Discretionary Information Flow Control for Interaction-Oriented Specifications

Lluch Lafuente, Alberto; Nielson, Flemming; Nielson, Hanne Riis

Published in:
Logic, Rewriting, and Concurrency

Link to article, DOI:
10.1007/978-3-319-23165-5_20

Publication date:
2015

Document Version
Peer reviewed version

Citation (APA):
Lluch Lafuente, A., Nielson, F., & Nielson, H. R. (2015). Discretionary Information Flow Control for Interaction-Oriented Specifications. In *Logic, Rewriting, and Concurrency: Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday* (pp. 427-450). Springer. https://doi.org/10.1007/978-3-319-23165-5_20

Discretionary Information Flow Control for Interaction-Oriented Specifications

Alberto Lluch Lafuente, Flemming Nielson, and Hanne Riis Nielson

DTU Compute, The Technical University of Denmark, Denmark

Abstract. This paper presents an approach to specify and check discretionary information flow properties of concurrent systems. The approach is inspired by the success of the interaction-oriented paradigm to concurrent systems (cf. choreographies, behavioural types, protocols,...) in providing behavioural guarantees of global properties such as deadlock-absence. We show how some information flow properties are easier to formalise and check on a global interaction-oriented description of a concurrent system rather than on a local process-oriented description of the components of the system. We use a simple choreography description language adapted from the literature of choreographies and session types. We provide a generic method to instrument the semantics with information flow annotations. Policies are used to specify the admissible flows of information. The main contribution of the paper is a sound type system for statically checking if a system specification ensures an information flow policy. The approach is illustrated with two archetypal examples of distributed and parallel computing systems: a protocol for an identity-secured data providing service and a parallel MapReduce computation.

Keywords: Information Flow Control, Discretionary Access Control, Choreographies, Communication Protocols, Interaction-Oriented Computing, Parallel Computing, Service-Oriented Computing, High-Performance Computing

1 Introduction

The flow of information within a concurrent system is often expected to satisfy some properties related to which components can access which data and how. Such properties are known as *discretionary* access control policies and provide a fine-grained control over the flow of information, as opposed to other kinds of security policies that often regard the non-interference between security levels of information. Consider for instance, the following concurrent program:

$$\big[\ \mathsf{k!x}\ \big]_{\mathsf{p}} \qquad \big[\ \mathsf{k?y}\ ;\ \mathsf{k'!``go"}\ ;\ \mathsf{k!y'}\ \big]_{\mathsf{q}} \qquad \big[\ \mathsf{k'?z}\ ;\ \mathsf{k?z}\ \big]_{\mathsf{r}}$$

where $u!v$ denotes the sending of a message v over a channel u, $u?w$ denotes the reception on variable w of a message on channel u, sequential composition is denoted by ; and the sequential code of the concurrent processes p, q, r composing the system is enclosed between square brackets. Is there a flow of information from variable x to variable z? A simple analysis of all possible executions of the system may provide a negative answer depending on the kind of information flow one is interested in (explicit, implicit and so on). However, such an *a posteriori* verification is undecidable in general and often unfeasible (e.g. due to state space explosion). An *a priori* static analysis, however, should be smart enough to discard the potential flow of information over channel k if one is interested in explicit data flows. Indeed, x is sent over channel k and z is obtained from k as well. It is a matter of synchronisation that z will not receive the value of x.

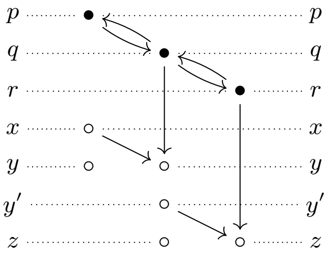

Fig. 1. A flow of information

Fig. 1 shows a graphical representation of some flow of information in our example. A detailed explanation of our graphical notation will be provided later, here it suffices to understand that the flow of information is represented with a graph (circles and arrows) equipped with an interface (left and right list of principal and variable names). Of course, the figure depicts only *some* flows of information, in particular it represents explicit data flows between variables, write access from principals to variables and control dependencies due to interactions. The precise notion of information flow one would like to consier may vary, depending of the properties of interest, the application domain, the context of execution where the system will be deployed and so on. Note, in particular, the absence of channels in the depicted flow, which may be an appropriate choice in case of deployment on a framework with synchronous channels that retain no data after synchronisation. Letting apart this simple illustrative example, a priori verification of discretionary information flow policies in concurrent systems is a challenging task that urges suitable solutions to ease the engineering of trustworthy-by-design systems in the era of of big data, data-drivenness and massive parallel/concurrent/high-performance computing.

Contribution. We investigate in this paper an application of the interaction-oriented paradigm to the specification of *trustworthy-by-design* concurrent systems. Our work shares the same constructive attitude towards security promoted in [4] of providing methodologies and techniques to support *security-by-design.* Even if the work was motivated by security concerns, our approach can be applied to other aspects of concurrent and distributed systems where information flows play a fundamental role, like performance issues related to locality of data access, or robustness issues related to control dependencies among processes. Our work is also inspired by the success of the interaction-oriented paradigm in providing deadlock-freedom by design in distributed systems (see e.g. [8]). Many information flow properties are global in nature, which suggests that they should be easier to formalise and check on a global description of the system rather than on the local description of the individuals.

The basis of the approach is the specification of the system by means of a choreography, i.e. a global description of the expected interactions of a system in terms of the messages exchanged between its components. Our choreographies are also enriched with a specification of the information being used by processes in their decision points and about local updates of data. Such descriptions allows one to design and analyse information systems top-down, where the behaviour of the individuals is synthesised from (or specified in) the description of the global behaviour of the system. As an example, consider the following choreography:

$$\mathsf{C1} \triangleq \mathsf{p.x} \;\texttt{->}\; \mathsf{q.y} : \mathsf{k} \;;\; \mathsf{q.``go"} \;\texttt{->}\; \mathsf{r.z} : \mathsf{k'} \;;\; \mathsf{q.y'} \;\texttt{->}\; \mathsf{r.z} : \mathsf{k}$$

where $u.e \;\texttt{->}\; u'.e' : c$ specifies that component u sends expression e over channel c to component u', that stores the message according to the pattern e'. This choreography specifies the very same concurrent system we saw before. However, statically checking the absence of an explicit data flow from x to z is easier as the choreography-based description resembles a sequential program where traditional information flow analysis techniques may be adapted and applied. For instance, the flow of Fig. 1 can be easily extracted from the static description of $\mathsf{C1}$.

We start our work defining a formal choreography description language for specifying the global behaviour of the system. The language is strongly based on existing approaches based on process calculi (e.g. session calculi) and behavioural types (e.g. session types). As usual in those traditions, we consider a notion of well-formedness for choreographies, to rule out systems for which providing information flow guarantees is not trivial. The main contribution of the paper is a sound type system for information flow policies, i.e. one that ensures that if a choreography C is typed with a policy Π, denoted $\mathbf{Ent} \vdash C : \Pi$, we can conclude that C is Π-secure, i.e. the information flows of the behaviours described by C satisfy the policy Π, denoted $C \models \Pi$. In the judgement $\mathbf{Ent} \vdash C : \Pi$, $\mathbf{Ent}$ denotes the set of *entities* (principals, variables, channels) involved in C and Π. A key role in our approach is played by the use of an *instrumented semantics* [24] for our language, where semantic rules are enriched with annotations relevant to the flows of information. The instrumentation of the semantics is parametric with respect to the flows associated to the main events in the choreography (interactions, local updates, choices). This provides a convenient degree of flexibility to the user, who can specify the notion of information flow that better suits his purposes. This is one of the reasons why information flow assurances in our approach are not related to a *non-interference* [10,16] result. In our experience, non-interference cannot be easily conveyed to software, safety or security engineers and often provides a too strong requirement with respect to the the kind of information flow properties of interest. Our information flow policies are based on the *Decentralized Label Model* [23]. We use here a graphical notation for information flows and policies based on graphs with interfaces [6,11]. Though not technically different from relational-based notations we think that the use of graphs provides a formal and visually appealing presentation, suitable for software, safety and security engineers and in line with other successful graphical notations such as message sequence charts, fault trees and attack trees. The type system is as well parametric with respect to the flows associated to the choreography events and its soundness relies on some well-formedness constraints of the flow annotations.

```
u.name -> rp.user : a ;
rp.user -> ip.id : b ;
u.my_pwd -> ip.pwd : a ;
if check(id,pwd)@ip then
    ip.“ok” -> rp.“ok” : b ;
    rp.class(user) -> s.class : c
else
    ip.“fail” -> rp.“fail” : b ;
    rp.“na” -> s.class : c ;
( ip.rep(id) -> rp.report : b
| ( data := first(class) @ s;
    while data ≠ nil @s do
        s.data -> u.info : a ;
        data := data.next @ s
    then
        s.“end” -> u.info : a  ) )
```

```
[ a!name ;
  a!my_pwd ;
  loop
     a?info
     ⊕ (a?“end” ;
         break )
]u
```

```
[ b?id ;
  a?pwd ;
  if check(id,pwd) then
     b!“ok”
  else
     b!“fail” ;
  b!rep(id) ;
]ip
```

```
[ a?user ;
  b!user ;
  ( (b?“ok” ;
     c!class(user))
     ⊕ (b?“fail” )
         c!“na”) ;
  b?report ;
]rp
```

```
[ c?class ;
  data := first(class) ;
  while data ≠ nil then
     a!data ;
     data := data.next ;
  then
     a!“end” ;
]s
```

Fig. 2. Interaction- (left) and process-oriented (right) specification of a service

Structure of the Paper. Section 2 presents two paradigmatic case studies aimed at providing some additional motivation and insights, and to serve as running examples. Section 3 presents a simple choreography description language, defining its formal semantics and a notion of well-formedness. Section 4 presents our graphical notation for information flows, the annotation-parametric instrumentation of the semantics and the well-formedness conditions on flow annotations. Section 5 formalises the notion of satisfaction of a policy by a choreography and presents the sound type system that allows us to statically check if a choreography satisfies a security policy. Section 6 discusses related works, concludes the paper and describes our current and future research investigations.

