A type language for message passing component-based systems

Zorica Savanović
IMT School for Advanced Studies, Lucca, Italy

Letterio Galletta
IMT School for Advanced Studies, Lucca, Italy

Hugo Torres Vieira
C4 - University of Beira Interior, Rua Marquês d'Ávila e Bolama, 6201-001, Covilhã, Portugal

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) [15], where the idea is to build systems relying on the composition of loosely-coupled and independent units called components.

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., [8]) 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 [18] or the choreography diagrams of BPMN [17]. A component should be able to carry out a certain sequence of I/O actions in order to fulfil its role in the protocol. One way to accomplish this is to implement a component in a way that 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 by reactive programming is to design components so that they can respond to external stimulus without any specific I/O sequence. The reactive program-

J. Lange, A. Mavridou, L. Safina, A. Scalas (Eds.):
13th Interaction and Concurrency Experience (ICE 2020).
EPTCS 324, 2020, pp. 3–24, doi:10.4204/EPTCS.324.3

ming 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?

Carbone, Montesi and Vieira [5] proposed a language that supports the development of distributed systems by combining the notions of reactive components with choreographic specifications of communication protocols [16]. 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. [5] propose a verification technique that ensures communication safety and progress. However, the approach requires checking the implementation of components each time the component is put in a different context, i.e., each 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 [5], 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

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

Table 1: Syntax of Governed Components

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 [1] 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.

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 [5] 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 list of (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). E.g., if $y = f(x_1, x_2) < \cdot$ and the component receives v_1 and v_2 in that order on port x_1 and v_3, v_4 and 5 on port x_2. The queue at this point has three mappings $(x_1 \to v_1, x_2 \to v_3), (x_1 \to v_2, x_2 \to v_4), (x_2 \to 5)$ where two are "complete" to compute the function and one is not.

The implementation of a composite component, represented by $\{G;R;D;\mathsf{r}[F]\}$, 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. E.g., every time a value is emitted from some port y it will be carried in a message labelled with e.g., ℓ and a value delivered on some port x is delivered on a message with the same label ℓ. Some other label cannot be attached to y, otherwise the association would not be unique. The $class$ message label (or some other) cannot be attached to y_p otherwise the association would not be unique. Finally, subterm $\mathsf{r}[F]$ 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 [5] is the absence of branching. Then we have terms $\mu\mathbf{X}.G$ and $\mathbf{X}$ 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 [5].

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

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 [5]) 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

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

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

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

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

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

Table 3: Semantics of composite components

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 $D = \mathsf{q}.z \overset{\ell}{\leftarrow} \mathsf{p}.u, D'$ 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 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 for protocol transitions are presented in [5]. 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.

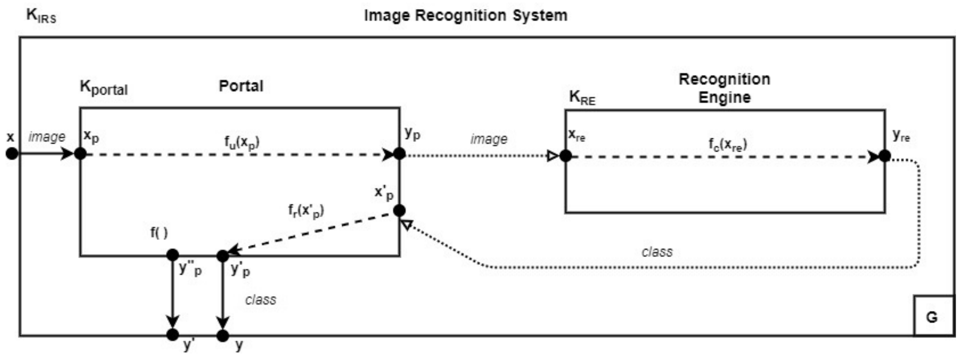
Rule InpChor is similar, but instead of message sending, it addresses the case when a subcomponent receives a message from another subcomponent. The premises are equivalent to the ones for Rule OutChor, but now regard reception. Namely, by saying that receiving component K can exhibit the respective 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 by a microservices scenario [1] that addresses an Image Recognition System (IRS). The basic idea is that users upload images and receive back the resulting classification. Moreover, users can get the current running version of the system whenever desired. The IRS is made of two microservices, Portal and Recognition Engine (RE), that interact according to a predefined protocol.

