Behavioural types for service integration: achievements and challenges

Antonio Brogi[1], Carlos Canal[2], and Ernesto Pimentel[2]

[1] Department of Computer Science, University of Pisa, Italy
[2] Department of Computer Science, University of Málaga, Spain

Abstract. Increasing attention is being paid to the development of so-called behavioural types as a means to formally address the problem of ensuring the correct interoperation of software components as well as of Web services. The objective of this paper is to provide an insightful synthesis of the state-of-the-art in this area, both to summarise the main results achieved and to point out some important challenges to be faced for a real impact of these ideas in the software world.

1 Introduction

Service integration is widely recognized to be one of the crucial problems in Component-Based Software Engineering [8]. Available component-oriented platforms address software interoperability at the signature level via Interface Description Languages (IDLs) that are designed for specifying the functionality offered by heterogeneous components. Unfortunately, while IDLs permit to overcome signature mismatches, they do not guarantee that the components will interoperate correctly, as undesired deadlocks may occur because of mismatches in the interaction behaviour of the components involved [12].

As advocated for instance in [14], in order to overcome behavioural mismatches component interfaces need to include some protocol information. A substantial amount of research has been hence devoted to advocate the application of formal methods to describe the interactive behaviour of software systems in order to support system analysis and verification in general, and behavioural mismatching detection in particular. Different formalisms have been proposed for extending IDLs with behavioural information, in particular finite state machines (e.g., [14]) and process calculi (e.g., [1]).

While finite state machines support a simple and efficient verification of protocol compatibility, their simplicity constitutes a severe expressiveness bound for modelling complex open systems. Dually, process calculi feature more expressive descriptions of protocols and enable more sophisticated analysis, but the inherent complexity of verification procedures inhibits their usability in practice. A suitable trade-off between expressiveness and amenability to efficient verification is therefore needed.

The problem of service integration is of primary importance also in the emerging field of Web Services [10]. The functionality offered (and required) by Web

Services may be currently expressed by means of the Web Service Description Language (WSDL), by declaring a set of message formats and their direction (incoming/outgoing). WSDL declarations can be exploited to verify the possibility of connecting different services, yet they do not ensure the correct interoperation of the services (e.g., safety or liveness properties) – very much like IDLs in the case of heterogeneous component integration. To overcome this limitation, different proposals, such as BPEL4WS or WSCI, have been put forward to describe the so-called Web service orchestration and choreography. These XML-based languages can be used to extend WSDL service interfaces with a description of the real interaction protocol followed by the service to exchange messages with other services. A serious criticism to available Web service choreography languages is, however, that they declare "too much" information (viz., the full protocol), hence inhibiting the possibility of implementing efficient verifications [7].

In this perspective, [7] puts forward the development of behavioural types, rigorous typing disciplines aiming at synthesizing the essential aspects of the interaction behaviour of services, while retaining efficient automatic verification of crucial properties of composed systems, such as lock-freedom. Differently from contract languages, such as BPEL4WS and WSCI that are computationally complete, the expressive power of these type systems is below Turing-completeness. The development of behavioural types has been receiving increasing attention, as witnessed by the recently proposed session types [3], action types [16], usage types [4], and process types [5].

The objective of this paper is to provide an insightful synthesis of the current state-of-the-art in the development of behavioural types for service integration. In order to help the reader get an intuitive understanding of behavioural types, we first introduce a simple motivating example of service integration (Sect. 2) and illustrate how it can be modelled by means of finite state machines, process algebras, and behavioural types, respectively. We then analyse different proposals of behavioural types that have been put forward in the literature (Sect. 3). We then single out (Sect. 4) a list of relevant properties that we use to analyse and assess the above considered proposals. Finally, we will draw some conclusions (Sect. 5) and try to highlight some important challenges to be faced before behavioural types can have an effective impact in the software world.

2 Motivating example

In this Section we will show, by means of an example, several approaches for modelling system behaviour. These approaches differ both in the formalism used, and in the level of abstraction of the specification.