2 Applications: Protocols, Services and HPC

We present in this section two case studies from different domains, to provide additional motivations and insights on the approach, as well as to serve as running examples throughout the paper. The first case study (Section 2.1) is a archetypal example of a distributed system, namely a protocol used in an identity-secured data providing service. The second case study (Section 2.2) is an archetypal example of a parallel program, namely a parallel MapReduce computation.

2.1 An Identity-Secured Data Providing Service

Our first case study is inspired on the OpenID example of [22], slightly adapted for presentation purposes. The system consists of a user u trying to retrieve some data from a server s. The access to the desired data is subject to authentication

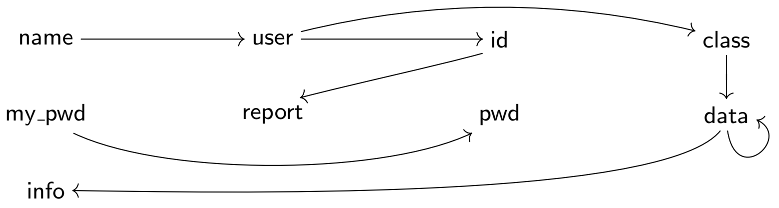

Fig. 3. Policy $\Pi_{s,1}$ for the Data Providing Service

by an identity authentication party ip. The interaction between u, s and ip is coordinated by a relying party rp.

Figure 2 provides two descriptions of this case study. The one on the left provides an interaction-oriented description (namely, a choreography), while the one on the right provides a process-oriented description (i.e. a distributed specification). The choreography, P for short , is specified in our language, to be presented later, while the process-oriented description is specified in a language based on standard constructs of concurrent languages (featuring asymmetric, binary, synchronous, pattern-based communication over channels). The four components of the system (u, rp, ip and s) exchange some information through three channels (a, b and c). Some information may be sensitive (e.g. passwords, data, user classes, etc.) so that we may want to impose some policy on the way such information flows.

As in the simple example presented in the Introduction, a first look at the process-oriented specification may suggest some potential flows of information. For instance, we can see that the content of data is sent over channel a, from which both the relying party rp and the authentication party ip read messages. The user password and the user class are involved in similar situations. A security engineer may want to restrict and check those flows of information.

An example of a policy that one may be interested in is depicted in Figure 3. The policy focuses on explicit data flows only. The policy allows explicit data flows between some of the variables involved in the system but forbids others. For instance, flows from my_pwd to report or data, which could compromise the password of the user, are not allowed. Information contained in data is only allowed to flow to data itself or to info, thus we forbid data to flow to any of the other variables that principals ip and rp may access.

The satisfaction of this policy depends on the kind of information flows one wants to consider. If only explicit flows are considered, the policy is satisfied. This may not be trivial by inspecting the process-oriented specification but a look at the interaction-oriented specification may be more reassuring. For instance, my_pwd flows only directly to pwd through the interaction u.my_pwd -> ip.pwd : a, but pwd is not explicity used to update any variable or and is never communicated. Moreover, the choreography clearly specifies that data is only used to calculate the next piece of data or to be transferred to u's info, and similarly for the rest of the sensitive informations we have mentioned.

The situation is different, of course, if explicit flows are considered as well. We will see this and examples of other policies in the rest of the paper.

2.2 A Binary Parallel Map-Reduce Computation

Our next case study is an archetypal example of parallel computations, namely MapReduce [12]. MapReduce is very popular pattern for processing large data sets in parallel, which has its origins in functional programming primitives such Lisp's map and reduce and High-Performance Computing primitives such as MPI's scatter and reduce operations. We consider here a simple case in which the programmer is interested in computing the function $\mathsf{red}(\mathsf{map}(\mathsf{x}_0, \ldots, \mathsf{x}_m))$, where $\mathsf{map} : T \rightarrow T'$ is a function that processes single values of some type T into single values of some other type T' (and that can be piece-wise lifted to vectors as we do above) and $\mathsf{red} : {T'}^* \rightarrow T'$ is a function to accumulate a vector of values of type T' into a single value of type T'. We assume, as usual in MapReduce, that red is associative and commutative and amounts to the identity function on vectors of size 1. This allows one to decompose function red as a binary function, which greatly helps the accumulation of values in parallel.

$$
\begin{array}{rcl}
\mathsf{M}_{i,0} & \triangleq & \mathsf{y}_i\mathsf{:=}\ \mathsf{map}(\mathsf{x}_i)@\mathsf{p}_i \\
\mathsf{M}_{i,n+1} & \triangleq & \mathsf{p}_i\texttt{->}\mathsf{p}_{i+2^n}\text{: } \mathsf{k}_{i,i+2^n}\ ; \\
 & & (\mathsf{M}_{i,n} \mid \mathsf{M}_{i+2^n,n})\ ; \\
 & & \mathsf{p}_{i+2^n}\texttt{->}\mathsf{p}_i\text{: } \mathsf{k}_{i+2^n,i}\ ; \\
 & & \mathsf{y}_i\mathsf{:=}\ \mathsf{red}(\mathsf{y}_i, \mathsf{y}_{i+2^n})@\mathsf{p}_i
\end{array}
$$

Fig. 4. MapReduce Scheme

Figure 4 is a general scheme of a possible interaction-oriented MapReduce specification in our choreography description language. The main idea of the scheme is that in a MapReduce choreography $\mathsf{M}_{i,n+1}$, a principal p_i will act as the *leader* of 2^{n+1} principals in the computation of $\mathsf{red}(\mathsf{map}(\mathsf{x}_i, \ldots, \mathsf{x}_{i+2^{n+1}}))$ to be stored in y_i. The leader principal will decompose that computation into the problem of computing $\mathsf{red}(\mathsf{map}(\mathsf{x}_i, \ldots, \mathsf{x}_{i+2^n}))$ in y_i and $\mathsf{red}(\mathsf{map}(\mathsf{x}_{i+2^n+1}, \ldots, \mathsf{x}_{i+2^{n+1}}))$ in y_{i+2^n} first, and accumulating the results afterwards. The first sub-problem will be solved by p_i itself, while the second sub-problem will be delegated to principal p_{i+2^n}. Both sub-problems are solved applying the very same scheme. The interactions do not involve any value passing and are simply used to trigger the computations first (roughly, p_i wakes up p_{i+2^n}) and later to ensure that the data to be accumulated has been indeed computed (roughly, p_i waits for p_{i+2^n} to finish). For this purpose the choreography uses pairs of channels $\mathsf{k}_{u,v}$ to be used exclusively for p_u to synchronise with p_v.

Figure 5 presents an instance $\mathsf{M}_{0,2}$ of the above choreography scheme on a scenario with four principals, together with an equivalent process-oriented specification. In this example an information flow analysis can reveal interesting information regarding data locality or control dependencies. For instance, one would be interested in controlling which principals can affect which other principals for the sake of a analysing the robustness of the computation in terms of failure dependencies. In addition, one would like to control the access of data by principals for the sake of ensuring performance by maximising access locality.

```
p_0->p_2: k_{0,2} ;
(
    p_0->p_1: k_{0,1} ;
    ( y_0:= map(x_0)@p_0 | y_1:= map(x_1)@p_1 ) ;
    p_1->p_0: k_{1,0} ;
    y_0:= red(y_0, y_1)@p_0
|
    p_2->p_3: k_{2,3} ;
    ( y_2:= map(x_2)@p_2 | y_3:= map(x_3)@p_3 ) ;
    p_3->p_2: k_{3,2} ;
    y_2:= red(y_2, y_3)@p_2
) ;
p_2->p_0: k_{2,0} ;
y_0:= red(y_0, y_2)@p_0
```

$$\left[\begin{array}{l} \mathsf{k}_{0,2}!\ ; \\ \mathsf{k}_{0,1}!\ ; \\ \mathsf{y}_0 := \mathsf{map}(\mathsf{x}_0)\ ; \\ \mathsf{k}_{1,0}?\ ; \\ \mathsf{y}_0 := \mathsf{red}(\mathsf{y}_0, \mathsf{y}_1)@\mathsf{p}_0\ ; \\ \mathsf{k}_{2,0}?\ ; \\ \mathsf{y}_0 := \mathsf{red}(\mathsf{y}_0, \mathsf{y}_2)@\mathsf{p}_0 \end{array}\right]_{\mathsf{p}_0} \quad \left[\begin{array}{l} \mathsf{k}_{0,1}?\ ; \\ \mathsf{y}_1 := \mathsf{map}(\mathsf{x}_1)\ ; \\ \mathsf{k}_{1,0}! \end{array}\right]_{\mathsf{p}_1}$$

$$\left[\begin{array}{l} \mathsf{k}_{0,2}?\ ; \\ \mathsf{k}_{2,3}!\ ; \\ \mathsf{y}_2 := \mathsf{map}(\mathsf{x}_2)\ ; \\ \mathsf{k}_{3,2}?\ ; \\ \mathsf{y}_2 := \mathsf{red}(\mathsf{y}_2, \mathsf{y}_3)@\mathsf{p}_0\ ; \\ \mathsf{k}_{2,0}! \end{array}\right]_{\mathsf{p}_2} \quad \left[\begin{array}{l} \mathsf{k}_{2,3}?\ ; \\ \mathsf{y}_3 := \mathsf{map}(\mathsf{x}_3)\ ; \\ \mathsf{k}_{3,2}! \end{array}\right]_{\mathsf{p}_3}$$

Fig. 5. Interaction- (left) and process-oriented (right) specification of MapReduce

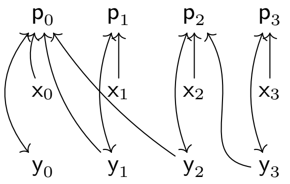

Fig. 6. Policy $\Pi_{m,1}$ for MapReduce

As an example of a concrete policy, consider Figure 6. The policy focuses on how principals access data, i.e. which principal in the computation is accessing which data variable. The policy allows each principal p_i to read variable x_i and to read and write on variable y_i. Additional write permissions are granted to allow accumulation: p_0 is allowed to read y_2 and y_1 and p_2 is allowed to read y_3. This policy is actually satisfied by the choreography (and its distributed process-oriented counterpart) and is indeed more permissive than needed. For instance, p_1 is allowed to read variable y_i but that does not occur. This policy illustrates that sometimes one is not interested in explicit data flows. Indeed, controlling similar policies can be very useful when the system is to be deployed on an parallel architecture and principals and data have to be allocated in computational resources such as processors, machines, memory locations, etc.

