Configuring Cloud-Service Interfaces Using Flow Inheritance

Pavel Zaichenkov Olga Tveretina Alex Shafarenko
University of Hertfordshire, United Kingdom
{p.zaichenkov,o.tveretina,a.shafarenko}@ctca.eu

Technologies for composition of loosely-coupled web services in a modular and flexible way are in high demand today. On the one hand, the services must be flexible enough to be reused in a variety of contexts. On the other hand, they must be specific enough so that their composition may be provably consistent. The existing technologies (WS-CDL, WSCI and session types) require a behavioural contract associated with each service, which is impossible to derive automatically. Furthermore, neither technology supports flow inheritance: a mechanism that automatically and transparently propagates data through service pipelines. This paper presents a novel mechanism for automatic interface configuration of such services. Instead of checking consistency of the behavioural contracts, our approach focuses solely on that of data formats in the presence of subtyping, polymorphism and flow inheritance. The paper presents a toolchain that automatically derives service interfaces from the code and performs interface configuration taking non-local constraints into account. Although the configuration mechanism is global, the services are compiled separately. As a result, the mechanism does not raise source security issues despite global service availability in adaptable form.

1 Introduction

Web services is a technology that facilitates development of large scale distributed applications. The application is composed from stand-alone services provided by various organisations on the web. However, consistent composition and reliable coordination of such application still remain challenging for the following reasons: 1) the services are provided as loosely coupled black boxes that only expose their interfaces to the environment; 2) interacting services are not usually known in advance: web services are dynamically chosen to fulfil certain roles and are often replaced by services with a similar functionality; 3) the nature of the service-based application is decentralised [8].

Over the last decade, a substantial body of research on communication safety in web services has been produced. Typically, service behaviour is defined as a contract specified via a suitable process algebra. Various formalisms for verifying the properties of the consistent composition exist. For example, in session types [2] a composition is defined as a global protocol in terms of the interactions that are expected from the protocol peers, and a set of local protocols, one for each peer, which describe the global protocol from the viewpoint of an individual peer. Another formalism is called interface automata [5]. It provides a method that constructs complex services with a specific behaviour from simpler services given semantic properties of the simple services. These approaches support guaranteed communication safety, deadlock-freedom and protocol conformance in service-based applications [6].

Flow inheritance. Web services are experiencing transition from batch to stream processing [1]. Under stream processing services often form long pipelines, where a service processes only a part of its input message, with the rest of it bypassed down the pipeline without modification. A mechanism that implicitly redirects a part of a message from input to output of a service is called *flow inheritance* [4]. Although existing technologies for service composition, such as session types, enhance flexibility and reuse of services with subtyping and polymorphism, they lack analysis and configuration capacity for

Submitted to:
iFMCloud 2016

flow inheritance. Providing support for flow inheritance in stateful services is not a trivial problem. A complete transition system, which specifies the behavioural protocol, needs to be analysed to trace the path of the inherited data. In this paper we focus only on stateless services that do not have an internal storage and which produce output messages in response to a single input message.

In [9] we presented a formal method for configuring interfaces in the presence of subtyping, polymorphism and flow inheritance. The method is based on CSP and SAT. In our approach, the interface of a single service is defined in generic form that is compatible with a variety of contexts. The main features of the generic interface are 1) free variables that implement parametric polymorphism by instantiation to particular context-specific values; 2) Boolean variables that are used to control the dependencies between any elements of interface collections in a way similar to intersection types [7] (those map input type to output type for an overloaded function); and 3) support of flow inheritance using extensible records and variants, which are implemented in a form of row polymorphism [3], that can be extended with additional elements that propagate data formats and functionality across the global communication graph. We illustrate using a running example introduced in Sect. 3.

