y) \uplus \boldsymbol{D}_t(\boldsymbol{C},F,LP,y) = \{x:N^1, x:N^2, x:N^k\} \uplus \overline{\boldsymbol{D}}$, *(where* $\neg\exists N$ *such that* $x:N \in \overline{\boldsymbol{D}}$*) then* $N^1 = N^2 = ... = N^k$.

Proof. The proof is divided in two cases: first one is when we assume that one of the dependencies is obtained in a direct way, and the other one is when all of them are obtained in a transitive way.

Case 1. Lets assume that $x:N^i \in \mathbf{D}_d(\mathbf{C},F,y)$ and $x:N^j \in \mathbf{D}_t(\mathbf{C},F,LP,y)$, where $i,j \in \{1,2,...,k\}$ and $i \neq j$.

Consider the scenario where $\mathbf{C} = \{y(b_1):\mathbf{B}:[\{x':N_2, x:N\} \uplus \mathbf{D}'], y'(b_2):\mathbf{B}':[\{x:N_1\} \uplus \mathbf{D}]\} \uplus \mathbf{C}'$, where $x \in F^i$, $y \in F^o$ and $x', y' \in fp(LP)$ and $y' \diamond_3^{LP} x'$. Then

$$N = N_1 + N_2 + \mathbf{vf}(LP, x', y')$$

where $N = N^i$ and $N^j = N^1 + N^2 + \mathbf{vf}(LP, x', y')$

We prove this case by the number values emitted from port y', denoted by n.

Base case. n=0. Since $n = 0$, $\mathbf{vf}(LP, x', y') = 0$. Moreover, by the description of relation $\diamond_3^{LP}$ we know that an input on x' is preceded by an output on y', so $N_2 = 0$.

Note that since $N_2 = 0$, the value cannot be output from port y since the dependencies are not satisfied.

Both ports y and y' have a per each value dependency on x and from non of them any value is output, so $N = N_1$. Then we have:

$$N = N_1 = N_1 + 0 + 0 = N_1 + N_2 + \mathbf{vf}(LP, x', y')$$

Induction hypothesis. Assume that the property holds for $n = k$.

Inductive step. We prove that the property holds for $n = k+1$.

If the value can be sent from port y', $k+1$-times, by Rule [T6] we know that the number of values input on x is $\geq k+1 = k+1+m$, $m > 0$. Assuming that there was no value output from y, by Rule [T3] we know that $N = k+1+m$.

Since the value is emitted from port y', $k+1$-times, then by Rule [T6] we have that $N_1 = k+1+m-(k+1) = m$. After an output from y' we have that $\mathbf{vf}(LP,x',y') = 1$

Moreover, since the value was already emitted k-times and it can be output one more time, thus the recursive protocol was unfolded k times. This implies that the value is received on port x', k-times. This implies that, since there was no output from y, by Rule [T3] $N_2 = k$.

So, we have that $N = k+1+m$, $N_1 = m$, $\mathbf{vf}(LP,x',y') = 1$ and $N_2 = k$, we conclude that

$$N = N_1 + N_2 + \mathbf{vf}(LP,x',y')$$

If now we have some number q of values emitted from y, (by Rule [T6] $q \leq k$) then by Rule [T6] we have that $N = k+m+1-q$ and $N_2 = k-q$. Again we have that $N = k+m+1-q = k-q+m+1 = N_2 + N_1 + \mathbf{vf}(LP,x',y')$. Since this case holds for arbitrary per each value obtained in a transitive way, we conclude that it holds for all of them (there is only one $x : N$ obtained in a direct way).

Case 2. Assuming that $\forall x : N^i$ where $i \in \{1,2,...,k\}$ holds that $x : N^i \in \mathbf{D}_t(\mathbf{C}, F, LP, y)$. Without loss of generality we prove that the property holds for two of them, i.e. $N^i = N^j$ where $i, j \in \{1,2,...,k\}$ and $i \neq j$. Since this case holds for arbitrary per each value obtained in a transitive way, we conclude that it holds for all of them.

We divide the proof for Case 2. into three subcases: the first one is when both transitive dependencies ($x{:}N^i$ and $x{:}N^j$) are obtained via the same output port attached to the protocol, but different input ports; the second one is when they are obtained via the same input port attached to the protocol, but different output ports; and the third one is that they are obtained via different ports attached to the protocol.