All in all, the case studies we have presented and the examples of information flow policies motivate the need to consider different notions of information flow and that is the reason why our framework is parametric with respect of the notion of flow to be considered.

3 Choreographic Specifications of Concurrent Systems

A choreography describes the expected interactions of a system in terms of the messages exchanged between its components. Choreographies can be used to automatically derive (via *endpoint projections*) distributed code skeletons or local specifications to be checked on existing implementations. The latter case is typical, for instance, of legacy systems (where existing implementations may be available) or open systems (where the principals may be governed by independent parties). Choreographies are usually given a so called *weak*

$$\begin{array}{rcll}
C & ::= & C; C' & \text{(sequential composition)}\\
 & | & C \mid C' & \text{(parallel composition)}\\
 & | & \mathsf{if}\ e@p\ \mathsf{then}\ C\ \mathsf{else}\ C' & \text{(choice)}\\
 & | & \mathsf{while}\ e@p\ \mathsf{do}\ C\ \mathsf{then}\ C' & \text{(loop)}\\
 & | & A & \text{(actions)}\\
A & ::= & x := e\ @p & \text{(update)}\\
 & | & p.e\ \texttt{->}\ q.e' : k & \text{(interaction)}\\
 & | & \mathsf{skip} & \text{(skip)}
\end{array}$$

Fig. 7. A simple choreography description language

(or *partial* or *constraint*) interpretation [21]: a (distributed) realisation S of a choreography C is admitted if it exhibits a subset of the behaviours specified by the choreography. In a trace-based setting, the question can be rephrased as $\mathit{Traces}(C) \subseteq \mathit{Trace}(S)$. This does not necessarily prescribe the presence of unobservable/hidden interactions aimed at realising the choreography. For instance, in a trace-based settings the above mentioned notion of admissibility of realisations can be relaxed to $\mathit{Traces}(C)_{|\mathcal{O}} \subseteq \mathit{Trace}(S)_{|\mathcal{O}}$, where $_{|\mathcal{O}}$ is the projection on a set of observables $\mathcal{O}$ that would discard hidden interactions. However, most approaches to choreographies are based on the idea that the choreography specifies *all* the interactions that may be observed in the system and assume that no additional interactions will take place in the realisation of the choreography. We believe that this is methodologically more adequate for information flow control: implicit hidden interactions may introduce unexpected flows of informations, whereas requiring all interactions to be explicitly declared should help understanding the actual flows of information and should mitigated the unintentional introduction of undesired flows.

As a consequence, not all choreographies are *realisable* in a distributed way. A typical example is the choreography `p.e -> q.x : k ; r.e' -> s.y : k'` where, clearly, there is no way of imposing the order of the interactions without introducing additional ones. Typically, well-formedness conditions are given to impose some semantic and syntactic constraints on choreographies that ensure good properties in terms of realizability and soundness of the endpoint projections.

Choreographies: Syntax. We consider a simple choreography description language inspired by process calculi and session types approaches to choreographies. In particular, the syntax of our language is close to the *interaction oriented language* [20], the *choreography calculus* [22,8] and the *global types* used in [9,3].

We use universes of variables **Var**, pattern expressions **Expr** over variables, principals **Prin**, and channels **Chan**. We denote the union of principals, variables and channels the *entities* and denote them with $\mathbf{Ent} = \mathbf{Prin} \cup \mathbf{Var} \cup \mathbf{Chan}$.

Definition 1 (syntax of choreographies). *The syntax of our choreography description language is defined by the grammar of Fig. 7, where* $e, e' \in \mathbf{Expr}$*,* $x \in \mathbf{Var}$*,* $p, q \in \mathbf{Prin}$ *are distinct principals (*$p \neq q$*), and* $k \in \mathbf{Chan}$.

The set of principals (reps. entities) in a choreography C is denoted by $pn(C)$ (reps. $en(C)$), the set of variables in an expression e is denoted by $v(e)$. These functions are defined as expected and we hence neglect their formal definition. We assume that variable names cannot be used as values in expressions, to forbid mechanisms such as indirect references and name passing in interactions, which may pose additional challenges in our technique.

The syntactic category C corresponds to choreographies. To avoid confusion between individual choreographies and the syntactic category, we sometimes use $\mathcal{C}$ to denote the set of all terms generated by C. The syntactic category A corresponds to actions. This syntactic category is not strictly necessary but simplifies the presentation of our approach. We shall use a set $\mathcal{A}$ of *events* defined as the union of all terms generated by A and all expressions of the form $e\texttt{@}p$.

The language includes classical constructs such as sequential and parallel composition, branching and loops. One feature to be remarked is that decision points are annotated with the name of a principal (cf. $\texttt{@}p$ in loops and choices). By doing so one can specify which p principal is the *selector*, i.e. the principal responsible for taking the control decision. Another remarkable difference with respect to standard languages is that the `while` construct has a termination code in addition to the body. Similar annotations are not new in choreographies and are used e.g. in [9,3]. The idea is that such code is used by the selector principal to notify termination to all passive processes involved in the body. This feature is not strictly necessary in our work but we prefer to have it in order make our language closer to the ones used in the literature of choreographies.

Actions include local updates of the form $x := e \ \texttt{@}p$ where principal p updates variable x with the result of evaluating expression e, the void action `skip` and a binary interaction $p.e \ \texttt{->} \ q.e' : k$ between principals p and q. In such an interaction principal p aims at sending over channel k the result of evaluating expression e to principal q. Expression e is to be matched against the pattern e' whose variables act as binders to collect the result of the interaction. As we have seen, e and e' can be void in which case we use the simplified notation $p \ \texttt{->} \ q : k$.

Semantics of Choreographies. We present two semantics for our language: an operational semantics, aimed at providing a first insight to the reader, and a denotational semantics, which eases the presentation of the main results.

The operational semantics of our language is the relation $\rightarrow \subseteq \ \mathcal{C} \times \mathcal{A} \times \mathcal{C}$ defined by the rules in Fig. 8. The rules are very similar to those of standard parallel programming languages or process calculi. We just remark here that the semantics is abstract with respect to the actual evaluation of expressions: the branching in choices and loops can be seen as non-deterministic choices.

The denotational semantics defines the traces of a choreography as words over the alphabet of events $\mathcal{A}$. As in some approaches to choreographies (e.g. [9]), we restrict ourselves to finite words, and hence finite traces.

$$\frac{C_1 \xrightarrow{\alpha} C_1'}{C_1; C_2 \xrightarrow{\alpha} C_1'; C_2} \qquad \frac{}{\mathsf{skip}; C \xrightarrow{\mathsf{skip}} C} \qquad \frac{A \neq \mathsf{skip}}{A \xrightarrow{A} \mathsf{skip}}$$

$$\frac{i \neq j \in \{1,2\} \qquad C_i \xrightarrow{\alpha} C_i' \qquad C_j' = C_j}{C_1 \mid C_2 \xrightarrow{\alpha} C_1' \mid C_2'}$$

$$\frac{i \in \{1,2\}}{\mathsf{if}\ e@p\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \xrightarrow{e@p} C_i} \qquad \frac{i \in \{1,2\} \quad C_1 = C\ ;\ \mathsf{while}\ e@p\ \mathsf{do}\ C\ \mathsf{then}\ C_2}{\mathsf{while}\ e@p\ \mathsf{do}\ C\ \mathsf{then}\ C_2 \xrightarrow{e@p} C_i}$$

Fig. 8. Operational Semantics of Choreographies

Definition 2 (trace semantics). *The trace semantics of our language is given by function* $Traces : \mathcal{C} \to 2^{\mathcal{A}^*}$ *defined by*

$$\begin{aligned}
Traces(C_1; C_2) &= Traces(C_1)\, Traces(C_2)\\
Traces(C_1 \mid C_2) &= Traces(C_1) \bowtie Traces(C_2)\\
Traces(\mathit{if}\ e@p\ \mathit{then}\ C_1\ \mathit{else}\ C_2) &= e@p\ (Traces(C_1) \cup Traces(C_2))\\
Traces(\mathit{while}\ e@p\ \mathit{do}\ C_1\ \mathit{then}\ C_2) &= e@p\ (Traces(C_1)\ e@p)^*\, Traces(C_2)\\
Traces(A) &= \{A\}
\end{aligned}$$

Above, juxtaposition denotes the concatenation of traces, the unary operator $\_*$ is the usual Kleene star of regular expressions, and the binary operator $\bowtie$ denotes the *shuffling* of trace sets, i.e. $T \bowtie T' = \{\sigma_1\sigma_1' \ldots \sigma_n\sigma_n' \mid \sigma_1 \ldots \sigma_n \in T \wedge \sigma_1' \ldots \sigma_n' \in T'\}$. The empty trace will be denoted by ϵ. Both semantics can be shown to be equivalent for finite behaviours: the finite traces of the transition system defined by the operational semantics coincide with the traces defined by the denotational semantics (up to occurrences of skip).