This paper presents a toolchain that performs a complete interface configuration. First, it derives the interfaces from the C++ code of individual services. We demonstrate that in our approach the complete interfaces can automatically be derived from the code. Next, the toolchain produces constraints for the interfaces of interconnected services given the derived interfaces and a specification of the communication graph. Then a CSP is solved and the values assigned to the variables are sent back to the services. Finally, the services are separately compiled using those values inserted in the code. In Sect. 4 we demonstrate that this approach does not raise vulnerability issues in security-critical services, because a service does not require a centralised compilation (in particular, a service can be separately compiled for each context and be exported in binary).

2 Constraint Satisfaction Problem for Web Services

In this section we briefly introduce a novel IDL called Message Definition Language (MDL) as an alternative to WSDL. The MDL is defined as a term algebra and is formally introduced in [9]. Its purpose is to describe flexible service interfaces.

A message is a collection of data entities, each specified by a corresponding *term*. The term is the MDL basic building block. MDL terms are built recursively according to the following grammar:

⟨*term*⟩ ::= ⟨*symbol*⟩ | term-variable | ⟨*record*⟩ | ⟨*choice*⟩
⟨*record*⟩ ::= {[⟨*element*⟩[,⟨*element*⟩]*[| term-variable]]}
⟨*choice*⟩ ::= (:[⟨*element*⟩[,⟨*element*⟩]*[| term-variable]]:)
⟨*element*⟩ ::= ⟨*label*⟩(⟨*bool-expr*⟩):⟨*term*⟩

Each term is either a *symbol*, a term variable or a recursively-defined labelled collection of terms. The symbol represents standard types such as `int` or `string`. Term variables are used to support parametric polymorphism in the interfaces similarly to type variables. A *record* is an extensible, unordered collection of labelled terms, where *labels* are arbitrary identifiers, which are unique within a single record. Records are subtyped covariantly (a larger record is a subtype). A *choice* is an extensible, unordered collection of labelled alternative terms. The difference between records and choices is in width subtyping: *choices* are subtyped contravariantly using set inclusion (a smaller choice is a subtype). We use choices to represent polymorphic messages and service interfaces at the top level (see Sect. 3 and 4 for more details). Records and choices are defined in *tail form*. The tail is denoted by a term variable that represents a term of the same kind as the construct in which it occurs. This is a form of row polymorphism [3], which we use to implement flow inheritance. The data that is matched by the same variable in input and

output interfaces is automatically bypassed to other services in the pipeline (see Sect. 3 for illustration of flow inheritance).

Elements of the collection may contain arbitrary Boolean expressions. They are used to specify relations between input and output interface elements of a single service as illustrated in Sect. 3. The intention is similar to intersection types, which increase the expressiveness of function signatures: the type $(a \to c) \wedge (b \to d)$ maps particular input types to particular output types. By analogy, assume that the term $\{\mathtt{a}(x)\colon \mathtt{int}, \mathtt{b}(y)\colon \mathtt{int}\}$ defines an input interface of a service and the term $\{\mathtt{c}(x)\colon \mathtt{int}, \mathtt{d}(x \vee y)\colon \mathtt{int}\}$ defines the output interface of the service. The Boolean variables x and y declare that the service produces messages with labels c and d only if a message with the label a was received as an input, and a message with the label d is produced if a message with the label b was received. This mechanism allows to trace message dependencies over the communication graph and check message format compatibility.

Def. 1 introduces the seniority relation $\sqsubseteq$ on terms for the purpose of structural subtyping. It specifies the subsumption relation between interfaces of service ports connected with a communication channel. In the sequel we use nil to denote the empty record $\{\ \}$, which has the meaning of unit type and represents a message without any data.