1. Consider the scenario where $\mathbf{C} = \{y(b) : \mathbf{B} : [\{x'_i{:}N'_i, x'_j{:}N'_j\} \uplus \mathbf{D}'], y'(b') : \mathbf{B}' : [\{x{:}N\} \uplus \mathbf{D}]\} \uplus \mathbf{C}'$, where $x \in F^i$, $y \in F^o$ and $x'_i, x'_j, y' \in fp(LP)$ and $y' \diamond_3^{LP} x'_i$ and $y' \diamond_3^{LP} x'_j$. Then

   $$N^i = N^j$$

   where $N^i = N'_i + N + \mathbf{vf}(LP, x'_i, y')$ and $N^j = N'_j + N + \mathbf{vf}(LP, x'_j, y')$

   Assume that in the description of *LP* an input on x_i precedes an input on x_j (the proof is the same for the case of the opposite order).

   We prove this case, again, by the number values emitted from port y', denoted by n.

   Base case. n=0. Since $n = 0$, $\mathbf{vf}(LP, x'_i, y') = 0$ and $\mathbf{vf}(LP, x'_j, y') = 0$. Moreover, by the description of relation $\diamond_3^{LP}$ we know that an input on x'_i and x'_j is preceded by an output on y', so $N'_i = 0 = N'_j$. So, $N^i = N + 0 + 0 = N^j$.

   Induction hypothesis. Assume that the property holds for n=k.

   Inductive step. $n = k+1$

   If the value can be sent from port y', $k+1$-times, by Rule [T6] we know that the number of values input on x is $\geq k+1 = k+1+m$, $m > 0$.

Since the value is emitted from port y', $k+1$-times, then by Rule [T6] we have that $N_1 = k+1+m-(k+1) = m$. After an output from y' we have that $\mathbf{vf}(LP, x'_i, y') = \mathbf{vf}(LP, x'_j, y') = 1$

Moreover, since the value was already emitted k-times and it can be output one more time, thus the recursive protocol was unfolded k times. This implies that the value is received on ports x'_i and x'_j, k-times. This implies that, asumming that there was no output from y, by Rule [T3] $N'_i = N'_j = k$.

So, we have that $N'_i = N'_j = k$, $N = m$, $\mathbf{vf}(LP, x'_i, y') = \mathbf{vf}(LP, x'_j, y') = 1$, we conclude that

$$N^i = N^j$$

Since an input on x'_j precedes an input on x'_j, consider the case when the $k+1$ value arrives on port x'_i. Then $N_i = k+1$ but $\mathbf{vf}(LP, x'_i, y') = 0$. So we have that $N^i = m+k+1$ and $N^j = m+(k+1)+0$ that again implies that $N^i = N^j$.

2. Consider the scenario where $\mathbf{C} = \{y(b) : \mathbf{B} : [\{x' : N'\} \uplus \mathbf{D}'], y'_i(b_i) : \mathbf{B}'_i : [\{x : N_i \uplus \mathbf{D}''], y'_j(b_j) : \mathbf{B}'_j : [\{x : N_j\} \uplus \mathbf{D}]\} \uplus \mathbf{C}'$, where $x \in F^i$, $y \in F^o$ and $x', y'_i, y'_j \in fp(LP)$ and $y'_i \diamond_3^{LP} x'$ and $y'_j \diamond_3^{LP} x'$. Then

$$N^i = N^j$$

where $N^i = N_i + N' + \mathbf{vf}(LP, x', y'_i)$ and $N^j = N_j + N' + \mathbf{vf}(LP, x', y'_j)$.

We prove this case, by the number values emitted from port y'_i, denoted by n.

Assume that in the description of LP an output on y'_i precedes an output on y'_j (the proof is the same for the opposite order and if we consider the number of values emmited from port y'_j).

Base case. n=0. Since $n = 0$, and an output on y'_i precedes an output on y'_j we have $\mathbf{vf}(LP, x', y'_i) = 0$ and $\mathbf{vf}(LP, x', y'_j) = 0$. Moreover, by the description of relation $\diamond_3^{LP}$ we know that an input on x' is preceded by an output on y'_i and y'_j, so $N'_i = 0 = N'_j$. So, $N^i = N + 0 + 0 = N^j$.

Induction hypothesis. Assume that the property holds for n=k.

Inductive step. $n = k+1$

If the value can be sent from port y'_i, $k+1$-times, by Rule [T6] we know that the number of values input on x is $\geq k+1 = k+1+m$, $m > 0$, for that is captured in the dependencies for both y'_i and y'_j.

Since the value is emitted from port y', $k+1$-times, applying Rule [T6], we have $N_i = k+1+m-(k+1) = m$. After an output from y'_i we have that $\mathbf{vf}(LP, x', y'_i) = 1$, but $\mathbf{vf}(LP, x', y'_j) = 0$

Moreover, since the value was already emitted k-times and it can be output one more time, thus the recursive protocol was unfolded k times. This implies that k values are emitted from port y'_j, hence $N_i = k+1+m-(k) = m+1$. Moreover, the value is received on port x', k-times. This implies that, assuming that there was no output from y, by Rule [T3] that $N' = k$.