Suppose a Client/Server system in which the Client opens a connection with the Server issuing a login command, and then repeatedly requests the Server for performing a query —getting the corresponding results from private reply channels. The Client ends the connection by sending a logout command. The specification of this behaviour in the π-calculus is as follows:

```
Client(login,request,logout) =
      login!(). ClientConnected(request,logout)

ClientConnected(request,logout) =
      (reply) request!(query,reply). reply?(result).
              ClientConnected(request,logout)
    + logout!(). 0
```

On the other hand, suppose that when the Server receives a request, it delegates the request to a Daemon, which is in charge of serving it. When the connection with the Client is closed the Server kills the Daemon:

```
Server(login, request, logout) =
      login?(user). (daemon, kill)
          ( ServerConnected(login, request, logout, daemon, kill)
          | Daemon(daemon, kill) )

ServerConnected(login, request, logout, daemon, kill) =
      request?(query,reply). daemon!(query,reply).
           ServerConnected(login, request, logout, daemon, kill)
    + logout?(). kill!(). Server(login, request, logout)

Daemon(daemon, kill) =
      daemon?(query,reply). (result) reply!(result). Daemon(daemon, kill)
    + kill?(). 0
```

As shown, the Client/Server system can be fully described using the π-calculus, and the description above could be used for the behavioural interface specification of components. However, there are some drawbacks to this kind of description:

- There is no possibility of distinguishing between data values (such as the user name, the query requested, or the result) and channels (such as the link reply passed for getting private replies to a query) —all of them are considered alike in the π-calculus as names. However, it seems reasonable that some kind of type information should be included in the interface description.
- The treatment of names as variables in the π-calculus, and in particular the possibility of sending and receiving link names in messages, makes formal verification of properties infeasible, even for simple protocols as the one shown.
- Since the π-calculus is computationally complete, there is always the temptation for over-specifying the interface, including implementation details (for instance, in the system above this would happen if the made explicit by means of the match operator the conditions on which the Client decides to perform a new request or to logout the connection).

Another approach for modelling the behaviour of the system is the use of a notation based on finite-state automaton. Here, we use Nierstrasz's Regular Types [9].

```
Client = login. ClientConnected

ClientConnected = request. ClientConnected
                + logout. nil

Server = login. ServerConnected

ServerConnected = request. daemon. ServerConnected
                + logout. kill. Server

Daemon = daemon. Daemon + kill. nil
```

Nierstrasz's proposal assumes an object-oriented scenario in which method invocation is the interaction mechanism. Consequently, only method calls (though no return values) are explicitly represented, and action signs are missed. Types for parameters and return values are indicated separatedly, using a short of IDL language:

```
login(TString)   -> Ok
request(TString) -> TService
daemon(TString)  -> TService
quit()           -> Ok
```

where TService is the (data) type of the query's result, TString represents the type for strings, and Ok prepresents the void type.

From the Regular Types specification above we can see that some problems of π-calculus interfaces are solved here. The use of types —instead of data— for describing parameters and return values, serves both for making the specification clearer, and for ensuring image-finiteness, and a reasonable complexity in property analysis. However, using finite-state automaton, we abstract too much from the full π-calculus specification:

- The composition mechanism is not clear. The lack of a parallel operator makes impossible describing for instance a Client/Server composed by three Clients, a Server and two Daemons.
- The predefined interaction mechanism is too restrictive, and general message-passing interactions should be allowed.
- Neither mobility, process creation, or reference passing can be expressed. The relations between the Server and the Daemon (i.e. the facts that the Server creates Daemons upon Client's connections, and delegates the reply of requests to these Daemons) are missing in the specification.
- States have no parameters. Although this serves for ensuring image-finiteness, it may be too restrictive for specifying certain systems.
- Regular types are fully non-deterministic, without any possibility for making explicit the responsibilities for action and reaction.

To sum up, using a process algebra like the π-calculus would lead to specifications too detailed, and too complex for analysis (and notice that same can

be said of other computationally complete proposals of interface languages like BPEL4WS or WSCI). On the other hand, formalisms based on finite-automata are not enough expressive and tend to simplify too much protocol descriptions, as can be found also in more recent works (see for instance [**?**]). An approach that is somehow in the middle of process algebras and finite automaton is that of behavioural types, which have been proposed for *typing* concurrent systems (similarly as signature IDLs are used for typing objects or components).