Definition 1. The seniority relation on terms is defined as follows:

1. $t \sqsubseteq \mathsf{nil}$ if t is a symbol or a record;
2. $t \sqsubseteq t$;
3. $t_1 \sqsubseteq t_2$, if for some $k, m > 0$ one of the following holds:
   (a) $t_1 = \{\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_1^k\colon t_1^k\}$ and $t_2 = \{\mathsf{l}_2^1\colon t_2^1, \ldots, \mathsf{l}_2^m\colon t_2^m\}$, where $k \geq m$ and for each $j \leq m$ there is $i \leq k$ such that $l_1^i = l_2^j$ and $t_1^i \sqsubseteq t_2^j$;
   (b) $t_1 = (:\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_1^k\colon t_1^k:)$ and $t_2 = (:\mathsf{l}_2^1\colon t_2^1, \ldots, \mathsf{l}_2^m\colon t_2^m:)$, where $k \leq m$ and for each $i \leq k$ there is $j \leq m$ such that $l_1^i = l_2^j$ and $t_1^i \sqsubseteq t_2^j$.

Connecting two service ports with a communication channel gives rise to a constraint on terms, which may contain Boolean and term variables. Therefore, the problem can be formulated as a constraint satisfaction problem for web services:

Definition 2 (CSP-WS)**.** Given a set of seniority constraints $\mathscr{C}$, a vector of Boolean variables $\vec{f} = (f_1, \ldots, f_l)$ and a vector of term variables $\vec{v} = (v_1, \ldots, v_m)$, find a vector of Boolean values $\vec{b} = (b_1, \ldots, b_l)$ and a vector of terms without variables $\vec{t} = (t_1, \ldots, t_m)$, such that for each constraint $t_1 \sqsubseteq t_2 \in \mathscr{C}$,

$$t_1[\vec{f}/\vec{b}][\vec{v}/\vec{t}] \sqsubseteq t_2[\vec{f}/\vec{b}][\vec{v}/\vec{t}],$$

where $t[\vec{p}/\vec{q}]$ denotes a substitution of variables $\vec{p}$ with values $\vec{q}$ ($|\vec{p}| = |\vec{q}|$). The tuple $(\vec{b}, \vec{t})$ is called a solution.

We designed a fixed-point algorithm that solves the CSP-WS, which is presented in [9]. It has been implemented as a constraint solver, which is used as part of a toolchain presented in Sect. 4.

3 Motivating Example

We use a simple but non-trivial example to illustrate the MDL and configuration mechanism from Sect. 4. The example is known as the *three-buyer use case* and is often called upon to demonstrate the capabilities of session types [6].

Three buyers called Alice, Bob and Carol cooperate in order to buy a book from a Seller. The service interfaces are defined as the MDL terms, which form a seniority constraint once the services get

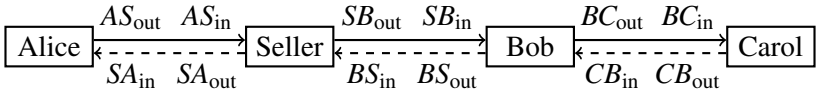

Figure 1: Service composition in a Three Buyer usecase

connected by a channel. Fig. 1 depicts composition of the application where Alice is connected to the Seller only and can interact with Bob and Carol indirectly using flow inheritance. AS, SB, BC, CB, BS, AS denote interfaces that are associated with service output/input ports. For brevity, we only provide AS, SB and BC (the rest of the interfaces are defined in the same manner), which are specified in the following way:

$$
\begin{array}{l|l}
AS_{\text{out}} = (:\text{request: } \{\text{title: } \mathbf{a}\}, & AS_{\text{in}} = (:\text{request: } \{\text{title: } \texttt{string}\}, \\
\quad \text{payment: } \{\text{title: } \mathbf{a}, \text{money: } \texttt{int}, \text{id: } \texttt{int}\}, & \quad \text{payment: } \{\text{title: } \texttt{string}, \text{money: } \texttt{int}\} \mid \mathbf{b}:) \\
\quad \text{share}(x)\text{: } \{\text{title: } \mathbf{a}, \text{money: } \texttt{int}\}, & SB_{\text{in}} = (:\text{share}(z) : \{\text{quote: } \texttt{string}, \text{money: } \texttt{int}\}, \\
\quad \text{suggest}(y)\text{: } \{\text{title: } \mathbf{a}\}:) & \quad \text{response : } \{\text{title: } \texttt{string}, \text{money: } \texttt{int}\} \mid \mathbf{c}:) \\
SB_{\text{out}} = (:\text{response : } \{\text{title: } \texttt{string}, \text{money: } \texttt{int}\} \mid \mathbf{b}:) & BC_{\text{in}} = (:\text{share : } \{\text{quote: } \texttt{string}, \text{money: } \texttt{int}\}:) \\
BC_{\text{out}} = (:\text{share}(z)\text{: } \{\text{quote: } \texttt{string}, \text{money: } \texttt{int}\} \mid \mathbf{c}:) &
\end{array}
$$