So, we have that $N' = k$, $N_i = m$, $N_j = m+1$, $\mathbf{vf}(LP, x', y'_i) = 0$, and $\mathbf{vf}(LP, x', y'_j) = 1$ we conclude that

$$N^i = N^j$$

Since an output from y'_i precedes an output from y'_j, consider the case when the $k+1$ value is emitted from port y'_j. Then $N_j = m$. but $\mathbf{vf}(LP, x', y'_j) = 1$. So we have that $N^i = m+k+1$ and $N^j = m+k+1$ that again implies that $N^i = N^j$.

With a $k+1$ input of a value on port x', we have that number N' by Rule [T3] increases by one, but there is no value flowing from ports y'_i and y'_j. So, $N^i = m+k+1+0$ and $N^j = m+k+1+0$, keeping $N^i = N^j$.

After an output on y for both 1) and 2) the equality will remain (similar reasoning as for the previous case).

3. The proof for the case where the two transitive per each value dependencies are obtained via different input and output ports of the protocol is a combination of the previous cases 1) and 2).

□

Definition B.2. *Let $\boldsymbol{W}$ be the set of tuples of the form $x:M$. Let $\boldsymbol{pr}$: $\boldsymbol{W} \to \boldsymbol{W}$ be the function ("Priority function") defined as follows:*

$$
\begin{array}{lcl}
\boldsymbol{pr}(\emptyset) & \triangleq & \emptyset \\
\\
\boldsymbol{pr}(\{x:M\} \cup \boldsymbol{W}) & \triangleq & \{x:M\} \cup \boldsymbol{pr}(\boldsymbol{W}) \\
 & & \textit{if } x \notin \boldsymbol{elements}(\boldsymbol{W}) \\
\boldsymbol{pr}(\{x:M\} \cup \boldsymbol{W}) & \triangleq & \{x:M\} \cup \boldsymbol{pr}(\boldsymbol{W}) \setminus \{x:M'\} \\
 & & \textit{if } x:M' \in \boldsymbol{pr}(\boldsymbol{W}) \textit{ and } M = N \textit{ and } M' = \Omega \\
\boldsymbol{pr}(\{x:M\} \cup \boldsymbol{W}) & \triangleq & \boldsymbol{pr}(\boldsymbol{W}) \\
 & & \textit{if } x:M' \in \boldsymbol{pr}(\boldsymbol{W}) \textit{ and } M' = N' \textit{ and } M = \Omega
\end{array}
$$

where $x \in \boldsymbol{elements}(\boldsymbol{W})$ if $x:M \in \boldsymbol{W}$ for some M.

Extracting the type of a composite component we use the port identifiers of the interfacing component. At the end of the extraction procedure we need to rename those ports to the ones of the interface of a composite component. The operation that allows us to achieve that is operation **ren**(,) defined as follows:

$$
\begin{array}{lcll}
\mathbf{ren}(F, < X_b; \mathbf{C} >) & \triangleq & < \mathbf{ren}(F, X_b); \mathbf{ren}(F, \mathbf{C}) > & \\
\mathbf{ren}(F, \{x_1(b_1)\} \uplus X_b) & \triangleq & \mathbf{ren}(F, \{x_1(b_1)\}) \uplus \mathbf{ren}(F, X_b) & \\
\mathbf{ren}(F, \{y(b) : \mathbf{B} : [\mathbf{D}]\} \uplus \mathbf{C}) & \triangleq & \mathbf{ren}(F, \{y(b) : \mathbf{B} : [\mathbf{D}]\}) \uplus \mathbf{ren}(F, \mathbf{C}) & \\
\mathbf{ren}(F, \{y(b) : \mathbf{B} : [\mathbf{D}]\}) & \triangleq & \{\mathbf{ren}(F, y(b)) : \mathbf{B} : \mathbf{ren}(F, \mathbf{D})\} & \\
\mathbf{ren}(F, \{x:M\} \uplus \mathbf{D}) & \triangleq & \mathbf{ren}(F, \{x:M\}) \uplus \mathbf{ren}(F, \mathbf{D}) & \\
\mathbf{ren}(F, \{x:M\}) & \triangleq & \{\mathbf{ren}(F, x) : M\} & \\
\mathbf{ren}(F, x) & \triangleq & x' & \text{if } F = x \leftarrow x', F' \\
\mathbf{ren}(F, x(b)) & \triangleq & x'(b) & \text{if } F = x \leftarrow x', F' \\
\mathbf{ren}(F, y(b)) & \triangleq & y'(b) & \text{if } F = y' \leftarrow y, F' \\
\mathbf{ren}(\emptyset) & \triangleq & \emptyset &
\end{array}
$$

