Noname manuscript No.
(will be inserted by the editor)

Non-Local Configuration of Component Interfaces by Constraint Satisfaction

Olga Tveretina · Pavel Zaichenkov · Alex Shafarenko

Received: date / Accepted: date

Abstract Service-oriented computing is the paradigm that utilises services as fundamental elements for developing applications. Service composition, where data consistency becomes especially important, is still a key challenge for service-oriented computing.

We maintain that there is one aspect of Web service communication on the data conformance side that has so far escaped the researchers attention. Aggregation of networked services gives rise to long pipelines, or quasi-pipeline structures, where there is a profitable form of inheritance called flow inheritance. In its presence, interface reconciliation ceases to be a local procedure, and hence it requires distributed constraint satisfaction of a special kind.

We propose a constraint language for this, and present a solver which implements it. In addition, our approach provides a binding between the language and `C++`, whereby the assignment to the variables found by the solver is automatically translated into a transformation of `C++` code. This makes the `C++` Web service context compliant without any further communication. Besides, it uniquely permits a very high degree of flexibility of a `C++` coded Web service without making public any part of its source code.

Keywords Interface Configuration · Constraint Language · Constraint Satisfaction

O. Tveretina
Department of Computer Science
University of Hertfordshire
E-mail: o.tveretina@herts.ac.uk

P. Zaichenkov
Department of Computer Science
University of Hertfordshire
Current affiliation: Google Inc
E-mail: zaichenkov@gmail.com

A. Shafarenko
Department of Computer Science
University of Hertfordshire
E-mail: a.shafarenko@herts.ac.uk

1 Introduction

Service-oriented computing is an effective technology that facilitates development of large-scale distributed systems. On the one hand, it enables enterprises to expose their internal business processes as services available to clients on the Internet, and on the other hand, the clients can combine services and reuse them for developing their own applications or constructing more complex services. Service composition reduces the cost and risks of new software development, because the software elements that are represented as Web services can be used repeatedly [42], but it is still a key challenge for service-oriented computing.

Web services are typically developed in a decentralised manner: an organisation which develops a single service is unaware of other services and the context where their service will be used. As a result, services ultimately become rather generic: they may contain numerous algorithms that are compatible with various contexts. When several Web services are harnessed behind a single interface in a service network, and when the services belong to different administrative domains, the problem of data consistency becomes especially important. Even more, the contributing services may not be mutually trustful while being able to self-configure and meet the demands of the eventual user.

We are interested in the situation where a Web service has only partial information about the message it receives, namely the part containing its own input data. The service communicating with its consumers has no knowledge of their requirements because it itself has no knowledge of the opaque part of its input message. All it can do in interactions with its consumers is to communicate its own input and output *constraints*. We focus on the static guarantee of communication *formats* when the message-format constraints are generally non-local.

The network of Web services can be large and generally cyclic, and, therefore, a nontrivial constraint satisfaction problem has to be solved. It should be noted that service constraints can have a very complex nature. Generally speaking, a Web service has a state, and it is, therefore, a transition system. In different states it may be able to accept messages of different kinds, and the output message types may generally depend on the state and the input message type as well. The combination of these factors makes it impossible to satisfy constraints by a single forward or backward calculation from network inputs to network outputs or vice versa. Instead, a nontrivial constraint satisfaction technique is necessary.

We do not concern ourselves with the communication protocol. From our point of view what is important is whether a message of a given type *can* be received or sent in *any* state. Our solution is exact for stateless services where state constraints do not arise at all, and is a conservative approximation for stateful services.

Our contribution is a novel constraint satisfaction technique introduced in the form of a constraint language called Message Definition Language (MDL). In this language unknown format information is present in the form of term variables for which the satisfying assignment is found by our constraint solver. Web services participating in the service network simply send their MDL declarations to the constraint solver without exposing their internal code. The solver sends back a satisfying assignment that then informs the services about what specific formats to use in

the messages, so that the piggybacked information is preserved and flow-inheritance paths are not disrupted. This has an important security implication: services can be co-configured in context without exposing their source code.