Well-Formed Choreographies. As mentioned, choreographies should enjoy a couple of properties to be useful in practice, e.g. to ensure distributed realizability and soundness of endpoint projections. It is common practice to define a notion of well-formed choreography and, possibly, syntactic restrictions to ensure well-formedness. In our work, well-formedness is just needed for the correctness of our type system and we hence provide a simple notion tailored for our purpose.

Definition 3 (well-formed choreography). *Let C be a choreography. We say that C is* well-formed *if the following conditions hold:*

1. *every occurrence of $C_1 \mid C_2$ in C should be such that $en(C_1) \cap en(C_2) = \emptyset$;*
2. *all traces $\sigma \in Traces(C)$ satisfy the following condition: If $\sigma = \sigma'\alpha\beta\sigma''$, with $\alpha, \beta \in \mathcal{A}$ then $pn(\alpha) \cap pn(\beta) \neq \emptyset$ or $\sigma'\beta\alpha\sigma'' \in Traces(C)$.*

Our notion of well-formedness is reminiscent of the semantic notion of well-formedness used in [9] and some syntactic restrictions taken from [19]. Intuitively, well-formedness requires (1) no entity can be involved in both branches of a parallel composition, and (2) that the set of traces of a choreography is closed under the transposition of actions involving disjoint principals. Note also, that our notion of well-formedness is not strong enough to guarantee realisability. For example, the choreography `p . e -> q . e' : k ; r . e'' -> q . e''' : k` (adapted from examples of ill-formed choreographies of [19]) is well-formed according to our notion, but cannot be realised due to the race condition on channel `k`. This is not a problem: it just means that our technique applies to more choreographies than needed in practice.

4 Information Flows in Choreographies

We provide in this section our notion of information flows and the mechanism to instrument the semantics of choreographies with flow annotations.

Information Flows as Graphs with Interfaces. A flow F is essentially a relation among entities, possibly expressing how entities influence or depend on each other. We represent flows in this paper using graphs with interfaces [6,11] as they provide an intuitive visual representation and elegant and well-defined notions of flow composition.

A graph with a discrete interface (cf. Def. 11 in Section A) is denoted by $I \xrightarrow{i} G \xleftarrow{o} O$ and is defined by an input interface I (a set of nodes), and output interface O (a set of nodes), a body graph G and a pair i, o of injective mappings from I and O to the nodes of G. A detailed presentation of graphs with interfaces can be found in Section A.

Definition 4 (flow graph). *A* flow graph *(or briefly a* flow*) is a graph with interface* $I \xrightarrow{i} G \xleftarrow{o} O$*.*

Examples of flow graphs can be found in Fig. 1 and Fig. 9. The visual representation places the input and output interfaces to the left and to the right of the body graph, respectively. The mappings are denoted with dotted lines, while normal arrows are used for the edges. We use two sorts of nodes to distinguish principals (•) from variables (○). We neglect channels in our examples, for the sake of simplicty, so we do not use any specific node sort for them.

As we have seen in the case studies of Section 2 we sometimes distinguish different kinds of information flows. A direct flow between two entities is denoted by an arrow, but we sometimes use a specific terminology depending on the sort of the source and the target of an edge. More precisely, we call an edge between variables a *data flow*, an edge between processes *a control flow*, and edge from a variable to a process an *data-to-control flow* and vice versa for *control-to-data flows*. We also call *flow* to a path in a body graph. A path between variables is called *explicit flow* if it does not contain any control point. Otherwise it is called *implicit flow*. The input interface can be understood as the entities being used in the flow, while the output interface can be seen as the entities being provided by the flow. Note that the input and output mappings may not agree on a given

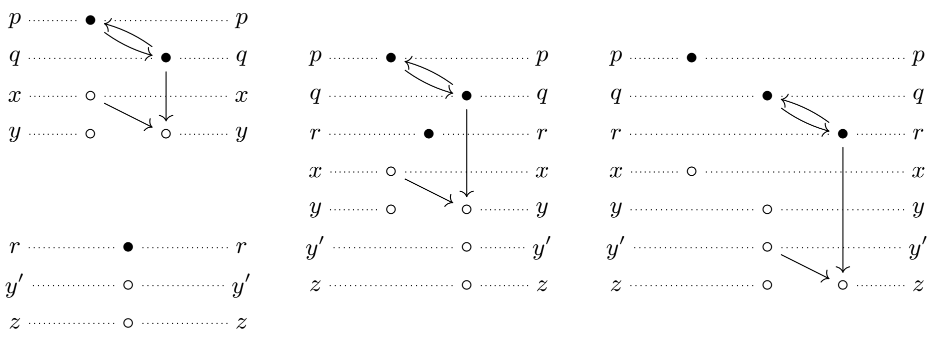

Fig. 9. Flows F_1 (top left), $\mathbf{Id}_{\{r,y',z\}}$ (bottom left), $F_1 \otimes \mathbf{Id}_{\{r,y',z\}}$ (mid) and $F2$ (right).

entity η, see e.g. how the overwrite of variable of z is modelled in the flow of Fig. 1.

In the following we denote the set of entities $I \cup O$ involved in a flow graph $F = I \xrightarrow{i} G \xleftarrow{o} O$ by $en(F)$. Our flow graphs can be related to standard concepts and terminology of information flow control. For instance, given a flow $F = I \xrightarrow{i} G \xleftarrow{o} O$, we can define the set of *influencers* of a set of entities $E \subseteq O$, denoted $\mathcal{I}_F(E)$, as $\{\eta \in I \mid i(\eta) \to_G^+ \eta' \wedge \eta' \in o(E)\}$, i.e. the set of input entities that have a flow towards some entity in E exposed in the output of F. Conversely, we can define the set of *readers* of a set of entities $E \subseteq I$, denoted $\mathcal{R}_F(E)$, as $\{\eta \in O \mid \exists \eta' \in i(E) \wedge \eta' \to_G^+ o(\eta)\}$, i.e. the set of output entities that have a flow from some entity in E exposed in the input of F. Here $u \to_G^+ v$ denotes that v is reachable from u through a path of positive length in graph G.

Flow graphs are equipped with suitable operations such as the empty flow $\mathbf{0}$, a family of identities $\mathbf{Id}_N$ indexed by a set of entities N (see e.g. $\mathbf{Id}_{\{r,y',z\}}$ in the bottom left of Fig. 9), a binary (associative, commutative) parallel composition operation $\otimes$ (see e.g. $F_1 \otimes \mathbf{Id}_{\{r,y',z\}}$ in the middle of Fig. 9) and a binary (associative) sequential composition operation $\circ$ (e.g. the flow graph in Fig. 1 can be obtained as the composition $(F_1 \otimes \mathbf{Id}_{\{r,y',z\}}) \circ F2$ of the flows in Fig. 9). A precise definition of those operations can be found in Section A (taken from [15]), which recasts the original presentations of [6,11] in the exact shape we need. The intuitive idea is that the sequential composition of a flow F with a flow G is the result of identifying the outputs of F with the inputs of G and merging their body graphs accordingly. The resulting graph has the input interface of F as input and the output of G as output. Instead, the parallel composition of a flow F with a flow G is the result of identifying inputs of F with inputs of G and outputs F with outputs G and merging their body graphs accordingly. The resulting graph has as input (resp. output) interface the union of the input (resp. output) interfaces of F and G.

$$\mathit{flows}(\alpha) = \mathit{lflow}(\alpha) \otimes \mathbf{I}_{\mathbf{Ent} \setminus en(\alpha)} \qquad \mathit{lflow}(\mathsf{skip}) = \mathbf{0}$$

$$\mathit{lflow}(e\texttt{@}p) = \mathbf{0} \qquad \mathit{lflow}(x := e\ \texttt{@}p) \qquad \mathit{lflow}(p.e\ \texttt{->}\ q.e' : k)$$

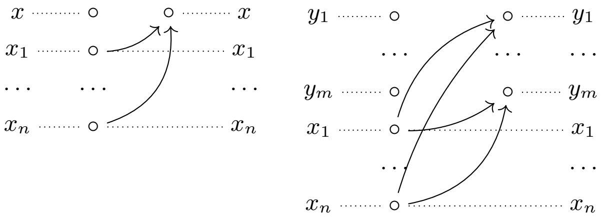

Fig. 10. Mapping of events into flows: explicit data flows

Instrumenting the Semantics. We denote the set of all flows by $\mathcal{F}$. The instrumentation of the semantics with flows is obtained by mapping actions in $\mathcal{A}$ into flows via some suitable function $\mathit{flows} : \mathcal{A} \to \mathcal{F}$.

Fig. 10 presents an example of such a mapping function, where $v(e) = \{x_1, \dots, x_n\}$ and $v(e') = \{y_1, \dots, y_m\}$. In particular, the function represents an annotation based on explicit data flows, where one is not interested in channels or principals, but just in direct transfer of data in interactions and assignments. The example function *flows* relies on a function *lflows* that defines the *local* flow of an event α. Such local flow is composed in parallel with $\mathbf{I}_{\mathbf{Ent} \setminus en(\alpha)}$, i.e. the identity on all entities not involved in α. The local flows for skip are defined to be the empty graph (no flow at all). It is worth remarking that in those flows the variables x or y_i can belong to $v(e)$ and hence coincide with some variable x_i. This kind of flow annotation can be useful, for instance, in the data providing case study of Section 2.1. We will refer to this flow annotation by flows_e.

Another example is depicted in Fig. 11. This flow annotation considers explicit data flows and some implicit data and control flows. For example, the local flows associated to conditions in decision points record the data-to-control flow from the free variables of e to the principal p and the data flow to variable x. Moreover, the flows for assignments and interactions are similar with the difference being that in interactions we record the mutual control flow between the interacting principals. We call this flow annotation function flows_e.