C Auxiliary results for Types

Proposition C.1. *Let $T =< X_b; \boldsymbol{C} >$. If $x(b^x) \in X_b$ for some b^x, then $T \xrightarrow{x(b^x)} T'$.*

Proof. Since Rules [T2] and [T3] are the axioms and Rule [T1] captures the case where x is not even in the domain of the dependencies of some constraint, we can conclude that the property holds. □

Lemma C.1. *If $T =< X_b; \{\boldsymbol{C}_i | i = 1, 2, \ldots, k\} >$ and $T \xrightarrow{x?(b)} T'$ then $T' =< X_b; \{\boldsymbol{inc}(\{\boldsymbol{C}_i, x\}) | i = 1, 2, \ldots, k\} >$.*

Proof. Following Rule [T5] we have that:

$< \{x(b) \uplus X_b\}; \{y_i(b_i) : \mathbf{B}_i : [\mathbf{D}_i] | i \in 1, \ldots, k\} > \xrightarrow{x?(b)} < \{x(b) \uplus X_b\}; \{y_i(b_i) : \mathbf{B}_i : [\mathbf{D}'_i] | i \in 1, \ldots, k\} >$.

By inversion we know that $\forall i \in 1, 2, \ldots, k : \quad y_i(b_i) : \mathbf{B}_i : [\mathbf{D}_i] \xrightarrow{x?} y_i(b_i) : \mathbf{B}_i : [\mathbf{D}'_i]$. Inspecting rules [T1],[T2] and [T3] we can conclude that the property holds.

□

Proposition C.2. *[Dependencies requirement] Let* $T =< X_b; \boldsymbol{C}_1, \boldsymbol{C}_2, \ldots, \boldsymbol{C}_k >$ *and* $\boldsymbol{C}_i = \{y_i(b_i) : \boldsymbol{B}_i : [\boldsymbol{D}_i]\} \uplus \boldsymbol{C}'$*. Then if* $T \xrightarrow{y_i(b_i)} T' \Rightarrow \boldsymbol{B}_i > 0 \wedge \forall x \in dom(\boldsymbol{D}_i) : \boldsymbol{D}_i = \{x : N\} \uplus \boldsymbol{D}'_i \wedge N > 0$.

Proof. We can directly prove this proposition from Rule [T6]. □

Proposition C.3. *[Constraint independency] Let K be a component with interface* $[\tilde{x} > y_1, \ldots, y_k]$ *and* $T =< X_b; \boldsymbol{C}_1 \uplus \cdots \uplus \boldsymbol{C}_k\}$ *be the type of K. Then for every* y_i*, where* $i \in \{1, \ldots, k\}$ *holds that* $K(y_i) \Downarrow < X_b; \boldsymbol{C}_i >$*, where* $\boldsymbol{C}_i = \{y_i(b^{y_i}) : \boldsymbol{B}^{y_i} : [\boldsymbol{D}^{y_i}]\}$.

Proof. Reffering to Definition 4.1 and Definition 4.2 the proof is direct. □

D Modified type $\mathscr{T}$

Now we introduce the modified type denoted by $\mathscr{T}$. 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 $\mathscr{T}(F, T)$. 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 $\mathscr{T}$-type is given in the Table 6. It is similar to a 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 7).

$\mathscr{T}$-Type syntax

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

Table 6: $\mathscr{T}$-Type syntax

$\mathscr{T}$-Type semantics

Definition D.1. $\mathscr{T}' \leq \mathscr{T}$ *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* $\mathscr{T}' \xrightarrow{x_1?(b_1)} \cdots \xrightarrow{x_k?(b_k)} \mathscr{T}$.

$$\frac{x \notin dom[\mathscr{D}]}{y(b):\mathbf{B}:[\mathscr{D}] \xrightarrow{x?} y(b):\mathbf{B}:[\mathscr{D}]}\ [\mathscr{T}1] \qquad \frac{}{y(b'):\mathbf{B}:[\{x:\Omega\}\uplus\mathscr{D}] \xrightarrow{x?} y(b'):\mathbf{B}:[\mathscr{D}]}\ [\mathscr{T}2]$$

$$\frac{}{y(b'):\mathbf{B}:[\{x{:}\mathscr{N}\}\uplus\mathscr{D}] \xrightarrow{x?} y(b'):\mathbf{B}:[\{x{:}\mathscr{N}+1\}\uplus\mathscr{D}]}\ [\mathscr{T}3] \qquad \frac{}{\mathscr{T} \xrightarrow{\tau} \mathscr{T}}\ [\mathscr{T}4]$$

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

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

Table 7: $\mathscr{T}$ Semantics

Definition D.2. *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 $\mathscr{T}(F,T_r)$ is the T_r-modified type where:*

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

Note that for $K=[\tilde{x}>\tilde{y}]\{G,r_1=K_1,r_2=K_2,\dots,r_n=K_n;D;r_1[F]\}$ we have that $\mathscr{T}(F,T_{r_2})=T_{r_2},\dots,\mathscr{T}(F,T_{r_k})=T_{r_k}$, since the only component that forwards the values from/to external environment is component K_1.

