Session Types for Orchestration Charts

Alessandro Fantechi[1] and Elie Najm[2]

[1] Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze, Firenze, Italy
fantechi@dsi.unifi.it
[2] Telecom ParisTech (ENST)
46 rue Barrault, F-75013 Paris, France
elie.najm@telecom-paristech.fr

Abstract. We present a novel approach for the sound orchestration of services. It is based on Orcharts and Typecharts: a service orchestration language and an associated behavioural typing language. Sessions play a pivotal role in this approach. Orcharts (orchestration charts) define session based services and Typecharts provide for session types with complex interaction patterns that generalise the request/response interaction paradigm. We provide an algorithm for deciding behavioural well typedeness. We claim that well typed service configurations have the soudness property, i.e., any session that can be initiated in a well typed configuration has its requestor and provider behave in mutual conformance and potentially reach service completion.

1 Introduction

Behavioural type systems have been defined in recent years with the aim to be able to check the compatibility of communicating components, not only regarding data exchanged, but also regarding the matching of their respective behaviour [14,8,13]. Recently, the focus moved from components to service-oriented architectures, and several calculi for *service orchestration* have been defined. Of them, Orc [7] uses few simple orchestration mechanisms but shows a very interesting expressive power. In this language, an invoked service provides a simple reply which can be piped to trigger other invocations. Thus, interface compatibility looses its interest because invocations which are not replied or replies which are not listened at are simply lost, with no possible identification of error states.

Although Orc is able to encode most common workflow patterns [5], the simplicity of the language is felt unsatisfactory for dealing with complex services in which different invocations of a service can trigger complex interaction patterns among several services. Often an interaction pattern constitute a *session* which clearly identifies which are the message exchanges belonging to the session. Session types, that is, behavioural types associated to sessions, have been studied for protocols [6] and software components [15]. Service orchestration calculi including the notion of session have also been defined [3,9].

A different approach can be chosen for relating messages of a complex interaction pattern: message exchanges that are logically related among them are identified as sharing the same *correlation data* [10], as it occurs, for example, when a

D. Lea and G. Zavattaro (Eds.): COORDINATION 2008, LNCS 5052, pp. 117–134, 2008.

unique id related to a client is passed in any message referring to that client. In both session-based and correlation based approaches, defining behavioural types has often proved difficult: while sessions make simpler, with respect to correlation approaches, to identify the interaction patterns that are to be typed, session based calculi with higher order session communication, defined in a π-calculus style, make typing non-trivial and not able to support automatic verification [4].

The aim of this paper is to investigate how we can maintain simple session typing, and therefore automatic verification, by defining an ad hoc session based service language which allows for an easy verification of the compatibility of interactions between services. The designed language, orcharts, expressing graphically data and control flows, allows for an easy traceabilty of sessions. This allows a finite-state behaviour type to be associated to a session, so that standard verification tools can be used to check compatibility between the client and the service. Indeed, our approach has been aimed at a language powerful enough to express common orchestration examples, but also simple enough to meet the typability requirement. The typing algorithm is briefly presented and the properties that can be verified over well typed services are discussed.

2 Informal Introduction to Typecharts and Orcharts

2.1 Sessions

A service oriented architecture is constituted by a collection of interacting services or *sites* (actually, in the following, we tend to use the word site to indicate a named entity that provides a service, and the word service when we refer to its behavioural aspects). Each site provides a service which may use services provided by other sites. Interactions between services occur by message exchange and in the context of shared *sessions*. Before invoking a service, the requestor creates a (unique) session name and attaches it to the name of the invoked service (example - the creation of a new session s bound to service ServiceFoo is written s@ServiceFoo). The session name is then used by the requestor in all subsequent interactions with the server pertaining to the same session (at a given point in time, a requestor may have many ongoing sessions with the same service). For instance, s.m() denotes the sending of message m() in the context of session s and hence s.m() is sent to ServiceFoo since s is bound to ServiceFoo. On the server side, at the reception of a first invocation message pertaining to a new session, a new session is started and a dialogue is initiated with the requestor in the context of this session. This dialogue takes place in both directions and on two new FIFO queues allocated for this purpose. In the present version of our approach we consider that different sessions that are being concurrently executed on the same server do not share information on that server. Sessions that are created on the server side (in order to provide services to requestors) are referred to as root sessions (root sessions are denoted by ρ).

2.2 Defining Services

The template for service definition is given in figure 1, where one can distinguish four main parts: Service name, Provides, Requires, and the defining Orchart. In figure 2 we present two definitions of services, namely, QuickNews and CollectNews, which revisit examples of News Services presented in [7]. Both services require the services of two News Agencies, CNN and BBC, and provide each a specific type of news service. The QuickNews service provides only one news item based on the first reply from the news agencies. The CollectNews service provides the news items collected from the two news agencies. The constructs used in QuickNews and CollectNews are commented in more detail section 2.4. Note that both required services, CNN and BBC, have the same required typechart, namely, NewsAgency-T. As can be seen in figure 3(a), NewsAgency-T is a typechart with a single request/response interaction scheme.