The 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 as: $\mathsf{Portal} \xrightarrow{image} \mathsf{RE}; \mathsf{RE} \xrightarrow{class} \mathsf{Portal}$. 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

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

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

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

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

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

Let us now assume a recursive version of protocol

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

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

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

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}

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

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 with 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. The first procedure is for base, and the second one is for composite components.

Syntax The syntax of types is presented in Table 4 and some explanations follow. A type T consists of two elements: a (possibly empty) set of input ports, where each one is associated with a basic type b (i.e., int, string, etc.), and a (possibly empty) set of constraints $\mathbf{C}$, 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 through ports, so as to ensure that no unexpected values arise. Each constraint in $\mathbf{C}$ contains a triple of the form

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

$y(b) : \mathbf{B} : [\mathbf{D}]$, 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. The set of constraints $\mathbf{C}$ is ranged over $\mathbf{C}_1, \mathbf{C}_2, \ldots, \mathbf{C}', \mathbf{C}'', \mathbf{C}^y, \ldots$ (likewise other syntactic categories like N, $\mathbf{B}$, $\mathbf{D}, \ldots$). 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.

Note that there are only two kinds of dependencies: a per each value dependency and an initial one. Since we aim at static typing, the dependencies that appear after the extraction of a type are either $x : 0$ or $x : \Omega$, but for the sake of showing our results, we need to capture these values in the evolution of the types. So, we need to capture the number of values available on the input ports, hence we have dependencies of the kind $x : 1, x : 2, \ldots$ (for $N = 1, N = 2 \ldots$).

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

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

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

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

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

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

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

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

Table 5: Type Semantics

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 the dependencies met, i.e., has at least one value for each x for which there is a dependency, and also that the boundary has not been reached (i.e., it is greater than zero), then the type can exhibit the corresponding output implying the decrement of the boundary and of the number of values available in dependencies. Notice that in order for a port to output a value, there can be no initial dependencies present (which are dropped once satisfied), only per each value dependencies.

In the following example and in the rest of the 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:

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

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, remembering their reactive flavour. The goal is to identify the basic types associated to the communication ports, as well as the dependencies between them, while checking that their usage is consistent throughout.

In order to extract the type of a base component we need to define two auxiliary functions. 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 we obtain the list of respective types (e.g., $\gamma(1,\texttt{hello}) = \mathit{integer},\mathit{string}$). 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 $\mathit{count}(x,\tilde{\sigma})$ 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 $\mathit{count} : X \times \Sigma \to \mathbb{N}_0$ is defined as follows:

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

Notice that mappings in $\tilde{\sigma}$ are handled following a FIFO discipline, so the first (oldest) mappings are the ones that need to be accounted for. 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

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

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 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 $\mathit{count}(x,\sigma^{y_i})$.

From an operational 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 from [5], where message labels are mapped to communication ports (thanks to distribution binders D) and also to basic types that describe the communicated values (that can be inferred from the ones of the ports). The syntax of local protocols LP is:

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

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

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

Example 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 $vf(LP,x,y)$ that there is a value flowing from y to x in LP, in which case $vf(LP,x,y) = 1$, otherwise $vf(LP,x,y) = 0$. 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.

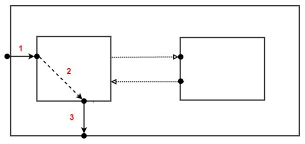


Figure 1: Direct Dependency

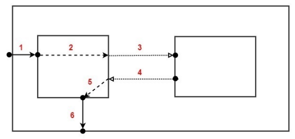


Figure 2: Transitive Dependency

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 $[\tilde{x}' \rangle \tilde{y}']\{G; r = K, R; D; \mathsf{r}[F]\}$ 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:

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

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 ($vf(LP, x', y') = 0$) or already available to be output on y' ($M = 0$), since only in such case (an initial) value is required from the external context (i.e., otherwise if $vf(LP, x', y') = 1$ or $M \geq 0$ then the chain of dependencies is already "internally" satisfied).