E Conformance relation

$$\frac{\mathscr{T} \xrightarrow{x?(b)} \mathscr{T}' \quad \Gamma\vdash\mathscr{T}'\bowtie LP}{\Gamma\vdash\mathscr{T}\bowtie x?{:}b.LP}\ [InpConf] \qquad \frac{\mathscr{T} \xrightarrow{y!(b)} \mathscr{T}' \quad \Gamma\vdash\mathscr{T}'\bowtie LP}{\Gamma\vdash\mathscr{T}\bowtie y!{:}b.LP}\ [OutConf]$$

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

$$\frac{\Gamma,X:\mathscr{T}\vdash\mathscr{T}\bowtie LP}{\Gamma\vdash\mathscr{T}\bowtie recX.LP}\ [RecConf]$$

Table 8: Conformance

Rule $[InpConf]$ ensures that a modified type $\mathscr{T}$ 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 values 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 $\mathscr{T}'\leq\mathscr{T}$ (Definition $D.1$). By Lemma $C.1$ the possible difference between types $\mathscr{T}'$ and $\mathscr{T}$ 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 $\mathscr{T}$ is conformant with a protocol $rec\mathbf{X}.LP$, provided that the type is conformant with the body of the recursion under the environment extended with assumption $\mathbf{X} : \mathscr{T}$.

F Subject Reduction Proof

We now prove the Subject Reduction theorem (Theorem 4.1). In reminder:

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

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] If $K = [\tilde{x} > \tilde{y}]\{L\} \xrightarrow{\text{x?v}} [\tilde{x} > \tilde{y}]\{L'\} = K'$, by inversion we know that $x \in \tilde{x}$ and $L \xrightarrow{x?v} L'$, so Proposition $A.1$ holds. Since $x \in \tilde{x}$, by the definition of the type extraction we know that $x(b^x) \in X_b$, where $T =< X_b; \mathbf{C} >$, for some b^x such that $\gamma(v) = b^x$. Since $x(b^x) \in X_b$, by Proposition $C.1$ we know that T evolves in $T \xrightarrow{x?(b^x)} T'$, for some T'. By Lemma $C.1$ and Definition 4.1 we conclude that $K' \Downarrow T'$.

[OutBase] If $K = [\tilde{x} > \tilde{y}]\{L\} \xrightarrow{\text{y!v}} [\tilde{x} > \tilde{y}]\{L'\} = K'$, by inversion we know that $y \in \tilde{y}$ and $L \xrightarrow{y!v} L'$. Since $y \in \tilde{y}$, by the definition of the type extraction we know that $y(b^y) : \mathbf{B} : [\mathbf{D}] \in \mathbf{C}$, where $T = \{X_b; \mathbf{C}\}$, for some $b^y, \mathbf{B}, \mathbf{D}$ where $\gamma(v) = b^y$. Since $[\tilde{x} > \tilde{y}]\{L\} \xrightarrow{\text{y!v}} [\tilde{x} > \tilde{y}]\{L'\}$ we know that then Proposition $A.2$ holds, so by the type extraction we conclude that also that for each $x : N \in \mathbf{D}$ holds that $N > 0$. Since K is a base component $\mathbf{B} = \infty$. By Rule $[T6]$ we know that then $T \xrightarrow{y!(b^y)} T'$, for some T'. By Proposition $A.2$, by Rule $[T6]$ and by Definition 4.1 we conclude that $K' \Downarrow T'$.

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 Proposition $C.1$ 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 $C.3$, 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. $\mathbf{C} = \{y(b^y) : \mathbf{B} : [\mathbf{D}]\} \uplus \mathbf{C}' \wedge x \notin dom(\mathbf{D})$ 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:
$y(b^y) : \mathbf{B} : [\mathbf{D}] \xrightarrow{x?} y(b^y) : \mathbf{B} : [\mathbf{D}]$.
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:
$\overline{y}(b^y) : \mathbf{B}_r : [\{z : N\} \uplus \mathbf{D}_r'] \xrightarrow{z?} \overline{y}(b^y) : \mathbf{B}_r : [\{z : N+1\} \uplus \mathbf{D}_r']$.
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 $N' + N'' + \mathbf{vf}(LP, z', y')$. So, $N = N' + N'' + \mathbf{vf}(LP, z', y')$.
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 $\mathbf{vf}(LP, z', y')$ 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 $\overline{N} = N' + N'' + 1 + \mathbf{vf}(LP, z', y')$. This number can be written as