The works on typing process algebras started more than a decade ago [11], and there are still new proposals being presented, overcoming the deficiencies or restrictions of the preceding ones. One of the recent proposals in this field is that of Process Types [5]. The specification of the Client/Server example using Process Types would be:

```
TServer =
      login?[TString]. (daemon)( TServerConnected | TDaemon )

TServerConnected =
      request?[T1]. TServerConnected + logout?[]. kill![]. 0

TDaemon =
      daemon?[T2]. TDaemon + kill?[]. 0

TClient =
      login![TString]. TClientConnected

TClientConnected =
      request![T1]. ( reply?[TService]. TClientConnected
                    | daemon![T2]. reply![TService] . 0 )
    + logout![]. 0
```

where TService, TString represents the same types as before, and the types T1, T2 represent anonymous internal types constructed while typing the system.

Notice that in the specification above, the expressiveness of process algebra is combined with the use of types (instead of data) for message parameters, making analysis feasible. Furthermore, the particular treatment of mobility allows to type the π-calculus using CCS (which has no mobility), and therefore, is much easier to analyze. Indeed, there is no name-passing in Process Types, and when a process sends a link to another one (like the reply channel in the example above), this behaviour is approximated by putting in the continuation of the output action how the link is used by the receiving process. This is the reason why the type TClientConnected contains actions corresponding to the Server and the Daemon (daemon! and reply!, respectively). However, notice that both the π-calculus process specification of the system shown in the first place, and the corresponding CCS typed specification above perform the same reduction steps.

In the following section we will described in more detail several relevant proposals for behavioural types.

3 Behavioural types: existing proposals

In order to describe the main features of existing proposals related with behavioural types, we have selected those we consider relevant for our discussion, which is contextualized in the field of components/Web services interoperability. Following subsections only try to give an idea of the most relevant and distinguishing characteristics of the models we are considering, and they cannot be self-contained because of lack of space. For a full understanding of these typing systems we refer to original works referenced in each case.

3.1 Action types

Yoshida, Honda and Berger proposed a typing systems for an asynchronous version of π-calculus [15] where the property of strong normalization is guaranteed by typability. This property, mainly referred to in typed λ-calculi, ensures that a computation necessarily terminates regardless of evaluation strategy. From a logical perspective this is especially interesting because strong normalization of certain λ-calculi implies consistency of the corresponding logical systems. However, this property has also relevant consequences in the context of communicating processes, because under certain conditions, it ensures the safe termination of computations, and the return of expected answers. In addition, it allows a finite axiomatisation of weak bilimilarity, which could give the possibility of making feasible the automated analysis of process equivalence.

Part of this work [15] combines methods coming from the field of λ-calculus and linear logic, with process-theoretic reasoning, and authors argue it is adaptable to other systems involving state, non-determinism, polymorphism and other extensions.

The source language to be typed with this proposal is the first-order linear π-calculus with free name passing, where the *branching* of input actions is restricted to a sum of inputs on the same channel, denoted by $x[\&_i y_i.P_i]]$ (where x and y_i are channel names, P_i are processes, and i ranges on a finite set of indexes), and the *selection* is an output with the syntax $\bar{x}\mathtt{in}_i(y)Q$. The behaviour of these two constructs is given by the following transition rule:

$$x[\&_i y_i.P_i]]|\bar{x}\mathtt{in}_i(y)Q \longrightarrow (\nu y)(P_i\{y/y_i\}|Q)$$

The main ideas behind this proposal are the notions of *action mode*, *channel type*, and *action type*. Action modes are a attached to types to ensure the linearity of channels, denoting how they are used in typed processes. There are two input modes associated with input actions, depending on whether they are single ($\downarrow$) or replicated (!). Then, other two output modes are considered corresponding to each of the input action modes ($\uparrow$ and ?, respectively). Thus, a process like $Fw(ab) = !a(x).\bar{b}(x)$, which repeatedly forwards inputs received by a to b, instead of being typed as $\vdash Fw(ab) \rhd a : (t), b : (t)$ (where t is the type of data passing through a and b), is typed attaching the mode to each action: $\vdash Fw(ab) \rhd a : (t)^!, b : (t)^?$. If the process is not replicated, then the associated type would be:

$a : (t)^{\downarrow}, b : (t)^{\uparrow}$. A mode duality is defined as a self-inverse mapping ($\bar{\uparrow} = \downarrow, \bar{!} = ?$), and it provides a notion of type composability.