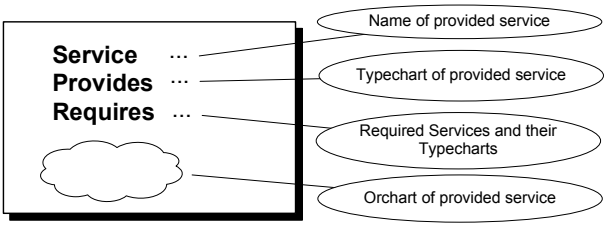

Fig. 1. Template of a site service definition

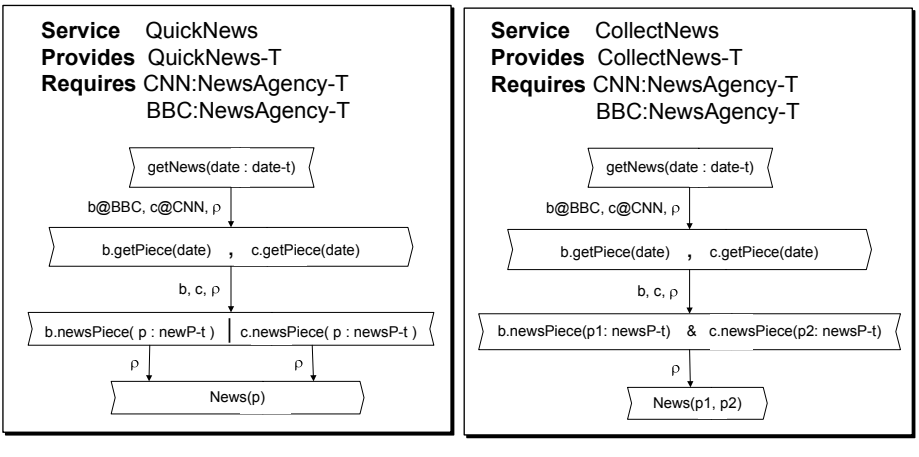

Fig. 2. The two versions of News service, with comments

2.3 Typecharts

Typecharts are a special kind of deterministic finite labelled transition systems where labels represent messages with parameter types. Parameter types can be data types (characterised by the -t suffix), or names of typecharts (characterised by the -T suffix). The transition system of a typechart has an initial state and one or more final states. States of a typechart are also partitioned in two subsets: sending states and receiving states (initial and final states can only be receiving states). Note that the typechart declared for a required service (e.g. CNN:NewsAgency-T) can be different from the one declared as provided in the service definition of this required service. For instance, 3(b) represents a possible provided typechart for the CNN and BBC services. This typechart allows for repeatable request/response interactions with the requestor. QuickNews and CollectNews do not exploit the possibility of reissuing a request in the same session but still can soundly interact with the CNN and BBC services. The relations between provided and required typecharts will be discussed in section 4.1. Note: we adopt a convention for typechart represntation which is to always adopt server's view. Hence, e.g, a sending state of a typechart has to be matched by a sending state in the server and a receiving state in the invoker.

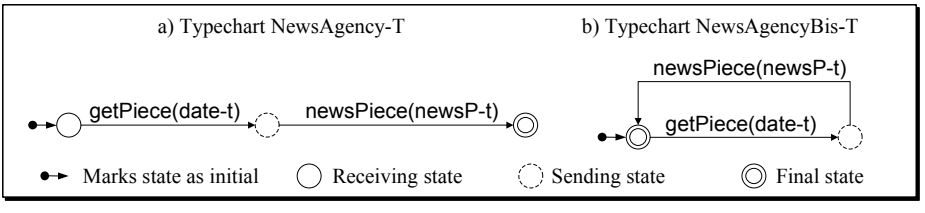

Fig. 3. Two typecharts of News Agency services

2.4 Orcharts

An Orchart is a finite directed acyclic graph where nodes can be of three types: input nodes, output nodes and instantiation nodes, and where edges can be of two types: data carrying edges and control edges.

Output Nodes. Figure 4 describes the input and output nodes. An output node may contain one or more message emissions. Messages may carry values that can be either simple data values or service names. Each message emission refers also to its emission context, i.e., a session name. Informally, one may think of output nodes as immediately executable: when the node receives control each of its messages is inserted in the FIFO queue corresponding to its named session.

Input Nodes. have an Internal Structure: They Are Subduivided in capsules (symbol | is used as a capsule separator). A capsule represents a possible branching from the output node. A capsule may contain one or more message receptions.

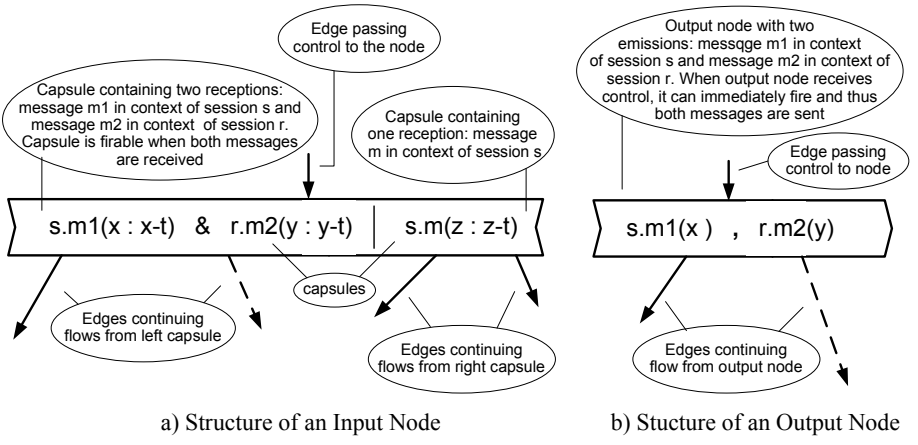

Fig. 4. Structure of Input and Output Nodes

In capsules with multiple receptions, all messages should pertain to different sessions. This constraint can be syntactically enforced. Capsules with multiple receptions are in fact a shorthand that can be rewritten in single message capsules. For lack of space, the details of this rewriting will not be addressed in this paper and in the sequel we conisder capsules to contain a single message. As an informal interpretation one may think of an input node to behave like a guarded command. When an input node receives control, its capsules can consume messages that are awaiting in the FIFO queues. When one message in a capsule is consumed this capsule is fired and the flow continues on all edges having their sources at this capsule. When a capsule is fired, all other capsules of the same input node (and their continuation flows) are discarded.

Data and Control Flow Edges. Nodes of an orchart can be joined with either control edges (represented by dotted arrows) or data flow edges (represented by solid line edges). Data flow edges in fact convey both control and data flow. Data flow edges are the means for binding variables with values: a use occurence of a variable can be bound with a binding occurence of this variable only if there is a directed path made of data flow edges starting at the binding occurence and ending at the use occurence. Moreover, variables are write-once, hence, in the semantics, we will use the replacement of variables by their values. Flow edges (control or data) can carry labels. These labels indicate the set of sessions that are continued on the flow and/or the set of sessions created on the flow. Examples of labelled flow edges are provided in the following sections.