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

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

Number $\mathbf{vf}(LP, z', y')$ 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] $K = [\tilde{x} > \tilde{y}]\{G; r = \overline{K}, R; D; r[F]\} \xrightarrow{y(v)} [\tilde{x} > \tilde{y}]\{G; r = \overline{K'}, R; D; r[F]\} = K'$, 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 4.2, 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 $\{z_1 : N_1, \ldots, z_k : N_k\} = ren(F, \{x_1 : N_1, \ldots, x_k : N_k\})$ 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 4.2 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 $y', z' \in fp(LP)$ such that $y' \diamond_3^{LP} z'$ (recap: recursive protocol, where both y' and z' are in $rep(LP)$ 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 $N^{z_i} = (N' - 1) + N_i^{y'} + \mathbf{vf}(LP, x', y')$.
Since $T =< X_b; \{y(b^y) : \mathbf{B} - 1 : [\{x_1 : N_1, \ldots, x_k : N_k\} \uplus \mathbf{D}]\} \uplus \mathbf{C}' >$ and by the type extracting procedure we know that $N_i = N' + N_i^{y'} + \mathbf{vf}(LP, z', y')$. By Rule [T6] we have that $T \xrightarrow{y!(b^y)} < X_b; \{y(b^y) : \mathbf{B} - 1 : [\{x_1 : N_1 - 1, \ldots, x_i : N_i - 1, \ldots, x_k : N_k - 1\} \uplus \mathbf{D}]\} \uplus \mathbf{C}' >= T'$.
This implies that the number of values from port x_i available for y ($i = 1, \ldots, k$) in type T' is $N_i - 1 = N' - 1 + N_i^{y'} - + \mathbf{vf}(LP, z', y') = N^{z_i}$.
Note that $\mathbf{vf}(LP, z', y')$ 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 $D.2$, since K_r is the only interfacing component, then $\mathscr{T}(F, T_i) = T_i$, where $i = 1, \ldots, m$.

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

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 $T_r =< Z_b; \mathbf{C}_r >$, $T_i =< Z_b^i; \mathbf{C}_i >$, where $i = 1, \ldots, m$. Since $z(b) \in Z_b$ and $z(b) \notin Z_b^i, \forall i = 1, \ldots, m$, by Rule [$\mathscr{T}5$] we have that:
$\mathscr{T}(F, T_r) \xrightarrow{z?(b)} \mathscr{T}'(F, T_r), T_1 \xrightarrow{z?(b)} T_1, \ldots, T_m \xrightarrow{z?(b)} T_m$.
By Rule $[InpConf]$ we have that: $\mathscr{T}'(F, T_r) \bowtie G' \downharpoonleft_r, T_1 \bowtie G' \downharpoonleft_{r_1} \ldots, T_m \bowtie G' \downharpoonleft_{r_m}$.

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 $LP^1 = \mathscr{C}[\mu\mathbf{X}.\mathscr{C}'[y'! : t_b'.\mathscr{C}''[x'? : .LP']]]$). 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 $\bar{y}$ on z_1 (up to renaming), since $\mathbf{vf}(LP,z,y') = 1$. After an input on z, $\mathbf{vf}(LP,z,y') = 0$, but the dependency of $\bar{y}$ on z will be dropped, hence, by the type extraction procedure, there is no dependency of $\bar{y}$ 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 $\bar{y}$ in T (up to renaming) is $N+N_1+\mathbf{vf}(LP,z,y') = N+N_1+1$ ($\mathbf{vf}(LP,z,y') = 1$). After an input on z, the number of values of z available for $\bar{y}$ (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 $\bar{y}$ is $N+1+N_1+\mathbf{vf}(LP,z,y') = N+1+N_1+0$ ($\mathbf{vf}(LP,z,y') = 0$ 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 $LP = y'!:b.LP'$;
- $\mathscr{T}(F,T_i) = T_i$, where $i = 1,\ldots,m$;
- $\mathscr{T}(F,T_r) \bowtie z?:b.LP',\ T_1 \bowtie G\downharpoonright_{r_1},\ \ldots,\ T_m \bowtie G\downharpoonright_{r_m}$;
- $G \xrightarrow{r?l<v>} G'$ then $G\downarrow_i = G'\downarrow_i,\ \ i = 1,\ldots m$;
- Let $T_r = < Z_b; \mathbf{C}_r >$, $T_i = < Z^i_b; \mathbf{C}_i >$, where $i = 1,\ldots,m$;
- $\mathscr{T}(F,T_r) \xrightarrow{y'!(b)} \mathscr{T}'(F,T_r), T_1 \xrightarrow{y'!(b)} T_1, \ldots, T_m \xrightarrow{y'!(b)} T_m$;
- By Rule $[OutConf]$ we have that: $\mathscr{T}'(F,T_r) \bowtie G'\downharpoonright_r, T_1 \bowtie G'\downharpoonright_{r_1} \ldots, T_m \bowtie G'\downharpoonright_{r_m}$.
  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 $T_r = < Z_b; \{\bar{y}(b) : \mathbf{B}^{\bar{y}} : [\mathbf{D}^{\bar{y}}]\} \uplus \mathbf{C}^1_r >$.