Channel types are defined as an alternative combination of input and output modalities on types, introducing a special channel type ($\updownarrow$) which represents the uncomposability of linear channels (when a process includes a channel typed by $\updownarrow$, then it cannot be composed with any other process containing a free occurrence of this channel, e.g. $\vdash x().0|\bar{x}() \rhd x : \updownarrow$). New input channel types (t_I) can be constructed by summing (&) output channel types ($[\&_i t_{Oi}]^{\xi}$, $\xi \in \{\downarrow, !\}$), and new output channel types (t_O) can be obtained by combining (linear additive operator) input channel types ($[\oplus_i t_{Ii}]^{\xi}$, $\xi \in \{\uparrow, ?\}$).

Finally, action types are defined as finite acyclic directed graphs whose nodes have the form $x : t$ such that no names occur twice in the graph, and whose edges present the following syntax: $x : t \rightarrow x' : t'$, where the outermost mode of t is always an input mode, and the outermost mode of t' in an output mode. Intuitively, nodes represent channel typing and edges denote causality of sequential operator. For instance, $\vdash !u(c).\bar{x}(e)e().\bar{c}() \rhd u : (()^{\uparrow})^{!} \rightarrow x : (()^{\downarrow})^{?}$. Action types can be composed in such a way two intuitive properties are preserved: (1) once input-output linear channels are composed, the channel becomes uncomposable, and (2) a server should be unique, but an arbitrary number of clients can request interactions on it.

Considering the previous ingredients, the linear typing system defined in [15] ensures strong normalisability by typability, and a basic interaction-based liveness property in linear processes can be proved: *"if a typed agent calls another replicated typed agent and waits for its answer at a linear channel x, then an answer is guaranteed to eventually arrive at x, however complex intermediate interaction sequences would be"*.

This typing system was also used by the same authors to define a new notion of process equivalence: *linear bisimilarity* [16]. To do this *affecting* and *enabled* types are defined in order to detect whether typable actions affect the environment non-trivially, and whether these actions are guaranteed to take place, respectively. The new notion of bisimilarity is strictly larger than the standard one, but preserves semantic soundness, and it can be applied to develop a behavioural theory of *secrecy* in the π-calculus, improving previous works on the field of security information and process calculi.

3.2 Session types

Session types where defined by Honda, Vasconcelos and Kubo in [3], and present some important features that distinguish them from processes written in a general process algebra like the π-calculus:

- session types abstract from data values, referring to the corresponding data types instead;
- sessions are limited to diadic communications between two components;
- mobility is expressed by means of explicit `throw`/`catch` actions, and since sessions are diadic, once a process throws a session, it cannot use it anymore;

– no mixed alternatives are allowed: input and output actions cannot be combined in a non-deterministic choice.

Under this approach, a program is considered as a collection of sessions, each one being a chain of diadic interactions. Each session is designated by a private channel, through which interactions belonging to that session are performed. The use of diadic sessions for the specification of software interaction allows a modular specification of complex systems. The objective is to provide a basic means to describe complex interaction behaviour with clarity and discipline at a high-level of abstraction, together with a formal basis for analysis and verification.

In this framework, a notion of duality among (session) types, similar to that defined in [15], is introduced in [13] based on the idea of subtyping. Intuitively speaking, a session type α is a subtype of β if α can be used in any context where β is used, and no error occurs in the session. Basically, this means that α should have more —or equal— branchings (input alternatives), and less —or equal— selections (output alternatives). Based on this subtyping relation, a notion of compatibility can be defined, ensuring successful composition of the corresponding processes when they can be correctly typed.