Yet another example can be found in Fig. 12. In this case, that we will refer to as flows_a, we are interested in flows related to how processes directly access data by either reading or writing variables. Note that, contrary to the previous cases, we are not interested in observing the fact that a variable has been overwritten. This is the kind of flow annotation that would make sense in the example we saw in Section 2.2, related to the MapReduce computation, where one is interested in controlling the locality of data accesses.

$$\mathit{flows}(\alpha) = \mathit{lflow}(\alpha) \otimes \mathbf{I}_{\mathbf{Ent} \setminus en(\alpha)} \qquad \mathit{lflow}(\mathsf{skip}) = \mathbf{0}$$

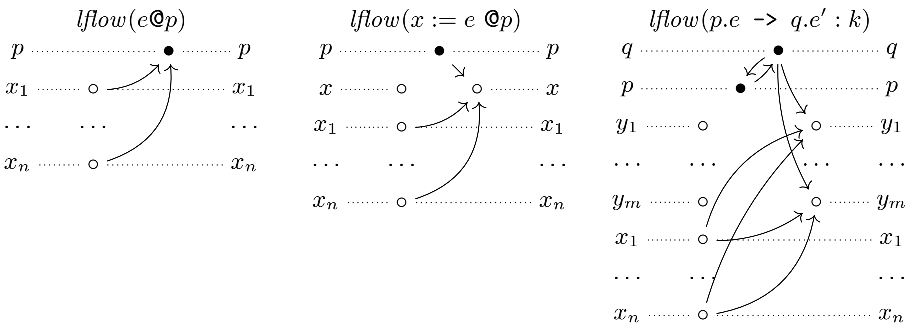
$\mathit{lflow}(e \texttt{@} p)$
p
x_1
$\cdots$
x_n
$\mathit{lflow}(x := e \ \texttt{@} p)$
p
x
x_1
$\cdots$
x_n
$\mathit{lflow}(p.e \ \texttt{->} \ q.e' : k)$
q
p
y_1
$\cdots$
y_m
x_1
$\cdots$
x_n

Fig. 11. Mapping of events into flows: explicit and implicit flows

As we have seen, our approach provides some flexibility in the definition of flow annotations. However, the soundness of our approach relies on some *well-formedness* restrictions of those annotations.

Definition 5. *(well-formed flow annotation) A flow annotation function flows* : $\mathcal{A} \to \mathcal{F}$ *is* well-formed *iff* $\forall \alpha \in \mathcal{A} : \mathit{flows}(\alpha) = F \otimes \mathbf{I}_{\mathbf{Ent} \setminus en(F)} \wedge en(F) \subseteq en(\alpha)$.

Intuitively, the idea is that a well-formed flow annotation associates a flow to an event α which is composed by an arbitrary flow F on some entities occurring in α and the identity on all other entities not occurring in F. It is easy to see that the flow annotations of Fig. 10, 11 and 12 are well-formed. Note that well-formedness also forbids the introduction of fresh entities outside **Ent** in the interface of the defined flows. Of course, flow annotations are a doubly-sharped mechanism: it provides a lot of flexibility to the information flow engineer, but it also discharges on him the responsibility of specifying the local flows of interest. As an extreme case consider that a flow annotation could be just defined as the constant $\mathbf{Id}_{\mathbf{Ent}}$. In this case, no flow would be observed and all policies would be satisfied.

From now on, we restrict our attention to well-formed annotation functions. Given a well-formed flow annotation function *flows* the flow-instrumentation of the operational semantics of our language can be obtained as the relation $\to \subseteq \mathcal{C} \times \mathcal{F} \times \mathcal{C}$ defined as $\{C \xrightarrow{\mathit{flows}(\alpha)} C' \mid C \xrightarrow{\alpha} C'\}$. Similarly, the traces defined by the denotational semantics can be transformed into flows by sequentially composing the flows associated to the events of a trace.

Definition 6 (trace flows). *The flows of a trace σ, denoted $\mathit{flows}(\sigma)$, is given by function flows* : $\mathcal{A}^* \to \mathcal{F}$ *defined as*

$$\mathit{flows}(\epsilon) = \mathbf{Id}_{\mathbf{Ent}} \qquad \mathit{flows}(\sigma\sigma') = \mathit{flows}(\sigma) \circ \mathit{flows}(\sigma')$$

$$\mathit{flows}(\alpha) = \mathit{lflow}(\alpha) \otimes \mathbf{I}_{\mathbf{Ent}\setminus en(\alpha)} \qquad \mathit{lflow}(\mathsf{skip}) = \mathbf{0}$$

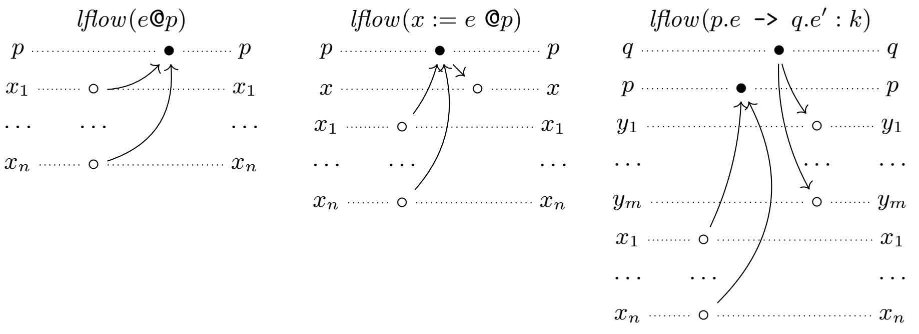
$\mathit{lflow}(e\,@p)$ $\mathit{lflow}(x := e\ @p)$ $\mathit{lflow}(p.e\ \texttt{->}\ q.e' : k)$

Fig. 12. Mapping of events into flows: data access flows

The above definition does not define $\mathit{flows}(\alpha)$ to provide the afore mentioned flexibility in the observation of flows in events. Function *flows* is lifted to set of traces T and choreographies C in the obvious way, i.e. $\mathit{flows}(T) = \{\mathit{flows}(\sigma) \mid \sigma \in T\}$ and $\mathit{flows}(C) = \mathit{flows}(\mathit{Traces}(C))$. As a simple example, the flow in Fig. 1 represents the flow of the only trace of the choreography $\mathsf{C1}$ discussed in the Introduction obtained with the mapping of events into flows defined in Fig. 11.

5 Typing Choreographies

We represent information flow policies with flow graphs with the idea that a flow graph denotes *all* flows that are allowed in a system.

Definition 7 (information flow policy). *An information flow policy Π is a graph with interface* $\mathbf{Ent} \xrightarrow{i} G \xleftarrow{o} \mathbf{Ent}$. *The set of all policies is denoted by* $\mathcal{P}$.

It is worth to note that we require the input and output of a policy to coincide with the set of all entities **Ent** of interest. We call a policy *coherent* if $i = o$, i.e. if all entities mapped to the same node in the body by both input and output mappings. Note that when a policy $F = \mathbf{Ent} \xrightarrow{i} G \xleftarrow{o} \mathbf{Ent}$ agrees in its input and output interfaces, we can rename some nodes in the body G with their (unique) images in the interface, possibly after alpha-renaming some internal nodes to avoid name clashes. This way we can provide the more compact notation that we use some of our figures. An additional simplification that we do in our graphical notation is that if a flow or policy is the identity on some entity η, then we neglect η in the visual notation. For instance, in all our examples we neglect the channels used in the choreographies.

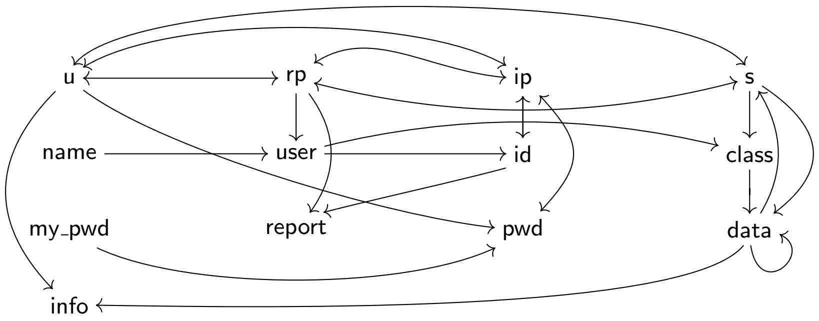

Fig. 13. Policy $\Pi_{s,2}$ in the Data Providing Service

We have already seen some examples of policies in Section 2, namely in Fig. 3 and 6. An additional example can be found in Fig. 13. It specifies a policy for the data providing case study which extends the policy of Fig. 3 to implicit flows. Figure 14, instead, provides two additional policies for the MapReduce case study. The one on the left focuses on control flows and may be useful to control how principals depend on each other, e.g. in case of failure. The one on the right is oriented to explicit data flows.