2.5 Revisiting the QuickNews and CollectNews Examples

In figure 5 we describe the different constructs of the orcharts used in service definitions of QuickNews and CollectNews introduced in section 2.2. For better

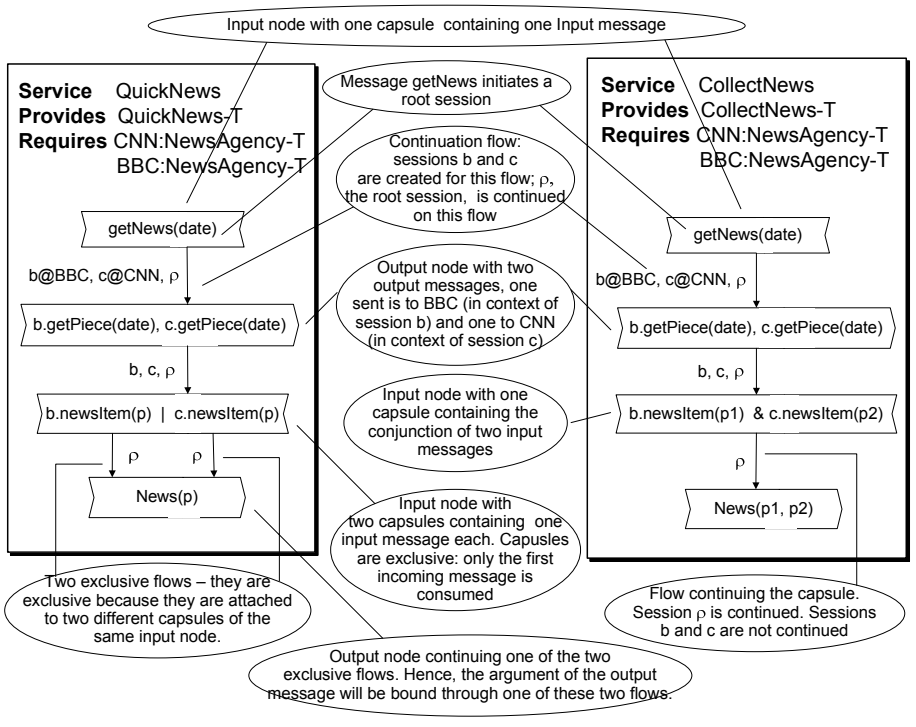

Fig. 5. The two versions of News service with comments

readability, we chose not to represent the types of the used data. The comments in the picture follow the flow of the behaviour of the orcharts, explaining the meaning of the various types of nodes and of the flow of session names and variables along the edges. Before being used for an interaction, session names must be bound to a service name. Variables are given a value in an input message (occuring in a capsule), which value is then used in ouput nodes. See for instance how variable date acquires a value in the getNews message which is used downstream in the getPiece output message.

2.6 Definition and Instantiation of Named Orcharts

In order to provide for recursion, orcharts use the classical approach of naming and instantiating behaviours. In figure 6 we give the definition of a GetBestPrice service which illustrates the use of a named orchart instantiation. Briefly, the GetBestPrice service returns, for a given product item requested by the user, the name of the shop that sells this item at the best price. The GetBestPrice service requires the services of ShopsFinder which provides all shops selling a given item; and of MinEval which provides the minimum of a set of values. The behaviour of this service is as follows. A (root) service session is started with the input of getBestPrice

message, then a session named sf (with ShopsFinder) is created then used to invoke ShopsFinder. Then a session m bound to MinEval is created and the HARVEST orchart is instantiated with session parameters sf, m and ρ and value item. HARVEST collects shops proposals coming from ShopsFinder and, for each response, invokes the shop to get the price of the item and then sends the price to MinEval. When HARVEST terminates, i.e., when its orchart reaches the exit node (exit nodes are described in the sequel), a flow is continued in which MinEval is invoked to get the shop having the best price. Finally, this information is returned to the user. It is worth noting in this example how orcharts are named and instantiated. Named Orcharts are defined within dotted rounded boxes and instantiated using solid line rounded boxes. The name of the orchart is placed inside the box and is followed by the session parameters (in square brackets) and value parameters (in parenthesis). The definition of a named orchart starts with an initial input node and may have exit nodes (zero or more) represented with small circles placed at the boundary of the definition box (on the dotted line). At exit nodes, the sessions that are continued are given in square brackets whereas the values that are returned are given in parenthesis (the HARVEST example only shows continued sessions). A syntactical constraint is enforced that the sessions continued at an exit node must be a subset of the session parameters. For instance, HARVEST has session parameters [sf, m, ρ] but only [m, ρ] are continued from the exit node. In order to simplify the presentation, but without a loss of generality, we consider in the present paper that named orcharts may have at most one exit node. To this exit node correspond an exit point in the instantiation diagram of the named orchart (exit points are also represented with a small circle). The dynamic semantics of instantiation is defined through unfolding. When control reaches an instantiation, the instantiation node is replaced by the definition of the named orchart. In this replacement,

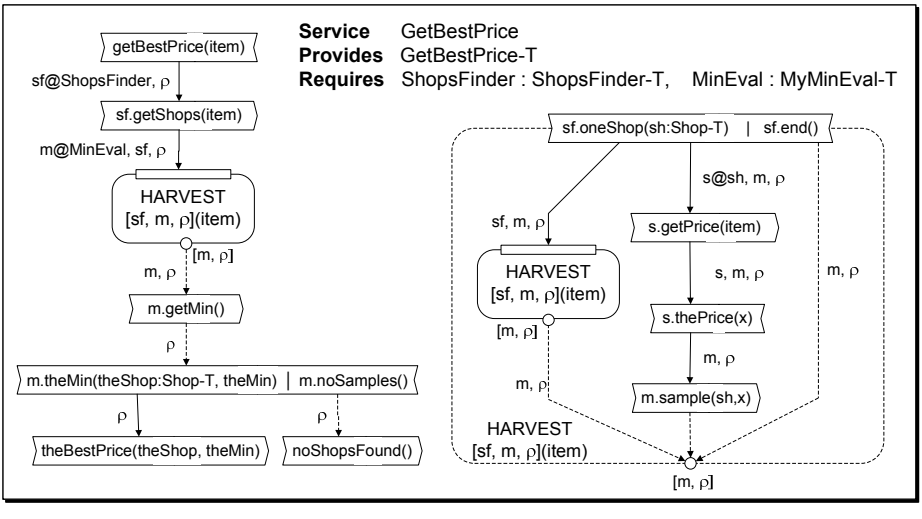

Fig. 6. The Get Best Price Service Definition

the edges in the definition orchart whose targets are at the exit node have their target redirected to the successor node of the exit point in the instantiation.