Since T_r can do an output from port y', by Rule [T6] we know that $T'_r = < Z_b; \{y'(b) : \mathbf{B}^{y'} - 1 : [\mathbf{D}^{y'}_1]\} \uplus \mathbf{C}^1_r >$, where $\forall z{:}N \in \mathbf{D}^{y'} \Rightarrow z{:}N-1 \in \mathbf{D}^{y'}_1$.
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 $\mathbf{C}$ 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 $N + N' + 0$ ($\mathbf{vf}(LP, z', y') = 0$ before an output on y'). After an output on y', N will decrease by 1 (Rule [T6]), but $\mathbf{vf}(LP, z', y')$ will increase by 1 ($\mathbf{vf}(LP, z', y') = 1$). Hence, in the extracted type from T_r' and LP' the number of values of port z available for y is $N - 1 + N_1 + \mathbf{vf}(LP, z, y') = N - 1 + N_1 + 1$. Therefore, the dependencies of the output ports in type T' are the same as those in T. Therefore, $K \Downarrow T'$, where $T' = T$.

□

G Progress Proof

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

Lemma G.1. *[Protocol Progresss] 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 progress property:*

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

Then for any trace such that

$$G \xrightarrow{\mathsf{p}_1\ell_1(v_1)} \cdots \xrightarrow{\mathsf{p}_k\ell_k(v_k)} G'$$

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 $G \xrightarrow{\mathsf{p}\ell(v)} G'$, where $\mathsf{p} \in \{\mathsf{p}!, \mathsf{p}?\}$. 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, $\mathscr{T}(F, T_p) \xrightarrow{a(b)} \mathscr{T}(F, T_p')$, where $a = x?$ if $\mathsf{p} = \mathsf{p}?$, $a = y!$

if $\mathsf{p} = \mathsf{p}!$, and $\gamma(v) = b$, where x and y are ports explained in a distribution binder ($\mathbf{D}$). After protocol G evolved to G', by Rule $[InpConf]$ for $a = x?$ or Rule $[OutConf]$ for $a = y!$, we have that $\mathscr{T}(F, T_p') \bowtie G' \downharpoonright_p$, 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 progress property, so exists some value v of type b, and K_p' such that $K_p \xRightarrow{a(v)} K_p'$ and $K_p' \Downarrow T_p'$, K_p' also enjoys the progress property. Applying Rule[$\mathsf{Internal}$] some number of times and then Rule [$\mathsf{InpChor}$] (or [$\mathsf{OutChor}$] 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 4.1 then $T \xrightarrow{\tau} \cdots T$ and $K' \Downarrow T$.

Induction hypothesis. 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.,

$$G \xrightarrow{\mathsf{p}_1 \ell_1(v_1)} \cdots \xrightarrow{\mathsf{p}_{n-1} \ell_{n-1}(v_{n-1})} G^{n-1} \xrightarrow{\mathsf{p}_n \ell_n(v_n)} G'.$$

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 4.1 m times we know that exists T'' such that $T \xrightarrow{\tau} \cdots \xrightarrow{\tau} T''$ and $K'' \Downarrow T''$.

By inversion on rules [$\mathsf{InpChor}$], [$\mathsf{OutChor}$] or [$\mathsf{Internal}$] 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 4.1 (applied multiple times) exists T_i' such that $K_i' \Downarrow T_i'$. Each K_i' enjoys the progress property.

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

$$G \xrightarrow{\mathsf{p}_1 \ell_1(v_1)} \cdots \xrightarrow{\mathsf{p}_{n-1} \ell_{n-1}(v_{n-1})} G^{n-1} \xrightarrow{\mathsf{p}_n \ell_n(v_n)} G'.$$

exists K' such that

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

$\square$

Now we prove the Progress theorem (Theorem 4.2). In reminder:

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

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 progress property holds for all the subcomponents of K.
- Finally, we prove that the progress property holds for K by inspection on the rules for types.

Base case. Proof by inspection on the rules for types. Let K be a base component $[\tilde{x}\rangle\,\tilde{y}]\{L\}$ of type T.