AS_{out} specifies an output interface of Alice, which declares functionality and a format of messages sent to the Seller in the following way:

- Alice can request a book's price from the Seller by providing the title of an arbitrary type (which is specified by a term variable **a**) that the Seller is compatible with. On the other hand, the Seller in AS_{in} declares that the title of type `string` can only be accepted, which means that **a** must be instantiated to `string`.
- Furthermore, Alice can provide a payment for a book. In addition to the title and the required amount of money, Alice provides her id in the message. Although the Seller does not require the id, the interconnection is still valid due to the width subtyping supported in the MDL.
- In addition, Alice can offer to share a purchase between other customers using flow inheritance. Although Alice is not connected to Bob or Carol and may even not be aware of their presence, the inheritance mechanism detects that Alice can send a message with "share" label to Bob by bypassing the message implicitly through the Seller. In order to enable flow inheritance in Seller's service, the mechanism sets the tail variable **b** to $(:\text{share: } \{\text{title: } \texttt{string}, \text{money: } \texttt{int}\}:)$. If Bob were unable to accept a message with the "share" label, the mechanism would instantiate x to false, which automatically removes the corresponding functionality from the term.
- Finally, Alice can suggest a book to other buyers. However, examination of other service interfaces shows that there is no service that can receive a message with the label "suggest". Therefore, a communication error occurs if Alice decides to send the message. To avoid this, the configuration mechanism excludes "suggest" functionality from Alice's service by setting the variable y to false.

The proposed configuration mechanism analyses the interfaces of services Seller, Bob and Carol in the same manner. The presence of **b** variable in both input and output interfaces of Bob enables support of flow inheritance on the interface level; even though it is defined locally, the effect of flow inheritance is non-local and potentially involves any number of actors. Furthermore, the Boolean variable z behaves as an intersection type: Bob has "purchase sharing" functionality declared as an element $\text{share}(z)\text{: } \{\ldots\}$ in its input interface SB_{in} (used by the Seller). The element is related to the element $\text{share}(z)\text{: } \{\ldots\}$ in its output interface BC_{out} (used by Carol). The relation declares that Bob provides Carol with "sharing" functionality only if Bob was provided with the same functionality from the Seller. In our example, z is true, because Carol declares that it can receive messages with the label "share". Note that there could be any Boolean formula in place of z, which wires any input and output interfaces of a single service in an

arbitrary way. The existing interface description languages (WSDL, WS-CDL, etc.) do not support such interface wiring capabilities.

4 Interface Configuration Protocol

We developed a complete interface configuration protocol and configuration toolchain for services coded in C++. Choosing C++ allowed us to consider not only web services, but also performance-critical applications. On the other hand, the approach could easily be extended to other languages, which are commonly used for web service development, such as Java or Python.