2.7 Parallel Flows

The orchart defining HARVEST involves the use of parallel flows. For instance, when message oneShop carrying a shop name sh is consumed, behaviour continues in two flows, the left flow is (re−)instantiation of HARVEST and the right flow proceeds with invoking the shop sh and storing the obtained price in MinEval. Note how sessions m and ρ are present in these two parallel flows whereas session sf is only present in the left flow. Note the use, in the definition of this named orchart, of flow control edges. The parallel flows feature allows for the creation of an unbounded number of sessions. The fact that any definition of a named orchart starts with an input node enforces that the creation of sessions and flows is always guarded.

3 Formal Syntax and Semantics

In this section we give a formal definition of the Orchart language. To remain within page limits, we simplify the syntax w.r.t. the one used in the examples, e.g. by considering only one message emitted by an output node, and no conjunction of messages inside a capsule. Indeed, both these constructs can be defined as shorthands for orcharts employing the basic constructs considered in this section.

3.1 Syntax of Orchestration Charts (Orcharts)

The syntax of the language assumes the following:
$r, r', s, s', \ldots$ range over session names, $\rho, \sigma, \sigma' \ldots$ over session values
$w, w' \ldots$ over service names, $a, a' \ldots$ over orchart names
$m, m' \ldots$ over message names, $G, G' \ldots$ over orcharts
$n, n' \ldots$ over nodes, $v, v' \ldots$ over values of any type
$x, x', y, y', z, z' \ldots$ over (any type of) variables
An orchart can be defined as a sestuple $(N, C, E, \mathcal{L}_N, \mathcal{L}_E, Exp)$ where:

- N is a set of nodes: $N = IN \uplus ON \uplus InstN$, where IN is the set of input nodes, ON the set of output nodes, $InstN$ the set of instantiation nodes;
- IN is defined as a partition of a set of capsules C: $\forall n \in IN, n \subseteq C \; and \; \forall n, n' \in IN, n \neq n' \implies n \cap n' = \emptyset$.
- E is a set of edges connecting nodes: $E \subseteq (C \times N \cup (ON \cup InstN) \times N)$, that is, edges starting an input node are actually associated to a capsule. Moreover, E is partitioned in DE, the set of data carrying edges, and CE, the set of control carrying edges: $E = DE \uplus CE$;
- $\mathcal{L}_N$ is a labelling function that associates to each node a set of expressions, whose number and syntax is depending on the type of node: $\mathcal{L}_N : N \to 2^{Exp}$; in particular, this function is defined on input nodes by means of a function $\mathcal{L}_C$ that labels capsules: $\mathcal{L}_C : C \to Exp$;

- $\mathcal{L}_E$ is a labelling function that associates to each edge two sets of expressions, namely the set of passed data (only for edges in DE) and the set of passed sessions: $\mathcal{L}_E : DE \rightarrow 2^{Exp} \times 2^{Exp} \uplus CE \rightarrow 2^{Exp}$;
- Exp is a domain of expression that are used to label nodes and edges: the syntax of the expressions and their association to the various kinds of nodes and edges is reported below.

$Exp ::= CExp \mid OExp \mid SExp \mid DExp \mid SSExp$
$Xlist ::= x \mid Xlist, Xlist$
$SExp ::= \sigma \mid s \mid SExp, SExp$
$SSExp ::= \sigma \mid s \mid s@w \mid s@x \mid SSExp, SSExp$
$CExp ::= s.m() \mid s.m(Xlist)$
$OExp ::= s.m() \mid s.m(DExp)$
$DExp ::= x \mid w \mid DExp, DExp$

In this section, for brevity, we ignore the syntax of types of expressions. Indeed, as shown in session 2, it is sufficient to consider typing expressions with the standard notation $x : T$. Expressions are used to label edges and nodes:

- each capsule in an Input node is labelled with an expression in $CExp$;
- each Ouput node is labelled with an expression in $OExp$;
- each Control edge is labelled with an expression in $SSExp$;
- each Data edge is labelled with an expression in $SSExp$;
- each Instantiation node is labelled with an orchart name, followed by an expression in $SExp$ and an expression in $DExp$ (respectively, actual session and data parameters);

The following use the additional notations:
$e, e', \ldots$ range over edges $\quad c, c' \ldots$ range over capsules

Given an orchart G $= (N, C, E, \mathcal{L}_N, \mathcal{L}_E, Exp)$, with $N = IN \uplus ON \uplus InstN$, we also define:
$Init(G) = \{n \in N \mid \nexists e \in E, \nexists n' \in N : e = (n', n)\}$
Given $n \in N$: $\quad OutE(n) = \{e \in E \mid \exists n' \in N : e = (n, n')\}$;
given $c \in C$: $\quad OutE(c) = \{e \in E \mid \exists n' \in N : e = (c, n')\}$
In particular, if $n \in InstN, \mid OutE(n) \mid \leq 1$: this means that only one edge can go from the exit point of an instantiation node.

A Named orchart DG is a quadruple: $(a, FParms, FRParms, G)$ where a is a name, $FParms \subseteq 2^{SExp} \times 2^{DExp}, FRParms \subseteq 2^{SExp} \times 2^{DExp}$ are respectively a set of formal parameters and a set of formal return parameters, G is an orchart having a single initial node n ($Init(G) = \{n\}$), and a set of Nodes augmented with an Exit Node r. That is, for such $G : N = IN \uplus ON \uplus InstN \uplus \{r\}$, $\mathcal{L}_N(r) = FRParms$; $OutE(r) = \emptyset$.