The reminder of the paper is organised as follows. In Section 2 we discuss different methods to solve the service configuration problem and potential applicability of the current techniques utilising satisfiability modulo theories. Section 3 explains the motivation for this research by providing an example of simplified service choreography. In Section 4 we present a term algebra called Message Definition Language (MDL), and the related Constraint Satisfaction Problem for MDL (CSP-MDL) is found in Section 5. Section 6 focuses at first instance on the algorithm for solving a CSP-MDL without Boolean variables and consecutively for the whole language. In Section 7 we propose a formal description of a service network topology in the form of a language of combinators. We illustrate our approach with two use case scenarions in Section 8. Section 9 discusses the configuration protocol and implementation, and Section 10 contains concluding remarks.

This work is elaboration of the results presented in [48] extended in particular, by providing a definition of service network topology, a proof of correctness of our approach, details of the implementation and an image classification use case.

2 Related Work and Discussion

In this section we discuss, on the one hand, different approaches for solving the problem of service composion and their limitations, and, on the other hand, potential applicability of the satisfiability module theories.

Basic technologies such as XML, SOAP, WSDL provide means to describe, and invoke services as one entity in its own right. Two main approaches here are service orchestration and service choreography. Service orchestration represents a single business process that coordinates the interaction among different services, and where the orchestrator is responsible for invoking and combining the services [20]. Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints [37,47].

Laneve and Padovani formalised a subcontract preorder relation in [31] and developed a subcontracting formalism which is based on two observations: 1) it is safe to replace a service that exposes a contract with a “more deterministic” one; 2) it is safe to replace a service that exposes a contract with another one that offers greater capabilities. This research helped to develop language-independent mechanism for assisting service developers to check service behaviour compatibility [2].

Yet another attempt to solve the behavioural consistency problem for Web services is based on session types [25]. Thus, Castagna, Gesbert and Padovani provide a foundation for behaviour subtyping in Web services [14] which facilitates software reuse. Moreover, Carbone, Honda and Yoshid in [11] and Honda, Yoshida and Carbone in [25] present work on multiparty session types. With this methodology, one can check whether the code is functionally correct and does not lead to communica-

tion errors. Although multiparty session types is a non-local mechanism for service composition, it does not support flow inheritance.

The above approaches have a serious limitation though: a behaviour protocol description for a service cannot be derived automatically from the service code. A service developer must formalise the behaviour and explicitly provide a protocol description in addition to the code. In this regard, two issues arise that compromise correctness of the behaviour: 1) a protocol description must correspond precisely to the provided code; 2) the protocol may require modification if the service implementation has changed. Both issues lead to a potential mismatch between a protocol and actual behaviour.

Conversely, the problem of service compatibility can be addressed from the perspective of the interfaces. A service interface provides the specification of a data format and the service functionality. The interfaces are less generic than behaviour protocols, and, therefore, matching the service interface to the service implementation is easier than matching protocols to the code.

In order to overcome the limitations of the above methods, we propose a constraint language defined in Section 4. To solve the related constraint satisfaction problem, we considered applicability of the solvers supporting the following (satisfiability module) theories.

Propositional logic. Various tools used for solving configuration problems rely on propositional logic. Clearly, due to presence of Boolean variables in our language, propositional logic presents a promising formalism. However, our constraint language also includes term variables with an infinite domain, and it limits the usage of propositional logic as it would require developing costly transformations.

Equality logic with uninterpreted functions. Uninterpreted functions and constants can be used to represent atomic terms, such as symbols or integers, and tuples. However, uninterpreted functions fail to encode terms that support flow inheritance based on a set-inclusion (records and choices). Furthermore, equality logic is not suitable for specifying the seniority relation on terms.