Finally, we have the case of per each value transitive dependency, that can only be when both y' and x' are repetitive in the protocol ($i = 3$) and internal dependencies M and M' are both per each value dependencies ($M = N$ and $M' = N'$), which means that the dependency chain is persistent. The number of values available of (external) x for y is the sum of the values available in the internal dependencies (N and N') plus one if there is a value flowing (zero otherwise). Notice that the definition of value flowing presented previously focuses exclusively in the case when y' and x' are repetitive in the protocol, since this is the only case where values might be flowing and the dependency is still present in the protocol structure (i.e., $y' \diamond_3^{LP} x'$ holds). In contrast, a dependency $y' \diamond_i^{LP} x'$ for $i \in \{1,2\}$ is no longer (structurally) present as soon as the value is flowing (i.e., a non-repetitive y' no longer occurs in the protocol after an output).

It might be the case that one output port depends in multiple ways on the same input port. For that reason we introduce a notion of *priority* among dependencies, denoted by $\mathbf{pr}(\ ,\ )$ that gives priority to per each value dependencies (with respect to "initial"). The priority builds on the property that if multiple per each value dependencies (including direct and transitive) are collected (e.g., $x : N_1, \ldots, x : N_k$) then the number of available values specified in them is the same (i.e., $N_1 = \ldots = N_k$). The list of dependencies for port y is then given by the (prioritised) union of direct and transitive dependencies:

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

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

$B_1 = \{N' \mid \mathbf{C} = y(b_1) : \mathbf{B} : [\{x' : N'\} \uplus \mathbf{D}'] \uplus \mathbf{C}' \land x' \notin (fp(LP) \cup F^i)\}$
$B_2 = \{0 \mid \mathbf{C} = \{y(b_1) : \mathbf{B} : [\{x' : \Omega\} \uplus \mathbf{D}']\} \uplus \mathbf{C}' \land x' \notin (fp(LP) \cup F^i)\}$
$B_3 = \{(N'+1) \mid \mathbf{C} = \{y(b_1) : \mathbf{B} : [\{x' : N'\} \uplus \mathbf{D}']\} \uplus \mathbf{C}' \land x' \in fp(LP) \land x' \notin (rep(LP) \cup F^i)\}$

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 a non-repetitive way, hence only one (further) value can be provided.

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

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

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 $\mathbf{ren}(\ ,\ )$ 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 $[\tilde{x}\rangle\, \tilde{y}]\{G; r = K, R; D; \mathsf{r}[F]\}$ be a composite component and $LP = G \upharpoonright_r$ the local protocol for component K. If $T_r = < X^r_b; \mathbf{C}^r >$ 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: $X_b = \{x(b) \mid x(b) \in X^r_b \land x \in F^i\}$
$\boldsymbol{C} = \{y(b') : \boldsymbol{B}(y, LP, \boldsymbol{C}^r) : [\boldsymbol{D}(\boldsymbol{C}^r, F, LP, y)] \mid \boldsymbol{C}^r = \{y(b') : \boldsymbol{B}' : [\boldsymbol{D}']\} \uplus \boldsymbol{C}' \land y \in F^o\}$.

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

Notice that the definition relies on modified types for conformance, but for any type T not associated with the interfacing component we have that $\mathscr{T}(F,T) = T$ 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 $K \xrightarrow{\lambda(v)} K'$. □

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 $K \overset{\lambda(v)}{\Longrightarrow} K'$ 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* $K \overset{\lambda(v)}{\Longrightarrow} K'$ *and* $K' \Downarrow T'$*.*

Proof. By induction on the structure of K. □

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. Intuitively, our types can be seen as *promises of behaviour* in the sense that whatever they prescribe as possible behaviours, the components will eventually deliver. For the sake of addressing any possible component configuration, in particular when components have already all the dependencies (internally) satisfied in order to provide some behaviour, it is crucial that types capture the number of (internally) available resources.