3.2 Static Constraints

Session names that label an edge departing from a node should be a subset of the union of the session names that label its incoming edges.

A node can refer to a session (for input/ouput messages) only if it is in the union of the session names that label its incoming edges.

As usual, actual parameters of an instantiation node should correspond in number, position and type to the formal parameters of the called definition. To be more precise, session actual parameters should correspond to session formal parameters, while data actual parameters should have the same type of the corresponding actual parameter variables. The same should hold for the return parameters. Session names used as actual parameters in an instantiation node should be a subset of the union of the session names that label its incoming edges. The union of the session names labelling the outcoming edges of an instantiation node should be a subset of the session names used as actual return parameters. The session names used as actual return parameters should be a subset of the session names used as actual parameters (this enforces that sessions created inside a named orchart are forgotten before reaching the exit node).

Any variable declaration in a capsule binds all use occurences of the same variable that can be reached from the capsule using a path made of contiguous data flow edges and not containing other binding occurence of the same variable (i.e., the closest occurence is the binding occurence). In well defined orcharts all use occurences must be bound. Furthermore, for a given use occurence of a variable there may be more than one binding occurence. A static rule (not detailed here) enforces that only one path from a binding to a bound occurence can be executed, i.e., if a path leading to the use occurence is executed then all the others have been discarded, and there is always one such path (there is no execution that discards all the paths linking all binding occurences with a use occurence). The same rules hold for binding sessions, with the particularity that binding occurences are session creations (all other occurences are use occurences) and the binding paths are made of any type of edges (control or data).

3.3 Informal Semantics of Orcharts and of Configurations of Services

We recall that a service is constituted by:

- a service name, w,
- a provided typechart,
- a set of required service names with their typecharts,
- an orchart, G_w, with a single initial node, which is an input node. This orchart can contain instantiation nodes that refer to named orcharts
- a set of definitions of named orcharts which are referred by the "main" orchart and which can refer each other, also in a recursive fashion.

Note: In the sequel, we consider that in definition orcharts, session name ρ is explicitely added as a prefix to the appropriate input and output messages, i.e, those with no session name prefix. Hence, occurence $m(v)$ becomes $\rho.m(v)$.

The dynamic semantics of orcharts is defined based on graph transformations along with the execution of input and output interactions. Depending on the type of the executed interaction, an orchart undergoes a series of transformation steps. These are explicited hereafter.

Message Output. This is the case where the orchart has one output node in its set of initial nodes:

- the message contained in the output node is deposited in the appropriate queue,
- the output node is removed, but its set of departing edges are kept (the edges remain pending inward, i.e, with their sources unattached),
- for each session creation label $(s@w)$ occurring on a pending edge:
  - a unique session id, σ, is generated,
  - an instance of the requested service (σ, G_w) is spawned and inserted at the server site hence $w(... \mid ...)$ becomes $w(... \mid (\sigma, G_w) \mid ...)$,
  - two empty FIFO queues (one for each direction) are added thus linking the present orchart and the spawned service instance,
  - all use occurences of s in the orchart that are bound to $(s@w)$ are substituted with the created session id σ.
- when all the session creation labels of pending edges have been treated, all pending edges are removed,
- all instantiations that appear as initial nodes in the resulting orchart are replaced by their corresponding definition,
- the orchart is ready for considering another execution step.

Message Input. This is the case where the orchart has no output nodes and at least one input node in its set of initial nodes (all output nodes must be executed before considering the execution of input nodes). If the input node has one of its capsules containing a reception that matches the frontmost message of the corresponding queue:

- the message is removed from the queue,
- the variables declared in the reception are replaced with the corresponding values in the received message,
- the substitution of the variables by their values is carried over all the bound occurences in the orchart,
- the edges originating in the capsules other than the one that received the message are discarded,
- the parts of the graph that are no more reachable from the initial nodes are removed,
- the input node is removed, but its set of departing edges are kept (the edges remain pending inward, i.e, with their sources unattached),
- session creations that label pending edges are treated in a way similar to the message output case,
- when all the session creation labels of pending edges have been treated, all pending edges are removed,
- all instantiations that appear as initial nodes in the resulting orchart are replaced by their corresponding definitions,
- the orchart is ready for considering another execution step.

In the sequel, we proceed with the formalisation of the above steps. We need to revisit the syntax of orcharts in order to include those elements that apppear during execution steps. Thus an *execution orchart* is an extension of orcharts that includes the possibility for nodes to have inward pending edges. The set of such edges for a node n is named $InE(n)$.

3.4 Structure of Running Configurations of Services

We first define the structure of a running configuration of services then we provide the rules that govern its global behaviour based on the behaviour of its service instances. A running configuration involves a set Σ of active session ids ranged over by σ. Σ is endowed with two functions Req and $Serv$. $Req(\sigma)$ allows to retrieve the session id of the service instance that created σ and $Serv(\sigma)$ yields the name of the service that is responding to the request issued in the context of σ. Hence, if (σ', G) executes session creation $s@w$ with session id being σ assigned to s, we will have $Req(\sigma) = \sigma'$ and $Serv(\sigma) = w$. Furthermore, the execution of $s@w$ creates also service instance (σ, G_w) which is dedicated to the execution of service requests from (σ', G) in the context of σ. A running configuration is given by:

$$Conf = Q_{RS} \mid Q_{SR} \mid w_1(S_1) \mid \cdots \mid w_n(S_n) \qquad where:$$

- S_i is a (possibly empty) set of instances of service w_i. An element of S_i is a pair (σ, G) where G is the current execution orchart of the service instance that is serving session σ,
- Q_{RS} and Q_{RS} are a pair of functions on session ids. $Q_{RS}(\sigma)$ is the Queue from $Req(\sigma)$ to its provider and $Q_{SR}(\sigma)$ is the dual queue.