We defined a fixed interface format for services. Fig. 2a illustrates the implementation code of the Seller from Sect. 3. `request_1` and `payment_1` are processing functions that are called when an input message arrives; the input data is passed with functions' arguments. The processing functions are distinguished from other functions by their return type `service`. On the other hand, output messages are produced by calling special functions called *salvo*s, which are declared by the programmer and must have `salvo` as the return type. The processing function sends output messages by calling salvo functions. The processing functions and salvos have suffixes `_1` or `_2` as part of their names. The suffixes of the processing functions denote the input ports that the functions are associated with. Similarly, the suffix of a salvo denotes a service output port where the messages are sent to. Since port routing is typically specific to the application, association with the ports is not fixed and can be redefined from the *shell*, which is described in Sect. 4. In Fig. 2a, error messages are sent to the second output port, which is considered as an auxiliary one: the application designer can decide whether they want to process error messages or not (in the latter case, the port may remain unwired).

Fig. 2b demonstrates the interfaces that the toolchain derives by analysing the source file. We define a fixed interface format for services as a choice-of-records term. Labels in the choice term of the input interface match processing function names tagged with corresponding labels. As a result, a message is structured as a labelled data record, where the label identifies the function that processes the message. The name of a salvo corresponds to one of the labels in the output choice term. The compatibility of two services connected by a channel is defined by the seniority relation.

The Boolean variables preserve the relation between choice variants in input and output interfaces: response is present in the output interface #1 only if request is present in the input one; similarly, error is present in the output interface #2 only if request or payment is present in the input interface. The intention is to forbid the service from accepting input messages with certain tags if the messages that the corresponding functions produce cannot be accepted by the consumer.

In Fig. 2c the Seller service is augmented with macros, which gets instantiated with particular values after the CSP has been solved. `BV_x` and `BV_y` are macros that correspond to Boolean variables x and y. Depending on instantiations of x and y, unnecessary processing functions and salvos can be removed from the service.

The tail variables are added to records and choices in the interfaces in Fig. 2b for flow inheritance support. For example, the function `request_1` may receive extra arguments that it does not require itself, but the consumers of output messages (i.e. the messages with labels response and error in our example) demand. Such arguments are matched by $\mathbf{a}$ in the input interface and are propagated further in variables $\mathbf{d}$ and $\mathbf{f}$. The constraints $\mathbf{a} \sqsubseteq \mathbf{d}$ and $\mathbf{a} \sqsubseteq \mathbf{f}$ specify the relation between the propagated data: the data that the function `request_1` produces must be compatible with the data that the function receives with its arguments.

In the service code we add macros directives (prefixed with `TV_` in Fig. 2c) to support flow inheritance. For example, assume that the CSP-WS solver concludes that the service that consumes a

```
salvo response_1(string title, int money);
salvo invoice_1(int id);
salvo error_2(string msg);
service request_1(string title) {
    try {
        int price = ...
        response_1(title, price);
    } catch (exception e) {
        error_2(e.what());
    }
}
service payment_1(string title, int money) {
    try {
        int invoice_id = ...
        invoice_1(invoice_id);
    } catch (exception e) {
        error_2(e.what());
    }
}
```

(a) The code of the Seller service

IN #1: (:request(x): {title: string|**a**},
payment(y): {title: string, money: int|**b**} | **c**:)
OUT #1: (:response(x): {title: string, money: int|**d**},
invoice(y): {id: int|**e**} | **c**:)
OUT #2: (:error($x \vee y$): {msg: string|**f**}:)

(b) One input and two output interfaces derived from the Seller service's code. Additional constraints $\mathbf{a} \sqsubseteq \mathbf{d}$, $\mathbf{a} \sqsubseteq \mathbf{f}$, $\mathbf{b} \sqsubseteq \mathbf{e}$, $\mathbf{b} \sqsubseteq \mathbf{f}$ must be generated

```
#if defined(BV_x)
salvo response_1(string title, int money TV_d_decl);
#endif
#if defined(BV_y)
salvo invoice_1(int id TV_e_decl);
#endif
#if defined(BV_x) || defined(BV_y)
salvo error_2(string msg TV_f_decl);
#endif
#ifdef BV_x
service request_1(string title TV_a) {
    try {
        int price = ...
        response_1(title, price TV_d_use);
    } catch (exception e) {
        error_2(e.what() TV_f_use);
    }
}
#endif
#ifdef BV_y
service payment_1(string title, int money TV_b) {
    try {
        int invoice_id = ...
        invoice_1(invoice_id TV_e_use);
    } catch (exception e) {
        error_2(e.what()  TV_f_use);
    }
}
#endif
```