The notion of session has also been used in [2] to ensure safe adaptation of software components, assuming the following scenario: sessions feature a modular projection of component behaviour both in space and in time, and each session provides a partial view of the behaviour of a component (w.r.t. another component that will be attached to it), thus partitioning the component behaviour into several facets or roles; on the other hand, each session describes a (possibly finite) connection, thus partitioning the full life of a component into a sequence of sessions. In this context, a property named *session-safety* is introduced to relax the standard notion of deadlock-freedom, in such a way that session safety only guarantees that a process does not deadlock in the middle of the computation of a session; in other words, once a session is open, then it will finish without deadlocks (i.e. deadlocks are limited to sessions). In [2], it is proved that, under certain conditions, the parallel composition of processes is session-safe.

3.3 Usage (and process) types

The works by Kobayashi and collaborators are among the most relevant in the literature of behavioural types, where their publications can be traced back to 1995, at least. Here, we are going to refer to two of their proposals: *usage types* [4], and *process types* [5]).

The motivation for usage types is dealing not only with safety properties (such as deadlock) but also with liveness properties. Deadlock is a condition by which a system S, formed by the parallel composition of a set of processes, cannot proceed because there is no chance that any pair of processes in S get engaged in a communication action (for instance, all of them are trying to engage in different communication actions). Most type systems in the literature ensure that if a process is *typable* (i.e. a well-formed type can be derived from the

process by applying the typing rules), then deadlock-freedom is preserved, and the process will not get "stuck" in the interaction between its parts. However, a deadlock-free process may be anyway locked, or unable to progress, if it diverges (e.g. if it is performing an infinite loop of internal computations, or of useless communications (e.g. a process that delegates a received request to another process that in turn delegates it to another one and so forth).

For instance, the process $(\nu x)(\ x? \mid (\nu y)(\ y! \mid\ *\,(y?.\ y!\ )\ )$ is deadlock-free. Although the input action on link x will never take place, the entire process is never blocked, performing communications through channel y.

Hence, deadlock-free processes may be livelocked. We can consider that a system of processes is *lock-free*, if it is both deadlock and livelock free (assuming fairness in the scheduling mechanism, for which a process that can be infinitely chosen for participate in a communication will be eventually participate on it).

Consequently, *lock-freedom* is defined in [4] as a combination of both deadlock and livelock-freedom, and a type system for ensuring lock-freedom is presented. The source language used for defining processes is a subset of the π-calculus, deprived from (internal) τ-actions, matching and choice operators (though a specialized *if-then-else* construct exists), and with restricted replication (only applicable to atomic processes). The author argues that these restriction could be removed with further work.

The name of usage types comes from the fact that in this proposal channel types are augmented with tags that contain information about the order in which each channel is used for input or output. This information can be used to guarantee that a process will perform a particular action in a given number of reduction steps, once the action has become available. Hence, it can be determined how long will take a given communication action to succeed.

The proposal in [4] is extended in [5], where a generic type system for the π-calculus is presented. The motivation of this work is to provide a general framework for behavioural types such that various advanced type systems can be derived from the the framework. The framework is instanced by defined a sub-typing relation and a consistency condition.

The main idea of the work is to express type and type environments as *abstract processes* for which a reduction relation is provided (hence, the name process types). A type judgement $\Gamma \rhd P$, which is normally read as "The process P is well-typed under the type environment Γ", is understood here as "the abstract process Γ is a correct abstraction of the process P", (so the typing system can be understood as an abstract interpretation, in the sense that P satisfies a given property if Γ satisfies the corresponding property). Γ is derived from P using the typing rules, and of course, it is easier to check properties on Γ than directly on P.

The process source language used is a subset of the π-calculus, again deprived from τ-actions. However, there are two alternative operators (+ and &), which represent external and internal choice, respectively. Both alternatives may be mixed (combining both input and output actions), though external and internal choices cannot be mixed in the same alternative. Hence, the setting is very

similar to Honda's distinction between branching and selection), though the responsibility for action and reaction is not bound to the sign of the actions.