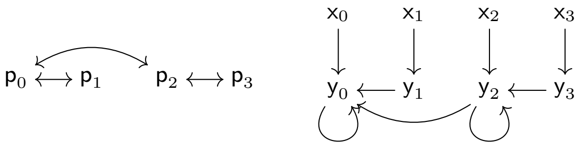

Fig. 14. Policies $\Pi_{m,2}$ (left) and $\Pi_{m,3}$ (right) for MapReduce

The following definition formalizes the notion of satisfaction of an information flow policy by a choreography, resp. a set of traces, a *decomposable* trace, an *atomic* trace, and an information flow. Here, a *decomposable* refers to the fact that we are interested in observing all components (i.e. subtraces) of a trace and *atomic* refers to the fact that we want to observe the trace as a whole.

$$\frac{\mathbf{Ent} \vdash C_1 : \Pi \quad \mathbf{Ent} \vdash C_2 : \Pi}{\mathbf{Ent} \vdash C_1 ; C_2 : \Pi} \qquad \frac{\mathbf{Ent} \vdash C_1 : \Pi \quad \mathbf{Ent} \vdash C_2 : \Pi}{\mathbf{Ent} \vdash C_1 \mid C_2 : \Pi} \qquad \frac{A \models \Pi}{\mathbf{Ent} \vdash A : \Pi}$$

$$\frac{e@p \models \Pi \quad \mathbf{Ent} \vdash C_1 : \Pi \quad \mathbf{Ent} \vdash C_2 : \Pi}{\mathbf{Ent} \vdash \mathsf{if}\ e@p\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : \Pi} \qquad \frac{e@p \models \Pi \quad \mathbf{Ent} \vdash C_1 : \Pi \quad \mathbf{Ent} \vdash C_2 : \Pi}{\mathbf{Ent} \vdash \mathsf{while}\ e@p\ \mathsf{do}\ C_1\ \mathsf{then}\ C_2 : \Pi}$$

Fig. 15. Type system

Definition 8 (policy satisfaction). *The set of policy satisfaction relations* $\_ \models \_ : \mathcal{C} \cup 2^{\mathcal{A}^*} \cup \mathcal{A}^* \cup \mathcal{F} \to \mathcal{P}$, $\_ \vdash \_ : \mathcal{A}^* \to \mathcal{P}$, *is defined by:*

$$\begin{array}{rll}
C \models \Pi & \textit{iff} & \mathit{Traces}(C) \models \Pi \\
T \models \Pi & \textit{iff} & \forall \sigma \in T : \sigma \models \Pi \\
\sigma \models \Pi & \textit{iff} & \sigma = \sigma'' \sigma' \sigma''' \Rightarrow \sigma' \vdash \Pi \\
\sigma \vdash \Pi & \textit{iff} & \mathit{flows}(\sigma) \models \Pi \\
F \models \Pi & \textit{iff} & \forall \eta, \eta' \in \mathit{en}(\Pi) : i_F(\eta) \to^*_{G_F} o_F(\eta') \ \Rightarrow \ i_\Pi(\eta) \to^*_{G_\Pi} o_\Pi(\eta')
\end{array}$$

Intuitively, the idea is that a choreography satisfies a policy Π if all its traces satisfy the policy Π. A trace σ satisfies Π if no subtrace of σ introduces a flow from an entity η to an entity η' that is not allowed in Π.

Our type system statically checks if a choreography C satisfies a policy Π. Our types are thus policies and our type judgements are of the form $\mathbf{Ent} \vdash C : \Pi$. The type system is sound for policies *coherent* policies, i.e. policies whose input and output interfaces agree.

Definition 9 (type system). *The type system for judgements* $\mathbf{Ent} \vdash C : \Pi$ *is defined by the rules of Fig. 15.*

The main result of our work is the soundness of the type system (cf. Th 1). In order to prove such result we have first to prove the following lemma that formalises the fact that trace concatenation preserves the satisfaction of policies.

Lemma 1 (concatenation). *Let* $\sigma_1, \sigma_2 \in \mathcal{A}^*$ *be two traces and* $\Pi \in \mathcal{P}$ *a coherent policy. If* $\sigma_1 \models \Pi$ *and* $\sigma_2 \models \Pi$ *then* $\sigma_1 \sigma_2 \models \Pi$.

Proof. The proof is by induction on the length of σ_1 and σ_2.

[$\sigma_1 = \epsilon$ or $\sigma_2 = \epsilon$] These are trivial cases.

[$\sigma_1 = \alpha_1$ and $\sigma_2 = \alpha_2$, $\alpha_i \in \mathcal{A}$]. To prove $\sigma_1 \sigma_2 \models \Pi$ we have to show that $\sigma \vdash \Pi$ for all subtraces of $\alpha_1 \alpha_2$. Those are four: ϵ, α_1, α_2 and $\alpha_1 \alpha_2$. The first three cases are trivial. The interesting case is the latter. The proof is by contradiction. Suppose that $\alpha_1 \alpha_2 \not\models \Pi$. This means that there exist two different entities $\eta, \eta' \in en(\Pi)$ such that $\eta \in \mathcal{R}_{\mathit{flows}(\alpha_1\alpha_2)}(\eta')$ but $\eta \notin \mathcal{R}_{\Pi}(\eta')$. Since $\alpha_i \models \Pi, i \in \{1,2\}$ we know that $\eta \notin \mathcal{R}_{\mathit{flows}(\alpha_i)}(\eta')$. Hence, the flow from η' to η must have been introduced in the composition of the flows of α_1 and α_2. There must exist an entity η'' such that $\eta'' \in \mathcal{R}_{\mathit{flows}(\alpha_1)}(\eta')$ and $\eta'' \in \mathcal{I}_{\mathit{flows}(\alpha_2)}(\eta)$. But since $\eta'' \in \mathbf{Ent}$ (by well-formedness of *flows*) and $\alpha_i \models \Pi, i \in \{1,2\}$ it must be the case that $\eta'' \in \mathcal{R}_{\Pi}(\eta')$ and $\eta'' \in \mathcal{I}_{\Pi}(\eta)$. Moreover, since Π s coherent, we know that $i(\eta'') = o(\eta'')$. Hence Π must be such that $\eta \in \mathcal{R}_{\Pi}(\eta')$. This is a contradiction. Hence, $\sigma_1\sigma_2 \vdash \Pi$. Since we have shown that all subtraces of $\alpha_1\alpha_2$ satisfy Π we can conclude that $\sigma_1\sigma_2 \models \Pi$.

$[\sigma_1 = \alpha_1\sigma_1',\ \alpha_1 \in \mathcal{A}]$ We have to show that $\sigma \vdash \Pi$ for all subtraces σ of $\alpha_1\sigma_1'\sigma_2$. We distinguish two cases (i) traces of the form σ' with $\sigma'\sigma'' = \sigma_1\sigma_2$ and (ii) traces of the form $\alpha_1\sigma'$ with $\sigma'\sigma'' = \sigma_1\sigma_2$. Consider case (i) first. We know that $\sigma_1' \models \Pi$ and $\sigma_2 \models \Pi$. By induction, we can conclude that $\sigma_1'\sigma_2 \models \Pi$. Hence all sub-traces of $\sigma_1\sigma_2$ not starting with α_1 satisfy Π. Consider now case (ii), i.e. traces of the form $\alpha_1\sigma'$ with $\sigma'\sigma'' = \sigma_1\sigma_2$. Again, we can apply induction since $\alpha_1 \models \Pi$ and $\sigma' \models \Pi$ (by case i). Thus we conclude that $\sigma_1\sigma_2 \models \Pi$.

□

Theorem 1 (soundness). *Let $C \in \mathcal{C}$ be a well-formed choreography and $\Pi \in \mathcal{P}$ be a coherent policy. If $\mathbf{Ent} \vdash C : \Pi$ then $C \models \Pi$.*

Proof. The proof is by induction on the structure of C.

$[C = A]$ This case is trivial since $A \models \Pi$ is precisely the premise for typing A.

$[C = C_1; C_2]$ By definition, every trace of $C_1; C_2$ is of the form $\sigma_1\sigma_2$ with $\sigma_i \in \mathit{Traces}(C_i), i \in \{1,2\}$. We have that $\sigma_i \models \Pi, i \in \{1,2\}$ by induction since the typing rules for $C_1; C_2$ require $\mathbf{Ent} \vdash C_i : \Pi, i \in \{1,2\}$. Hence, we can apply Lemma 1 to conclude that $\sigma_1\sigma_2 \models \Pi$.

$[C = C_1 \mid C_2]$ The proof of this case relies on well-formedness, which allows us to transform every trace $\sigma \in \mathit{Traces}(C_1 \mid C_2)$ into a trace σ' in one of the forms used in the above case, while having $\mathit{flows}(\sigma) = \mathit{flows}(\sigma')$.

We start proving that the transformation is possible and later prove that it indeed preserves the flows. Every trace $\sigma \in \mathit{Traces}(C_1 \mid C_2)$ is of the form $\sigma_{1,1}\sigma_{1,2} \ldots \sigma_{n,1}\sigma_{n,2}$ with $\sigma_{1,i} \ldots \sigma_{n,i} \in \mathit{Traces}(C_i), i \in \{1,2\}$. It is easy to see if σ is of the form $\sigma'\sigma_{k,2}\sigma_{k+1,1}\sigma''$ with $\sigma_{k,2} = \sigma_{k,2}'\alpha$ and $\sigma_{k+1,1} = \beta\sigma_{k+1,1}'$, with $\alpha, \beta \in \mathcal{A}$, we can transform σ into trace $\sigma''' = \sigma'\sigma_{k,2}'\beta\alpha\sigma_{k+1,1}'\sigma''$ which belongs to $\mathit{Traces}(C_1 \mid C_2)$ by conditions (1) and (2) of well-formedness. Indeed, condition (1) requires α and β two involve disjoint entities ($en(\alpha) \cap en(\beta)$) and (2) ensures that transposing α and β in σ yields a trace that belongs to $\mathit{Traces}(C_1 \mid C_2)$.