(c) The code for the Seller service augmented with preprocessor's directives. `BV_...` and `TV_...` are macro names

Figure 2: Illustration of the configuration mechanism using the Seller service's code. Every term/Boolean variable in the interface has a corresponding macro definition in the augmented code

message with tag response requires information about an author of the requested book and the producer of the function `request_1` may actually provide this information. Then the solution is $\mathbf{a} = \mathbf{d} = \{\text{author: string}\}$, and `TV_a` and `TV_d` are defined as "`, string author`" and "`, author`" respectively.

The Shell. So far the flexibility of our approach is impeded by fixed labels in service interfaces. The problem is that the corresponding names in the MDL interfaces of the connected services must be identical for a safe communication. For example, a term that specifies an output message {user_id: int} and the one that specifies an input message {user: int} are not compatible because of the label mismatch. The services cannot be decontextualised because a service developer is forced to modify the services according to the context where the service is used.

To solve the problem, we introduce service structuring that consists of two parts: a *core* that contains an implementation code as presented in Fig. 2a; and a *shell* that is used for (re)routing messages and renaming labels in output messages. Such separation facilitates decontextualisation and service reuse. The core is generic and does not need to be changed to match the context; the shell, however, is specific to the context. Since the number of input/output channels and label names varies from context to context, the shell provides a flexible mechanism for service integration. In the example in Fig. 3, the

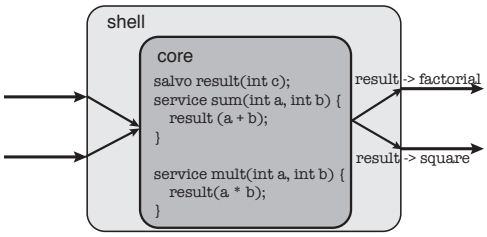

Figure 3: The core and the shell of the service

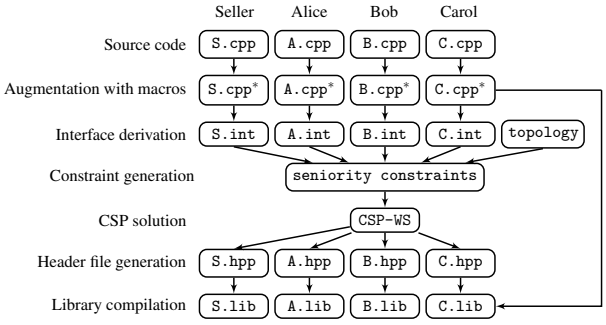

Figure 4: An interface reconciliation workflow performed by the toolchain

core of the service has one input and one output channel and the environment provides two input and two output channels. The shell provides the declaration that messages from both input channels must be merged into one input channel and output messages must be copied to two output channels with label `result` renamed to `factorial` and `square`. As a result, the output interfaces that are derived from the services are associated with each of output channels are (:factorial: {c: `int`}:) and (:square: {c: `int`}:). Without the shell, the interface of the service would be derived using the mechanism from Sect. 4: (:result: {c: `int`}:).

Implementation. The toolchain is developed in C++ and OCaml, also using Clang for the C++ code analysis and the PicoSAT solver as part of the CSP-WS solver. The toolchain configures the service interfaces in five steps as illustrated in Fig. 4:

1. For each file that contains a service source code in C++, augment them with a macros acting as placeholders for the code that enables flow inheritance (as illustrated in Fig. 2c);
2. Derive the interfaces from the code (see Fig. 2a);
3. Given the interfaces and the application topology, construct the constraints to be passed on to the CSP-WS algorithm;
4. Run the CSP-WS solver;
5. Based on the solution, generate header files for every service with macro definitions. In addition, the tool generates the API functions to be called when a service sends or receives a message;
6. Given the modified service source files and the generated header files, create a binary library for every service.