Unless the π-calculus processes represented in the source language, in the type language there is no creation of new channels, nor channels can be passed through other channels. The reason is to make easier reasoning in the type language than in the π-calculus. However, *mobility* can be typed. Consider for instance the process:

$$x?^{t1}[z].\ z!^{t2} \mid x!^{t3}[y]$$

whose type is:

$$x?^{t1}[\tau] \mid x!^{t3}[\tau]\ .y!^{t2}$$

(where $\tau = (z)z!^{t2}$).

Hence, the type abstracts from the link passing in the synchronized actions on channel x, but makes explicit that an action will be performed on link y (the link sent through mobility), and that this action will be performed only after the synchronization on x.

Actions in their type language are **labelled**, and labels are used for checking properties. For instance, $t.\Gamma$ is the type of a process that behaves as Γ after an (either input or output) action labelled with t is performed.

Several interesting properties (such as subject reduction, normalization of type derivation and type soundness) can be proved for the framework, and these properties hold in any type system instance derived from it. Among the properties that can be ensured for well-formed processes, are absence of arity mismatch in messages, absence of race conditions, safe deallocation of channels, and lock-freedom.

Finally, the type system is guided by the syntax and some indications about how to construct both type-check and reconstruction algorithms are provided

4 Assessment and summary

After presenting the main features of the existing proposals of behavioural types, we now try to assess them according to three main criteria:

- **Type-inference mechanisms.** The behavioural type of a service is aimed at suitably synthesizing and abstracting the interaction protocol of the service, in order to ease the verification of its correct interoperability with other parties. In order to avoid requiring system developers to hand-write the behavioural type of their service, suitable type-inference mechanisms are to be provided so as to allow the automatic generation of the behavioural type of a given interaction protocol. It is easy to foresee that the availability of type-inference mechanisms (for different protocol languages) will be crucial for a future penetration of this technology in the software development world. An important (often under-estimated) issue here is the possibility of separately

typing (in general incomplete) protocols. Indeed, if behavioural types are to be used as part of the service contracts to be exposed in system interfaces, type-inference will have to be applied separately to partial protocols (rather than to a whole system in a single shot).

- **Expressiveness of the source protocol language.** It is important to observe that different type systems may differ also with respect to the source language they refer to. In the case of behavioural types, different systems may differ in the source protocol languages they are aimed at modelling. Such differences obviously affect the applicability of the type system, cutting out protocols that cannot be expressed in the considered source protocol language(s).
- **Provable protocol properties.** Obviously, the set of protocol properties that can be proved by means of a type system is of primary importance to assess the usefulness of the type system itself. Such properties may range from simple connectivity issues (as ensured by available IDLs) to interoperability issues, such as safety or liveness properties. Intuitively speaking, the lower the abstraction of the type-inference mechanisms, the stronger the properties that can be proved on the types associated with protocols. A crucial aspect here is whether the verification of protocol properties can be automated, and which is the complexity of such verification. A further interesting aspect of a type system is whether (and how easily) it allows to state and verify new, user-defined properties.

Table 1 aims at synthesizing how the examined proposals can be classified according to the above stated criteria. To simplify the reading, those criteria have been expanded into seven finer-grained aspects, so as to better highlight similarities and differences among the different approaches. The first aspect concerns the expressiveness of the source protocol language to which the approach can be applied (e.g., full pi-calculus or different restrictions of it). The second and third rows make explicit whether the approaches support the automated and modular type-inference of source protocols. Modular here means the possibility of separately typing parts of interaction protocols (such as a server and a client). The fourth and fifth rows summarise the expressiveness of the language of types employed and the kind of properties that can be proved over such types. Finally, the last row indicate the type of properties that can be proved (in a feasible way) over protocols (e.g., liveness, safety, user-defined).

5 Concluding remarks

In the previous sections we have tried to provide an insightful synthesis of the current state-of-the-art in the development of behavioural types for service integration. The various emerging proposals discussed in Section 3, beyond witnessing the increasing attention devoted to behavioural types, illustrate the application interest of the technical results that have already been achieved in the area. However, much still remains to be done in order to achieve a broad adoption of this technology in the world of software development.