[T5] $<\{x(b^x)\}\uplus X_b;\{y_i(b_i):\mathbf{B}_i:[\mathbf{D}_i]|i\in 1,\ldots,k\}>\xrightarrow{x?(b^x)}<\{x(b^x)\}\uplus X_b;\{y_i(b_i):\mathbf{B}_i:[\mathbf{D}_i']|i\in 1,\ldots,k\}>$, then by inversion we know that $\forall i\in 1,2,\ldots,k$ 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 Definition 4.1, 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 $K\xrightarrow{\mathrm{x?(v)}} K'$. We need to prove that $K'\Downarrow T'$.

We have that $T=<X_b;\{\mathbf{C}_i|i=1,2,\ldots,k\}>$ and $T\xrightarrow{x?(b^x)} T'$ then, by Lemma $C.1$, $T'=<X_b;\{\mathbf{inc}(\{\mathbf{C}_i,x\})|i=1,2,\ldots,k\}>$ (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 Rule [InpBase] we know that $L\xrightarrow{x?v} L'$ and that $x\in\tilde{x}$, so Proposition $A.1$ holds.

By Definition 4.1 we have that $K'\Downarrow T'$.

[T6] $<X_b;\{y(b^y):\mathbf{B}:[\{x_i{:}N_i|i\in 1,\ldots,k\}]\}\uplus\mathbf{C}>\xrightarrow{y!(b^y)}<X_b;\{y(b^y):\mathbf{B}-1:[\{x_i{:}N_i-1|i\in 1,\ldots,k\}]\}\uplus\mathbf{C}>$. By inversion we know that $B>0$ and $N_i>0$ for all $i\in 1,\ldots,k$.

Since it holds that $y(b^y):\mathbf{B}:[\{x_i:N_i|i\in 1,\ldots,k\}]\in\{y(b^y):\mathbf{B}:[\{x_i:N_i|i\in 1,\ldots,k\}]\}\uplus\mathbf{C}$, then by Definition 4.1 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 Rule [OutBase] we know that $L\xrightarrow{y!(v)} L'$ and $y\in\tilde{y}$, so Proposition $A.2$ holds.

By Definition 4.1 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 progress property.

Inductive step We prove that component K enjoys the progress 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 component K_r. Since K is well-typed, all of its subcomonent's modified types are conformant with their local protocol (e.g. $\mathscr{T}(F,T_r)\bowtie G\downharpoonright_r=LP$).

Let us now divide the proof depending on label λ.

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

Since $T\xrightarrow{x?(b^x)} T'$ we know by 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 $\exists K_r',\gamma,v\mid K_r\xrightarrow{\mathrm{z?(v)}} K_r'\wedge K_r'\Downarrow T_r'\wedge\gamma(v)=b^x$ (since the value is forwarded the value is directly input, i.e., the number of internal moves is zero). Applying Rule [InpComp] we have that then $K\xrightarrow{\mathrm{x?(v)}} K'$. Since $K\Downarrow T$ and $K\xrightarrow{\mathrm{x?(v)}} K'$, applying Theorem 4.1 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 Rule [$\mathscr{T}5$] and Definition $D.2$, 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 [$\mathsf{OutComp}$] we have that $K = [\tilde{x} > \tilde{y}]\{G; r = K_r, R; D; r[F]\} \xrightarrow{\mathrm{y!(v)}} K = [\tilde{x} > \tilde{y}]\{G; r = K'_r, R; D; r[F]\}$. 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 $\mathscr{T}(T'_r)G \downharpoonright_r = LP$. Since all the modified type of other components remain the same, by Theorem 4.1 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 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 $fp(LP)$:

Assume that there is one input port, e.g., $z' \in fp(LP)$ (without loss of generality since the reasoning can be reproduced) such that $\overline{y}$ depends on it, and on which does not have the dependency satisfied. Let $LP = G \downharpoonright_r$ 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 $\mathscr{T}(T_r) \bowtie LP$ and $z' \in fp(LP)$ then we can write that $\mathscr{T}(F, T_r) \bowtie \mathscr{C}[z' : b'.LP']$. This implies that $LP' = G' \downarrow_r$ where $G \to \cdots \xrightarrow{r\ell(v')} G'$. Since $K \Downarrow T$ and by induction hypothesis all the subcomponents enjoy the progress property, by Lemma $G.1$ 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 Rule [$InpConf$] (possibly multiple times, toghether with the Rule [$OutConf$]) we have that $\mathscr{T}(F, T'_r) \bowtie G' \downharpoonright_r$ 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 $K'' \xrightarrow{\mathrm{y!(v)}} K'$. Then we have $K \xRightarrow{\mathrm{y!(v)}}$ and Theorem 4.1 (applied multiple times) exists T' such that $K' \Downarrow T'$.

$\square$