The steps 3–5 are performed by a centralised *composition coordinator* (which could be any service selected arbitrarily) while other configuration steps can be performed independently for each service. The coordinator only receives information about service interfaces.

As a result, the toolchain generates a set of service libraries from the topology and service source files. The libraries can be used with any runtime, which is able to communicate with services using the API. Application execution strategy and resource management are up to designers of a particular web service technology. In addition to flexibility, advantages of the presented design are the following:

- Interfaces and the code behind them can be generic as long as they are sufficiently configurable. No communication between services designers is necessary to ensure consistency in the design.
- Configuration and compilation of every service is separated from the rest of the application. This prevents source code leaks in proprietary software running in the Cloud.[1]

[1] which is otherwise a serious problem. For example, proprietary C++ libraries that use templates cannot be distributed in binary form due to restrictions of the language's static specialisation mechanism.

5 Conclusion and Future Work

We have presented a static interface configuration mechanism of service-based applications based on CSP and SAT. We rely on the Message Definition Language that defines a term algebra and is used for specifying flexible service interfaces. We defined the format of services written in C++ to demonstrate the binding between the MDL and message processing functions. Our approach supports subtyping, polymorphism and flow inheritance, thanks to the order relation defined on MDL terms. We developed a toolchain that automatically derives service interfaces from the code, generates constraints, solves the CSP using an original algorithm and propagates the result with configuration information back to the services.

Currently, our mechanism supports services that respond to at most one message at a time, avoiding any form of message accumulation. This is due to the fact that the linkage between input and output is done using explicit Boolean selectors rather than from a behavioural description such as the component's session type. However, the term type of a component as defined by us could be the end-result of a derivation based not only on the analysis of the source code but also on the component's session type, in which case it could be possible to *derive* the Boolean selectors from it. An integration between MDL and session types is in our future plans.

Our results may prove useful to the software-as-a-service community, because our mechanism supports more flexible interfaces than are currently available without exposing the source code of proprietary software behind them. Building services the way we do could enable service providers to configure a solution for a network customer based on the services that they have at their disposal as well as those provided by other providers and the customer themselves, all solely on the basis of interface definitions and automatic tuning to non-local requirements.

References

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom & Sam Whittle (2013): *MillWheel: fault-tolerant stream processing at internet scale*. *Proceedings of the VLDB Endowment* 6(11), pp. 1033–1044.

[2] Mariangiola Dezani-Ciancaglini & Ugo De'Liguoro (2010): *Sessions and session types: An overview*. In: *Web Services and Formal Methods*, Springer, pp. 1–28.

[3] Benedict R Gaster & Mark P Jones (1996): *A polymorphic type system for extensible records and variants*. Technical Report.

[4] Clemens Grelck, Sven-Bodo Scholz & Alex Shafarenko (2010): *Asynchronous stream processing with S-Net*. *International Journal of Parallel Programming* 38(1), pp. 38–67.

[5] Seyyed Vahid Hashemian & Farhad Mavaddat (2005): *A graph-based approach to web services composition*. In: *Applications and the Internet, 2005. Proceedings. The 2005 Symposium on*, IEEE, pp. 183–189.

[6] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. *ACM SIGPLAN Notices* 43(1), pp. 273–284.

[7] Benjamin C Pierce (1997): *Intersection types and bounded polymorphism*. *MSCS* 7(2), pp. 129–193.

[8] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne & Xiaofei Xu (2014): *Web services composition: A decades overview*. *Information Sciences* 280, pp. 218–238.

[9] Pavel Zaichenkov, Olga Tveretina & Alex Shafarenko: *A constraint satisfaction method for configuring non-local service interfaces*. To appear in *iFM 2016* proceedings. Available at `http://arxiv.org/abs/1601.03370`.