Criteria	Automata	Processes	Action types	Session types	Usage types
Expressiveness of source protocols	—	full	linear π-calculus with free names passing	summation-less asynchronous π-calculus	π-calculus
Automated type inference	—	—	syntax-directed	syntax-directed	syntax-directed
Modular type inference	—	—	partly	—	partly
Expressiveness of type language	low	—	acyclic directed graphs	subset of π-calculus: restricted mobility, binary comm., and no mixed choices	CCS
Type properties	deadlock freedom	—	composability	composability, subtyping	user-defined
Verification of protocol properties	—	unfeasible*	strong normalizat., liveness, security, bisimilarity	session-safety	locks

Table 1. Comparison among different proposals.

- *Type-inference tools.* The availability of type-inference tools will be necessary in order to allow system developers to easily generate the behavioural types of their services, to be exposed in service contracts. It is important to stress that the non-availability of (user-friendly) type-inference tools will require system developers to be acquainted with the technical details of behavioural types, and this will *de facto* simply impede the diffusion of this technology.
- *Type-checking tools.* As pointed out also in [7], the inclusion of behavioural information in service contracts paves the way for the development of powerful mechanisms of service match-making. Namely, service look-up may go beyond simple connectivity issues, and take into account also the effective interoperability of services. Once service behaviour is expressed by means of behavioural types, interoperability can be checked by means of type-checking mechanisms. As for the case of type-inference, the availability of automatic type-checking tools will be a must to enable the development of behaviour-aware service look-up tools.
- *Behavioural conformance and security issues.* An orthogonal, but crucial issue, is the so-called issue of behavioural conformance. Namely, how is it guaranteed that a service or client will effectively behave as per the behaviour declared in its exposed interface? While different solutions to this issue have been already proposed in different contexts, such as wrappers or proof-carrying code, solid and certified guarantees will be needed for a massive employment of this technology for industrial applications.
- *Other aspects.* While behavioural information is necessary to ensure the correct interoperability of services, other aspects are of no less importance, such Service-Level Agreement (SLA) information. The integration of behavioural information with non-functional information will be a further need step to achieve high-quality service aggregations.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, 1997.
2. A. Brogi, C. Canal and E. Pimentel. Behavioural Types and Component Adaptation. In *10th International Conference on Algebraic Methodology And Software Technology. LNCS*. Springer-Verlag. August, 2004.
3. K. Honda, V.T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.
4. N.A. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2): 122–159, 2002.
5. A. Igarashi, N. Kobayashi. A generic type system for the Pi-calculus. *Theoretical Computer Science*, 311(1-2): 121-163, 2004.
6. A. Keller, H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and System Management*, 11(1): 57–81, 2003.
7. L.G. Meredith, S. Bjorg. Contract and types. CACM 46(10), 2003.

8. B. Morel and P. Alexander. Automating component adaptation for reuse. In *Proc. IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 142–151. IEEE Computer Society Press, 2003.
9. O. Nierstrasz. Regular Types for Active Objects, In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Eds.), pages 99–121, Prentice Hall, 1995.
10. M.P. Papazoglou and D. Georgakopoulos: "Service-Oriented Computing", *Communications of the ACM*, 46(10), 2003.
11. B. Pierce, D. Sangiorgi. Typing and Subtyping for Mobile Processes, In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 376–385, IEEE Computer Society Press, 1993.
12. A. Vallecillo, J. Hernández, and J.M. Troya. New issues in object interoperability. In *Object-Oriented Technology*, *LNCS* 1964, pages 256–269. Springer, 2000.
13. A. Vallecillo, V.T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. *ENTCS*, 68(3), 2003.
14. D.M. Yellin and R.E. Strom. Protocol specifications and components adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.
15. N. Yoshida, K. Honda, M. Berger. Strong Normalisation in the π-Calculus . Available as Technical Report at `http://www.doc.ic.ac.uk/~yoshida/paper-ic.html#TYPES`. 2001. To appear in *Journal of Information and Computation*.
16. N. Yoshida, K. Honda, M. Berger. Linearity and bisimulation. In *Proc. of FoSSaCS02, LNCS*, 2303, pp. 417-434, 2002.