5 Concluding Remarks

In this paper we introduce a type language for the choice-free subset of the GC language [5] 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 [5]). 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. [10]) 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 [9] which actually lay the basis for the protocol language of our target model [5]. The model builds on the idea that protocols can be used to directly program the interaction, and not only serve as a specification/verification mechanism, following the approach of choreographic programming [4, 16].

We discuss some closely related work, starting by Open Multiparty Sessions [3] which to some extent shares the same goals and the same background (cf. [9]). The approach in [3] targets the composition of protocols by considering that one of the participants can actually be instantiated by an external environment. Two protocols can then be connected if there is a participant in each that can serve as the interface to the other interaction. So protocols can be viewed as the units of composition instead of components like in our case, and reusing such protocols in other compositions requires compatibility between the I/O actions which are prescribed for the interfacing role. The main difference is therefore that we consider components that are potentially more reusable considering the I/O flexibility provided the reactive flavour.

We also identify the CHOReVOLUTION [11]project where the assembly of services via a choreography is addressed. The I/O flexibility is provided by adapters at assembly time that can solve I/O interface mismatches between service and choreography. We remark that the CHOReVOLUTION approach is at a very mature state (including tool support [2]), where however an assembly of services cannot be provided as a unit of reuse (like our composite components). Differently, our type-based approach aims at abstracting from the implementation and provides more general support for component substitution and reuse. Similar to our work, the model of *service based architectures* (SOA's) [13] exposes components that exchange services between each other via ports. However, the authors do not present the type language, where some ideas we presented for extracting a type of a component could be used for the model they present.

We may also find the notion of distributed components is the Signal Calculus (SC) [6], where in each component a process is located. The type-based approach presented in [7] addresses the issue of ensuring SC local processes are composed and interact in a way such that they follow a well-defined protocol of interaction. Our model embeds the protocol in the operational semantics so such coordination is ensured by construction. Our types focus on a different purpose of ensuring data dependencies are met in order to ensure components are not stuck in the sense of the progress result.

In the work about Interactive verification of ADPs [14] the authors introduce a notion of component type which characterizes components with a certain behaviour. One core difference between [14] and our work is that the authors do not provide any means to automatically extract types from given components.

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 [12] 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.

Acknolegements We thank the anonymous reviewers for their suggestions and comments which helped us to improve the paper.

References

[1] Amazon Web Services, Inc (2019): *AWS Lambda: Developer Guide*. Available at `https://docs.aws.amazon.com/en_pv/lambda/latest/dg/lambda-dg.pdf#welcome`.

[2] Marco Autili, Amleto Di Salle, Francesco Gallo, Claudio Pompilio & Massimo Tivoli (2019): *CHOReVOLUTION: Automating the Realization of Highly–Collaborative Distributed Applications*. In Hanne Riis Nielson & Emilio Tuosto, editors: *21th International Conference on Coordination Languages and Models (COORDINATION), Coordination Models and Languages* LNCS-11533, Springer International Publishing, pp. 92–108, doi:10.1007/978-3-030-22397-7_6. Part 2: Tools (1).

[3] Franco Barbanera & Mariangiola Dezani-Ciancaglini (2019): *Open Multiparty Sessions*. In Massimo Bartoletti, Ludovic Henrio, Anastasia Mavridou & Alceste Scalas, editors: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, *EPTCS* 304, pp. 77–96, doi:10.4204/EPTCS.304.6.

[4] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In Roberto Giacobazzi & Radhia Cousot, editors: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, ACM, pp. 263–274, doi:10.1145/2429069.2429101.

[5] Marco Carbone, Fabrizio Montesi & Hugo Torres Vieira (2018): *Choreographies for Reactive Programming*. CoRR abs/1801.08107. Available at `http://arxiv.org/abs/1801.08107`.