The operational semantics of a running configuration of services is given by reduction rules (section 3.5) that define possible execution steps. Configurations can evolve either by an output move by a service instance which puts a message in the proper queue (rules OUT-S and OUT-R define such a move for the two cases, server to requestor and requestor to server, respectively); or an input move of a service instance which removes a message from a queue (rules IN-S and IN-R define such a move for the two cases, requestor to server and server to requestor, respectively); or a creation of a new session by a requestor instance, which adds a new (server) service instance to the configuration, and adds a pair of empty queues to the set of queues, both bound to the requestor and server service instances (this move is mirrored in rule CREATE). As can be seen from the rules, there are many sources of non determinism in the execution of an orchart that the user should be aware of: (i) in case of two parallel flows starting each with the reception of the same message, (ii) in case of the same message present in two different capsules of the same input node, (iii) in case where two FIFO queues have their head messages ready to be received in different capsules of the same node. Initially, a configuration which is made of an empty set of queues and of no service instances cannot proceed. In fact, we need to designate a "main" client (not considered here for lack of space) in order to trigger the behaviour and to animate the configuration.

3.5 Operational Semantics Rules

The operational semantics of a running configuration of services is given by reduction rules that define possible execution steps:

$$\text{OUT-S}\ \frac{nocreate(S_1,\ldots,S_n),\ (\sigma,G)\in S_i,\ n\in OutN(G)\cap Init(G)\quad \mathcal{L}_N(n)=\rho.m(v),\ G'=rmnode(n,G)}{Q_{RS}\mid Q_{SR}\mid w_1(S_1)\mid\ldots w_i(S_i)\mid\ldots w_n(S_n)\rightarrow Q_{RS}\mid Q'_{SR}\mid w_1(S_1)\mid\ldots w_i(S'_i)\mid\ldots w_n(S_n)}$$

where: $S'_i=S_i\backslash(\sigma,G)\uplus(\sigma,G'),\ Q'_{SR}=Q_{SR}\backslash(\sigma,Q_{SR}(\sigma))\uplus(\sigma,add(m(v),Q_{SR}(\sigma))$

$$\text{OUT-R}\ \frac{nocreate(S_1,\ldots,S_n),\ (\sigma',G)\in S_i,\ n\in OutN(G)\cap Init(G)\quad \mathcal{L}_N(n)=\sigma.m(v),\ G'=rmnode(n,G)}{Q_{RS}\mid Q_{SR}\mid w_1(S_1)\mid\ldots w_i(S_i)\mid\ldots w_n(S_n)\rightarrow Q'_{RS}\mid Q_{SR}\mid w_1(S_1)\mid\ldots w_i(S'_i)\mid\ldots w_n(S_n)}$$

where: $S'_i=S_i\backslash(\sigma',G)\uplus(\sigma',G'),\ Q'_{RS}=Q_{RS}\backslash(\sigma,Q_{RS}(\sigma))\uplus(\sigma,add(m(v),Q_{RS}(\sigma))$

$$\text{IN-S}\ \frac{onlyin(S_1,\ldots,S_n),\ (\sigma,G)\in S_i,\ n\in IN\cap Init(G),\ c\in n\quad \mathcal{L}_C(c)=\rho.m(x),\ m(v)=head(Q_{RS}(\sigma)),\ G'=rmcaps(n,c,G)[x/v]}{Q_{RS}\mid Q_{SR}\mid w_1(S_1)\mid\ldots w_i(S_i)\mid\ldots w_n(S_n)\rightarrow Q'_{RS}\mid Q_{SR}\mid w_1(S_1)\mid\ldots w_i(S'_i)\mid\ldots w_n(S_n)}$$

where: $S'_i=S_i\setminus(\sigma,G)\uplus(\sigma,G'),\ Q'_{RS}=Q_{RS}\setminus(\sigma,Q_{RS}(\sigma))\uplus(\sigma,tail(Q_{RS}(\sigma))$

$$\text{IN-R}\ \frac{onlyin(S_1,\ldots,S_n),\ (\sigma',G)\in S_i,\ n\in IN\cap Init(G),\ c\in n\quad \mathcal{L}_C(c)=\sigma.m(x),\ m(v)=head(Q_{SR}(\sigma)),\ G'=rmcaps(n,c,G)[x/v]}{Q_{RS}\mid Q_{SR}\mid w_1(S_1)\mid\ldots w_i(S_i)\mid\ldots w_n(S_n)\rightarrow Q_{RS}\mid Q'_{SR}\mid w_1(S_1)\mid\ldots w_i(S'_i)\mid\ldots w_n(S_n)}$$

where: $S'_i=S_i\setminus(\sigma',G)\uplus(\sigma',G'),\ Q'_{SR}=Q_{SR}\setminus(\sigma,Q_{SR}(\sigma))\uplus(\sigma,tail(Q_{SR}(\sigma))$

$$\text{CREATE}\ \frac{(\sigma,G)\in S_i,\ n\in Init(G),\ s@w_j\in InE(n)\quad G'=rmlabel(s@w_j,n,G)[s/\sigma'],\sigma'\text{ fresh}}{Q_{RS}\mid Q_{SR}\mid w_1(S_1)\mid\cdots\mid w_i(S_i)\mid\cdots\mid w_j(S_j)\mid\cdots\mid w_n(S_n)\rightarrow Q'_{RS}\mid Q'_{SR}\mid w_1(S_1)\mid\cdots\mid w_i(S'_i)\cdots\mid w_j(S'_j)\mid\cdots\mid w_n:S_n}$$

where: $S'_j=S_j\uplus(\sigma',G_{w_j}),\ S'_i=S_i\backslash(\sigma,G)\uplus(\sigma,G'),Q'_{SR}=Q_{SR}\uplus(\sigma',\emptyset),Q'_{RS}=Q_{RS}\uplus(\sigma',\emptyset)$

The above rules are based on the use of some auxiliary functions, that allow to work on the queues associated to sessions and on the execution graph itself, or to give a priority to the application of the above rules. We present them here informally for sake of brevity:

- $add(m,queue), tail(queue), head(queue)$ - usual functions over a FIFO queue;
- $rmnode(n,G)$ - removes the node n from G, with the following steps:
  - cancel n from G, but retaining its outcoming edges from it
  - if any retained edge hits an instantiation node, substitute it with its definition
  - if any retained edge is not labelled with a session creation ($s@w$), it is cancelled
- $rmcaps(n,c,G)$ - (here c is a capsule of n) removes the node n from G, with the following steps:
  - Given that $OutE(n)=OutE(c)\cup Excluded$, if $Excluded\neq\emptyset$: cancel from G all the edges in $Excluded$, then cancel all the nodes which are

no more reachable from nodes in $Init(G)$, together with their outcoming edges.
  - apply $rmnode(n, G)$
- $rmlabel(s@w, n, G)$ - removes label $s@w$ from the edge pointing at n in G, then proceeds with removing all edges in G having no session creation labels.
- $nocreate(S_1, \ldots, S_n)$ is a predicate defined as: $\forall i, \sigma, G : (\sigma, G) \in S_i, n \in Init(G) : InE(n) = \emptyset$
- $onlyin(S_1, \ldots, S_n)$ is a predicate defined as: $nocreate(S_1, \ldots, S_n)$ and $\forall i, \sigma, G : (\sigma, G) \in S_i, n \in Init(G) : n \in IN$.

4 Type Verification and Properties

4.1 Behavioural Types

A typechart is a quintuple $(S, s_0, S_F, Act, \rightarrow)$ where:

- S is a finite set of states, defined as $RS \uplus SS$, that is, a state is either a *receiving state* or a *sending state*.
- $s_0 \in RS$ is the initial state
- $S_F \subseteq RS$ is the set of $final$ states
- Act is a set of actions, which are in the form $?m(Type)$ (input message) or $!m(Type)$ (output message), where m is a message name and *Type* is either a basic type or a reference to another typechart. Since in general messages can carry more data values, we assume for simplicity that structured types are included in basic types to cover such cases.
- $\rightarrow$: $S \times Act \times S$ is the labelled transition relation, such that: $s \xrightarrow{?m(T)} s' \implies s \in RS$, $s \xrightarrow{!m(T)} s' \implies s \in SS$.

A session has two ends: the end of the client and the end of the service. Session types differ for a session if seen from the two ends, in the fact that what is an input on one side is an output on the other side. This is called *type duality* in [15]. The type T as seen from the other end of the session is written $Dual(T)$. In particular, subtyping of [15] can be expressed in a way resembling the classical simulation relation typical of a process algebraic framework, by distinguishing sending and receiving states (we abstract here from the exchanged messages, to which a classical notion of subtyping could be applied as well):

$$T_1 \textit{ is a subtype of } T_2 \ (T_1 \preceq T_2) \textit{ iff } \begin{cases} T_2 \xrightarrow{?m} T_2' \textit{ implies } \exists T_1' : T_1 \xrightarrow{?m} T_1' \textit{ and } T_1' \preceq T_2' \\ T_1 \xrightarrow{!m} T_1' \textit{ implies } \exists T_2' : T_2 \xrightarrow{!m} T_2' \textit{ and } T_1' \preceq T_2' \end{cases}$$

which is read: T_1 is a subtype of T_2 if in any receiving state, T_1 is able to receive all the messages that T_2 is able to receive, and in any sending state T_2 is able to send all the messages that T_1 is able to send. Consequently, substitutability and compatibility are defined, as in [15]:

- a session type T can *safely substitute* T' if $T \preceq T'$;
- a session type T is *compatible* with T' if $T \preceq Dual(T')$.

That is, two type sessions are said compatible if any sending of one is matched by a reception of the other one: hence, a session having at its two ends compatible types does not internally deadlock. The typecharts NewsAgency-T and NewsAgencyBis-T shown in section 2.3 are defined so that NewsAgencyBis-T *can safely substitute* NewsAgency-T.

4.2 A Well Typedness Algorithm

Herefater we present a well typedness algorithm, i.e., which verifies that an orchart defining a site conforms to its provided and required typecharts. For the sake of brevity, we limit its description to a brief sketch, sufficient in our opinion to show that well-typedness of orcharts can be computed.

First, the provided and required typecharts need to be transformed. The provided typechart is transformed into its dual. Then its sending transitions are prefixed with τ, i.e., every transition $T \xrightarrow{!m} T'$ becomes $T \xrightarrow{\tau} \bullet \xrightarrow{!m} T'$ where $\bullet$ is a new state with only one sending transition $\xrightarrow{!m}$. On the other hand, the typecharts of the required services only undergo the τ prefixing transformation. The introduction of τ transitions is meant to mimic the fact that the decision of sending a message is taken autonomously by the sender.

The algorithm proceeds by discharging proof obligations. Discharging a proof obligation either fails, in which case the whole algorithm immediately terminates concluding a typing error, or produces a set of new proof obligations to be discharged. When no more proof obligations are left to be discharged, the algorithm terminates, establishing conformance. A trivial proof obligation, (e.g. the one in which an empty orchart is compared against a terminal state of a typechart) is immediately discharged producing no new proof obligations.

The initial proof obligation is $(G_{serv},\ s_0 : T_{prov})$ where G_{serv} is the orchart of the service and T_{prov} the provided typechart (in this algorithm, we chose to rename ρ by s_0, which simplifies the presentation). From this initial proof obligation we proceed with symbolic co-execution steps, where the orchart and the associated typecharts are executed in a synchronised fashion. The format of a running proof obligation is given by $(G,\ s_0 : T_0,\ s_1 : T_1, \ldots, s_n : T_n)$ where G is the current state of the orchart, T_0 its current provided typechart and $s_1 : T_1, \ldots, s_n : T_n$ the set of active sessions and their associated typecharts. To discharge a proof obligation $(G,\ s_0 : T_0,\ s_1 : T_1, \ldots, s_n : T_n)$, which we assume for the moment having no instantiation nodes, we perform the following steps:

- If some typechart has a τ transition:
  - For each typechart T_i with a τ transition $T_i \xrightarrow{\tau} T'$: create a new proof obligation obtained by replacing T_i with T';
  - Discharge proof obligation $(G,\ s_0 : T_0,\ s_1 : T_1, \ldots, s_n : T_n)$;
- If no typechart has a τ transition and G's initial nodes are only input:
  - If G has no initial input capsule $s_i.m$ that matches a transition $T_i \xrightarrow{!m} T'$ of its corresponding typechart T_i then the proof fails;
  - For each initial capsule $s_i.m$ matching one transition $T_i \xrightarrow{!m} T'$ of its associated typechart T_i: a new proof obligation is produced, applying the

execution step involving the capsule $s_i.m$, so obtaining an execution orchart G', and advancing to T' the typechart of s_i. In this step the set of active sessions is obtained by collecting the labels of the edges outcoming from the capsule. This may involve the creation of new active sessions produced from the labels having the $s@serv$ format;
  - When all initial capsules with matching typecharts are treated the current proof obligation is discharged;
- If no typechart has a τ transition and G has initial output nodes:
  - for each output node emitting $s_i.m()$, if $T_i \xrightarrow{?s_i.m} T'$, a new proof obligation is produced applying the execution step on that node, so obtaining an execution Orchart G', and advancing T_i to T' ;
  - if for some output node there is no matching typechart, the proof fails;
- If no typechart has a τ transition and G is empty: if T_0 is a terminal state then the proof is discharged, otherwise the proof fails

Since an orchart is acyclic, the algorithm is guaranteed to terminate, since its number of steps depends on static metrics (number of nodes and capsules, of sessions, of alternative sendings in typecharts).

On top of this basic algorithm, orchart instantiation is addressed as follows. If an instantiation of a named orchart is encountered for the first time, it is replaced by its definition and the algorithm continues with the creation of a proof obligation for the definition of this named orchart (parametrized with the states of the active sessions). The created proof obligation is discharged when the algorithm has explored, in the current orchart, the part that comes from the definition orchart. Another condition has also to be checked which ensures that in case of parallel flows, if a session is present in the instantiated part and also in another parallel flow, the behaviour of its associated typechart is uniform (i.e., roughly, the state of the typechart does not change) along all parallel flows where the session is present.

4.3 Properties of Well Typed Configurations of Services

A configuration of services is well typed iff (i) each service is well typed (its defining orchart conforms to its required and provided types as given in the algorithm of section 4.2), and (ii) if a service in this configution requires a type T_1 and the partner service provides a type T_2, then ($T_2 \preceq T_1$). If we assume that defining orcharts have a stubborn terminal output node (a node with no outgoing edges and which is always reachable - this can be statically checked), if we assume also that there is no invocation cycles (a typical cycle is when service w_1 invokes w_2 and vice versa - this also can be statically checked) then we claim that well typed configurations have the soundness property: any service invocation potentially reaches a termination state. More precisely, let us consider a sound configuration $Conf = (\ w_0(G_0), w_1(), \ldots, w_n()\ )$ where w_0 is a client (with behaviour given by orchart G_0) ready to invoke service w_1 with some session σ, then for any run $Conf \xrightarrow{*} Conf'$, there exists a configuration, $Conf''$, reachable from $Conf'$ and such that $Conf'' = (\ w_0(G'_0), w_1(\sigma, G'), \ldots, w_n(\ldots)\ )$ and where G'_0 and G' are empty.

5 Future Work and Conclusions

We have presented an approach for verifying service composition based on behavioural typing, in which sessions play a pivotal role. In this work we sought for a language powerful enough to express common service orchestration examples, but which is also simple enough to associate finite state behavioural types to sessions. The first results about typing are encouraging: we can cite the fact that the language, admitting parallel flows and recursion, allows infinite state behaviours to be defined while also being typable, that is, to which finite state session types can be associated. It is worth noting that the properties that are claimed for in well typed orchart configurations are similar to those obtained for object configurations in [12] with, however, two major improvements: (i) orcharts are more expressive as they provide for parallel flows; and (ii) orcharts are less constraining as they do not impose that services are always ready for all input messages that are expected for by their current behavioural types. The precise tradeoff between expressive power of orcharts and their finite typability has still to be assessed. Moreover, several improvements to the language are planned, for example in the treatment of abandoned sessions, with the introduction of explicit and implicit abort of sessions.

Acknowledgments. This work has been partially supported by the ACI project FIACRE and by the EU project FET-GC II IST-2005-16004 Sensoria.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed Parallel Databases 14(1), 5–51 (2003)
2. de Alfaro, L., Da Silva, L.D., Faella, M., Legay, A., Roy, P., Sorea, M.: Sociable Interfaces. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 81–105. Springer, Heidelberg (2005)
3. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V., Zavattaro, G.: SCC: A Service Centered Calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
4. Caires, L., Ferrari, G., Pugliese, R., Ravara, A.: Behavioural Types for Service Composition. Deliverable D2.3.a, Sensoria project (September 2006)
5. Cook, W.R., Patwardhan, S., Misra, J.: Workflow Patterns in Orc. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 82–96. Springer, Heidelberg (2006)
6. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
7. Kitchin, D., Cook, W.R., Misra, J.: A Language for Task Orchestration and Its Semantic Properties. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 477–491. Springer, Heidelberg (2006)
8. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the Pi-Calculus. ACM TOPLAS 21(5), 914–947 (1999)

9. Lanese, I., Vasconcelos, V.T., Martins, F., Ravara, A.: Disciplining Orchestration and Conversation in Service-Oriented Computing (2007)
10. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
11. Larsen, K.G., Nyman, U., Wasowski, A.: An Interface Theory for Input/Output Automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 82–97. Springer, Heidelberg (2006)
12. Najm, E., Nimour, A., Stefani, J.B.: Guaranteeing liveness in an object calculus through behavioral typing. In: Proceedings of FORTE/PSTV 1999, Beijing, China, October 1999, Kluwer, Dordrecht (1999)
13. Najm, E., Nimour, A.: Explicit Behavioral Typing for Object Interface. In: Semantics of Objects as Processes, ECOOP 1999 Workshop, Lisbon, Portugal (June 1999)
14. Nierstrasz, O.: Regular types for active objects. In: Nierstrasz, O., Tsichritzis, D. (eds.) Object-Oriented Software Composition, pp. 99–121. Prentice-Hall, Englewood Cliffs (1995)
15. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the Behavior of Objects and Components using Session Types. Fundamenta Informaticae 73(4) (2006)