It remains to show that σ and σ''' have the same flows. The main idea is that $\mathit{flows}(\alpha\beta) = \mathit{flows}(\beta\alpha)$ since $en(\alpha) \cap en(\beta) = \emptyset$. This can be shown as follows

$$
\begin{aligned}
\mathit{flows}(\alpha\beta) &= \mathit{flows}(\alpha) \circ \mathit{flows}(\beta) \\
&= (F \otimes \mathbf{Id}_{\mathbf{Ent}\setminus en(F)}) \circ (G \otimes \mathbf{Id}_{\mathbf{Ent}\setminus en(G)}) && \text{(well-formed } \mathit{flows}\text{)} \\
&= \begin{pmatrix} F \\ \otimes \\ \mathbf{Id}_{\mathbf{Ent}\setminus(en(F)\cup en(G))} \\ \otimes \\ \mathbf{Id}_{en(G)} \end{pmatrix} \circ \begin{pmatrix} \mathbf{Id}_{en(F)} \\ \otimes \\ \mathbf{Id}_{\mathbf{Ent}\setminus(en(F)\cup en(G))} \\ \otimes \\ G \end{pmatrix} && \text{(since } en(F) \cap en(G) = \emptyset\text{)} \\
&= \begin{pmatrix} (F & \circ & \mathbf{Id}_{en(F)}) \\ & \otimes & \\ (\mathbf{Id}_{\mathbf{Ent}\setminus(en(F)\cup en(G))} & \circ & \mathbf{Id}_{\mathbf{Ent}\setminus(en(F)\cup en(G))}) \\ & \otimes & \\ (\mathbf{Id}_{en(G)} & \circ & G) \end{pmatrix} && \text{(distribution)} \\
&= F \otimes \mathbf{Id}_{\mathbf{Ent}\setminus(en(F)\cup en(G))} \otimes G && \text{(identity)} \\
&= G \otimes \mathbf{Id}_{\mathbf{Ent}\setminus(en(F)\cup en(G))} \otimes F && \text{(commutativity)}
\end{aligned}
$$

from which we can apply the same equations (upwards, replacing G by F) to obtain $\mathit{flows}(\beta\alpha)$.

Applying the above described transpositions in σ as much as needed results in σ being rewritten into $\sigma_1\sigma_2$ with $\sigma_i \in \mathit{Traces}(C_i), i \in \{1,2\}$. Since $\mathbf{Ent} \vdash C_i : \Pi, i \in \{1,2\}$, the proof schema based on the application of Lemma 1 used in the above case can then be applied to conclude that $C \models \Pi$.

$[C = \mathsf{if}\ e@p\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]$ By definition, every trace σ in $\mathit{Traces}(\mathsf{if}\ e@p\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2)$ is of the form $e@p\ \sigma'$ with $\sigma' \in \mathit{Traces}(C_i), i \in \{1,2\}$. The typing rule for C requires $e@p \models \Pi$ and $\mathbf{Ent} \vdash C_i : \Pi, i \in \{1,2\}$. By induction we have $C_i \models \Pi, i \in \{1,2\}$ so that we can apply the Lemma 1 to conclude that $e@p\ \sigma' \models \Pi$.

$[C = \mathsf{while}\ e@p\ \mathsf{do}\ C_1\ \mathsf{then}\ C_2]$ This case is similar to the above one.

□

Let us now consider the choreographies P and $\mathsf{M}_{0,2}$ of our case studies, the flow annotation functions flows_e, flows_i, and flows_a and the policies $\Pi_{s,1}$, $\Pi_{s,2}$, $\Pi_{m,1}$, $\Pi_{m,2}$ and $\Pi_{s,3}$. For ease of notation, let us use the notation $C \models_f \Pi$ to refer to $C \models \Pi$ when flows is f.

One can easily see that the following satisfaction statements regarding the choreography P of the identity-secured data providing service can be concluded from our type system: $\mathsf{P} \models_{\mathit{flows}_e} \Pi_{s,1}$, $\mathsf{P} \models_{\mathit{flows}_e} \Pi_{s,2}$, and $\mathsf{P} \models_{\mathit{flows}_i} \Pi_{s,2}$. An easy way to see this is to note that all events in the choreography (actions and expressions used in the branching statements) satisfy the corresponding policy. Instead the statement $\mathsf{P} \models_{\mathit{flows}_i} \Pi_{s,1}$ cannot be concluded. As a matter of fact, policy $\Pi_{s,1}$ is not satisfied by P, since $\Pi_{s,1}$ does not allow implicit flows, that P actually has. For instance, the flow annotation function flows_i would reveal an implicit flow from my_pwd to class that is not allowed by the policy $\Pi_{s,1}$.

Regarding the MapReduce case study, we can conclude, for instance that $\mathsf{M}_{0,2} \models_{\mathit{flows}_a} \Pi_{m,1}$, $\mathsf{M}_{0,2} \models_{\mathit{flows}_e} \Pi_{m,3}$, and $\mathsf{M}_{0,2} \models_{\mathit{flows}_i} \Pi_{m,3}$. However, we cannot conclude that $\mathsf{M}_{0,2} \models_{\mathit{flows}_i} \Pi_{m,2}$ since flows_i requires us to observe data flows that $\Pi_{m,2}$ does not allow. A flow annotation function like flows_e but limited to control flows would then allow us to check the policy.

$$\mathit{flows}(\alpha) = \mathit{lflow}(\alpha) \otimes \mathbf{I}_{\mathbf{Ent} \setminus en(\alpha)} \qquad \mathit{lflow}(\mathsf{skip}) = \mathbf{0}$$

$$\mathit{lflow}(e @ p) = \mathbf{0} \quad \mathit{lflow}(x := e\ @ p) = \mathbf{0} \qquad \mathit{lflow}(p.e \texttt{ -> } q.e' : k)$$

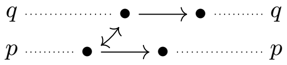

Fig. 16. Mapping of events into flows: control flows with temporal dependencies

Actually, the policy $\Pi_{m,2}$ is more permissive than it could be. It allows one to have dependencies between all principals, including p_1 and p_3, whose respective controls do not actually depend on each other in P. A more relaxed policy forbidding control flow dependencies between p_1 and p_3 can be found in Fig. 17. Contrary to all policies we have presented so far, policy $\Pi_{m,4}$ is not coherent, i.e. it does not agree in the interface and, thus, our type system cannot be applied. That is, some entities in the interface correspond to different nodes in the body graph. This is essential to capture some information about the temporal order of flows. It is easy to check that the policy allows flows from p_1 to p_0 and p_2 but not to p_3.

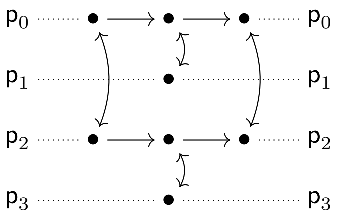

Fig. 17. Policy $\Pi_{m,4}$

To check policies like $\Pi_{m,4}$ we would have to consider the richer control flow annotation flows_d of Fig. 16 to avoid spurious flows. Indeed, flow annotation functions like flows_e abstract away from temporal information in the control flow. Such abstractions introduce spurious flows between, for instance, p_1 and p_3 whose respective controls do not depend on each other. Instead, flows_d records some basic information about the temporal order of interactions. Fig. 18 illustrates the flow of all maximal traces of choreography P. The policy $\Pi_{m,4}$ would be satisfied. For example, the above mentioned spurious flows between p_1 and p_3 are not present now. Indeed, p_1 interacts directly with p_0 and indirectly with p_2 but only *after* p_2 has finished interacting with p_3. Extending our type system to deal with such policies is subject of current work.

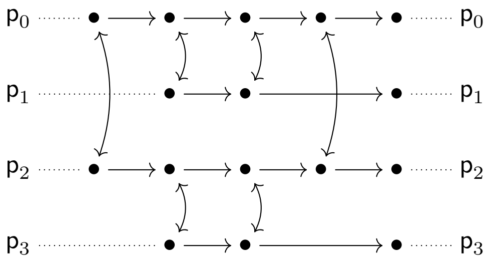

Fig. 18. Control dependencies in MapReduce

6 Related Works and Conclusion

The use of typing disciplines for security systems has a long tradition. The recent years have seen an increasing interest in applying techniques based on behavioural types to security analysis. We refer to [2] for a comprehensive survey and limit our discussion to the most relevant and recent works in that area.

A first work worth mentioning is [7], which presents an approach for dealing with non-interference properties in distributed systems where components interact within multiparty sessions. Systems are described with a session calculus featuring, among others, session creation, inter-session interaction, and session delegation. The approach includes a session type system whose rules include information flow requirements to ensure both behavioural and non-interference properties. Another relevant work is presented in [14]. The work focuses on *data provenance*, i.e. the problem of keeping track of how data flow and are processed. The authors present a calculus to describe how processes use, consume and publish linked data. The approach is equipped with type systems for the calculus to ensure mandatory access control properties based on security levels [14] and role-based access control properties [13].

There are two main differences between the above discussed works and our own work. The first one is our focus on *discretionary* information flow, instead of other forms of information flow control (mandatory, role-based, non-interference, etc.). A second difference lies in the specification languages used; our choreography description language differs from the calculi used in the above mentioned works. It would not be trivial, for instance, to integrate our type system with the type system of [7] as both focus on related but significantly different languages and properties. We would also like to remark that our work is still in a preliminary phase and does not yet consider aspects of choreography realizability, local projections and systems with multiple-sessions.

As a matter of fact, those aspects are part of our research agenda. We are currently developing a suitable notion of local projections and a type system that ensure that projections enjoy suitable semantic relations with choreographies, from which we can conclude that local projections do not introduce flows not specified in the choreography. The main contribution of the paper would then be applied to ensure Π-secure distributed implementations.

We also plan to investigate the development of our approach in several directions, including the possibility to specify and check temporal aspects by extending our type system beyond coherent policies. We would also like to develop an inference system to compute of over- and under-approximation of information flows, and to consider of *intransitive* information flow properties. On the application side, we plan to investigate the suitability of our approach to popular parallel programming frameworks such as MPI. There is indeed an urgent demand of formal guarantees for systems developed in such frameworks [18] and some flow analysis works already exist (e.g. [5,1]) also based on behavioural types (e.g. [17]).