[6] Gian Luigi Ferrari, Roberto Guanciale & Daniele Strollo (2006): *JSCL: A Middleware for Service Coordination*. In Elie Najm, Jean-François Pradat-Peyre & Véronique Donzeau-Gouge, editors: *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006*, *Lecture Notes in Computer Science* 4229, Springer, pp. 46–60, doi:10.1007/11888116_4.

[7] GianLuigi Ferrari, Roberto Guanciale, Daniele Strollo & Emilio Tuosto (2007): *Coordination Via Types in an Event-Based Framework*. In John Derrick & Jüri Vain, editors: *Formal Techniques for Networked and Distributed Systems - FORTE 2007, 27th IFIP WG 6.1 International Conference, Tallinn, Estonia, June 27-29, 2007, Proceedings*, *Lecture Notes in Computer Science* 4574, Springer, pp. 66–80, doi:10.1007/978-3-540-73196-2_5.

[8] Nicola Dragoniand Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin & Larisa Safina (2017): *Microservices: Yesterday, today, and tomorrow*. In Manuel Mazzara & Bertrand Meyer, editors: *Present and Ulterior Software Engineering*, Springer, pp. 195–216, doi:10.1007/978-3-319-67425-4_12.

[9] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[10] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira & Gianluigi Zavattaro (2016): *Foundations of Session Types and Behavioural Contracts*. *ACM Comput. Surv.* 49(1), pp. 3:1–3:36, doi:10.1145/2873052.

[11] S. Keller, M. Autili & M. Tivoli (2014): *CHOReVOLUTION project*. `http://www.chorevolution.eu`.

[12] Barbara Liskov & Jeannette M. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Trans. Program. Lang. Syst.* 16(6), pp. 1811–1841, doi:10.1145/197320.197383.

[13] A. Malkis & D. Marmsoler (2015): *A Model of Service-Oriented Architectures*. In: *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 110–119, doi:10.1109/SBCARS.2015.22.

[14] Diego Marmsoler & Habtom Kashay Gidey (2019): *Interactive verification of architectural design patterns in FACTum*. *Formal Aspects of Computing* 31(5), pp. 541–610, doi:10.1007/s00165-019-00488-x. Alessandra Russo, Andy Schuerr, and Heike Wehrheim.

[15] M. Douglas Mcllroy (1969): *Mass Produced Software Components*. In: *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, pp. 138–155.

[16] Fabrizio Montesi (2013): *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen. Available at `http://fabriziomontesi.com/files/choreographic_programming.pdf`.

[17] Object Management Group, Inc. (OMG) (2014): *Business Process Model and Notation, specification version 2.0.2*. Available at `https://www.omg.org/spec/BPMN/2.0.2/`.

[18] W3C WS-CDL Working Group (2004): *Web Services Choreography Description Language Version 1.0*. Available at `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/`.

A 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	Boundaries	
$\mathscr{M} ::= \mathscr{N} \mid \Omega$	$\mathbf{B} ::= N \mid \infty$	$k \geq 0; N \in \mathbb{N}_0$
$\mathscr{N} ::= N \mid \infty$		

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

$\mathscr{T}$-Type semantics

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

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

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

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

Table 7: $\mathscr{T}$ Semantics

Definition A.1. *If $T_r =< X_b;\boldsymbol{C} >$ is a type of interfacing subcomponent $\overline{K}$ of composite component $[\tilde{x} > \tilde{y}]\{G; r = \overline{K}, R; D; r[F]\}$ then $\mathscr{T}(F,T_r)$ is the T_r-modified type where:*

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

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

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

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

B Conformance relation

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

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

Table 8: Conformance

Definition B.1. *$\mathscr{T}' \leq \mathscr{T}$ if exists a (possibly empty) set of typed input ports $\{x_1(b_1), x_2(b_2), \dots, x_k(b_k)\}$ such that $\mathscr{T}' \xrightarrow{x_1?(b_1)} \cdots \xrightarrow{x_k?(b_k)} \mathscr{T}$.*

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 $B.1$). Observing the semantics of modified types, 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 $\mathbf{X} : \mathscr{T}$.