Acknowledgement. We would like to dedicate this work to José Meseguer, for his inspiring and influencing works in the areas of Concurrency Theory, Algebraic Specifications and Security. We hope that José will understand (and forgive!) the absence of a non-interference result in our work. The first author would like to express his gratitude to José for honouring him with his friendship and giving him the unique experience of scientific collaboration. The first author is also grateful to Fabrizio Montesi, Emilio Tuosto and Marco Carbone for fruitful feedback on early versoins of this work, and to the organizers of the BETTY COST Action for their gentle invitation to present an early version of the work in a meeting.

References

1. Aananthakrishnan, S., Bronevetsky, G., Gopalakrishnan, G.: Hybrid approach for data-flow analysis of MPI programs. In: Malony, A.D., Nemirovsky, M., Midkiff, S.P. (eds.) International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013. pp. 455–456. ACM (2013), `http://doi.acm.org/10.1145/2464996.2467286`
2. Bartoletti, M., Castellani, I., Denielou, P.M., Dezani-Ciancaglini, M., Ghilezan, S., Pantovic, J., Pérez, J.A., Thiemann, P., Toninho, B., Vieira, H.T.: Combining behavioural types with security analysis, state of The Art Report of WG2 – European Cost Action IC1201 BETTY (Behavioural Types for Reliable Large-Scale Software Systems), available at `http://www.behavioural-types.eu/publications/WG2-State-of-the-Art.pdf/at_download/file`
3. Bocchi, L., Melgratti, H.C., Tuosto, E.: Resolving non-determinism in choreographies. In: Shao, Z. (ed.) 23rd European Symposium on Programming Languages and Systems (ESOP 2014). Lecture Notes in Computer Science, vol. 8410, pp. 493–512. Springer (2014), `http://dx.doi.org/10.1007/978-3-642-54833-8_26`
4. Boudol, G.: Secure information flow as a safety property. In: Degano, P., Guttman, J.D., Martinelli, F. (eds.) 5th International Workshop on Formal Aspects in Security and Trust (FAST 2008). Lecture Notes in Computer Science, vol. 5491, pp. 20–34. Springer (2008), `http://dx.doi.org/10.1007/978-3-642-01465-9_2`
5. Bronevetsky, G.: Communication-sensitive static dataflow for parallel message passing applications. In: Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009. pp. 1–12. IEEE Computer Society (2009), `http://dx.doi.org/10.1109/CGO.2009.32`
6. Bruni, R., Gadducci, F., Montanari, U.: Normal forms for algebras of connection. Theor. Comput. Sci. 286(2), 247–292 (2002), `http://dx.doi.org/10.1016/S0304-3975(01)00318-8`
7. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Typing access control and secure information flow in sessions. Inf. Comput. 238, 68–105 (2014), `http://dx.doi.org/10.1016/j.ic.2014.07.005`
8. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013). pp. 263–274. ACM (2013), `http://doi.acm.org/10.1145/2429069.2429101`
9. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multiparty session. Logical Methods in Computer Science 8(1) (2012), `http://dx.doi.org/10.2168/LMCS-8(1:24)2012`

10. Cohen, E.: Information transmission in computational systems. In: Sixth ACM Symposium on Operating Systems Principles. pp. 133–139. SOSP '77, ACM, New York, NY, USA (1977), `http://doi.acm.org/10.1145/800214.806556`
11. Corradini, A., Gadducci, F.: An algebraic presentation of term graphs, via gs-monoidal categories. Applied Categorical Structures 7(4), 299–331 (1999), `http://dx.doi.org/10.1023/A:1008647417502`
12. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
13. Dezani-Ciancaglini, M., Ghilezan, S., Jaksic, S., Pantovic, J.: Types for role-based access control of dynamic web data. In: Mariño, J. (ed.) 19th International Workshop on Functional and Constraint Logic Programming (WFLP 2010). Lecture Notes in Computer Science, vol. 6559, pp. 1–29. Springer (2010), `http://dx.doi.org/10.1007/978-3-642-20775-4_1`
14. Dezani-Ciancaglini, M., Horne, R., Sassone, V.: Tracing where and who provenance in linked data: A calculus. Theor. Comput. Sci. 464, 113–129 (2012), `http://dx.doi.org/10.1016/j.tcs.2012.06.020`
15. Gadducci, F.: Graph rewriting for the pi-calculus. Mathematical Structures in Computer Science 17(3), 407–437 (2007), `http://dx.doi.org/10.1017/S096012950700610X`
16. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20 (1982)
17. Gopalakrishnan, G., Kirby, R.M., Siegel, S.F., Thakur, R., Gropp, W., Lusk, E.L., de Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs. Commun. ACM 54(12), 82–91 (2011), `http://doi.acm.org/10.1145/2043174.2043194`
18. Honda, K., Marques, E.R.B., Martins, F., Ng, N., Vasconcelos, V.T., Yoshida, N.: Verification of MPI programs using session types. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) 19th European MPI Users' Group Meeting on Recent Advances in the Message Passing Interface (EuroMPI 2012). Lecture Notes in Computer Science, vol. 7490, pp. 291–293. Springer (2012), `http://dx.doi.org/10.1007/978-3-642-33518-1_37`
19. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008). pp. 273–284. ACM (2008), `http://doi.acm.org/10.1145/1328438.1328472`
20. Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the gap between interaction- and process-oriented choreographies. In: Cerone, A., Gruner, S. (eds.) Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008. pp. 323–332. IEEE Computer Society (2008), `http://dx.doi.org/10.1109/SEFM.2008.11`
21. Lohmann, N., Wolf, K.: Decidability results for choreography realization. In: Kappel, G., Maamar, Z., Nezhad, H.R.M. (eds.) 9th International Conference on Service-Oriented Computing (ICSOC 2011). Lecture Notes in Computer Science, vol. 7084, pp. 92–107. Springer (2011), `http://dx.doi.org/10.1007/978-3-642-25535-9_7`
22. Montesi, F.: Choreographic Programming. Ph.D. thesis, IT University of Copenhagen (2013), `http://www.fabriziomontesi.com/files/m13_phdthesis.pdf`
23. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: SOSP. pp. 129–142 (1997), `http://doi.acm.org/10.1145/268998.266669`
24. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999)

A Graphs with Interfaces

We recall and adapt a few definitions concerning graphs, and their extension with *interfaces*, referring to [15,6,11] for a more detailed presentation.

Definition 10 (graphs). *A is a four-tuple* $\langle V, E, s, t\rangle$ *where* V *is the set of nodes,* E *is the set of edges and* $s, t : E \to V$ *are the source and target functions. A graph morphism is a pair of functions* $\langle f_V, f_E\rangle$ *preserving the source and target functions, i.e.* $f_V \circ s = s \circ f_E$ *and* $f_V \circ t = t \circ f_E$*.*

Definition 11 (graphs with interfaces). *A* graph with interfaces *is a span of graph morphisms* $I \xrightarrow{i} G \xleftarrow{o} O$*, where* G *is a* body *graph,* I *and* O *are the* input *and* output graph interfaces*, and* $i : I \to G$*,* $o : O \to G$ *are the* input *and* output graph morphisms*. An* interface graph morphism $f : \mathbb{G} \Rightarrow \mathbb{H}$ *is a triple of graph morphisms* $\langle f_I, f, f_O\rangle$*, preserving the input and output morphisms.*

With an abuse of notation, we sometimes refer to the image of the input and output morphisms as inputs and outputs, respectively, and we use the term *graph* to refer to abbreviate *graphs with interfaces*. We restrict our attention to graphs with *discrete* interfaces, i.e., such that their set of edges is empty.

The following definitions define the sequential and parallel composition of graphs with interfaces, whose informal description can be found in Section 4.

Definition 12 (sequential composition of graphs). *Let* $\mathbb{G} = I \xrightarrow{i} G \xleftarrow{j} J$ *and* $\mathbb{G}' = J \xrightarrow{j'} G' \xleftarrow{o} O$ *be graphs with interfaces. Then, their* sequential composition *is the graph* $\mathbb{G} \circ \mathbb{G}' = I \xrightarrow{i'} G'' \xleftarrow{o'} O$*, for* G'' *the disjoint union* $G \uplus G'$*, modulo the equivalence on nodes induced by* $j(x) = j'(x)$ *for all* $x \in N_J$*, and* i', o' *the uniquely induced arrows.*

Definition 13 (parallel composition of graphs). *Let* $\mathbb{G} = I \xrightarrow{i} G \xleftarrow{o} O$ *and* $\mathbb{H} = I' \xrightarrow{i'} H \xleftarrow{o'} O'$ *be two graphs with interfaces. Then, their* parallel composition *is the* GWDI $\mathbb{G} \otimes \mathbb{H} = (I \cup I') \xrightarrow{i''} G' \xleftarrow{o''} (O \cup O')$*, for* G' *the disjoint union* $G \uplus H$*, modulo the equivalence on nodes induced by* $o(y) = o'(y)$ *for all* $y \in N_O \cap N_{O'}$ *and* $i(y) = i'(y)$ *for all* $y \in N_I \cap N_{I'}$*, and* i'', o'' *the uniquely induced arrows.*

With an abuse of notation, the set-theoretic operators are defined componentwise. The operations are concretely defined, modulo the choice of canonical representatives for the set-theoretic operations: the result is independent of such a choice, up-to isomorphism of the body graphs.

A *graph expression* is a term over the syntax containing all graphs with discrete interfaces as constants, and parallel and sequential composition as binary operators. An expression is *well-formed* if all occurrences of those operators are defined for the interfaces of their arguments, according to Definitions 12 and 13; its interfaces are computed inductively from the interfaces of the graphs occurring in it, and its *value* is the graph obtained by evaluating all operators in it. For the axiomatic properties of the operations (e.g. associativity, identity of $\circ$, associativity, commutativity and identity of $\otimes$) we refer to [6,11].