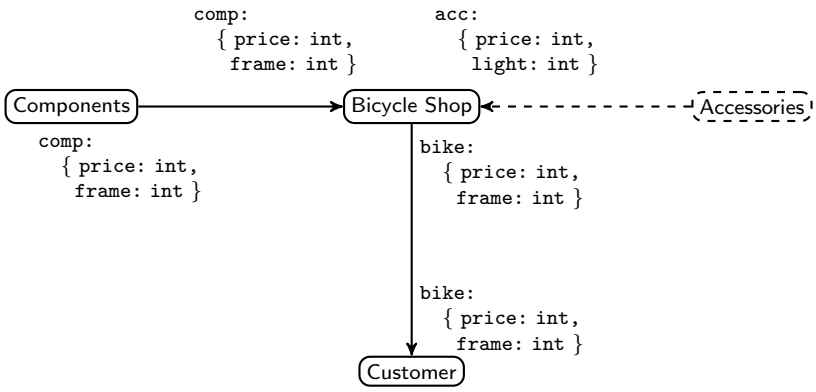
Bit vectors. Bit vectors cannot encode all the properties of records and choices. Thus, hierarchical structuring of bit vectors is not supported, and therefore, support for records and choices that may contain each other as subterms is impossible.

To sum up, theories commonly supported by the satisfiability module solvers fail to encode the terms of our language. Moreover, extending SMT-LIB [4][1] would require significant efforts dictated by the non-trivial definition of well-formedness and, in particular, by the definition the seniority relation for records, choices and switches. On the other hand, approaches taking advantage of the structure of a problem can potentially benefit from it and, as a result, be more efficient.

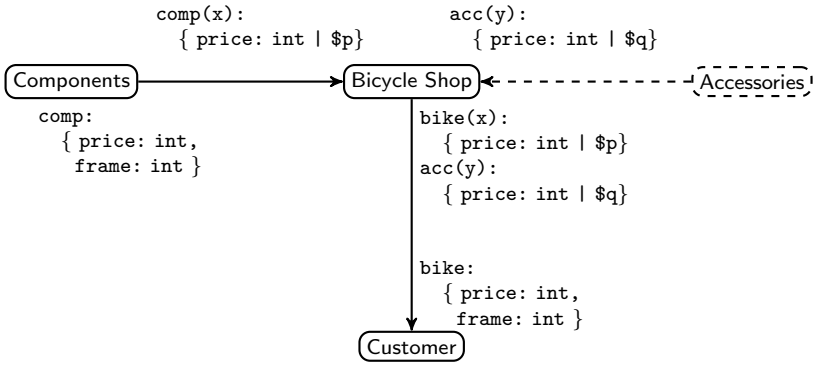
3 Motivation Example: A Service-based Application

Service functionality is exposed in the interface by specifying the operations it can perform, supported data formats, etc. The interacting services are required to have

[1] A language used for formal specification of constraints in SMT solvers (such as Z3, for instance [17]).



(a) Interfaces specified in one of the existing IDLs



(b) Flexible interfaces specified in the IDL augmented with two sorts of variables

Fig. 1: A service-based application that illustrates advantages of introducing variables to the existing IDLs

compatible interfaces in order to prevent protocol errors. The existing composition technologies only provide mechanisms for a pairwise service connection without taking the rest of the topology into account. Neither can services automatically adapt to changes in other services.

Those issues that arise in Web service composition are best illustrated with an example. Thus, the service-based network in Fig. 1 illustrates a purchasing system scenario in which an online bicycle shop sells bicycles and accessories. The suppliers provide the shop with available configurations. After computing the final price, the shop sends a quotation to the customer. That is, Fig. 1a depicts a simplified service choreography with interfaces that can be specified in WSDL or any other of the existing Interface Description Languages (IDLs), such as Google Protocol Buffers [21] or Apache Thrift [43, 3].

The interfaces in a pair of communicating services are identical: a message that the consumer expects should have precisely the same format as the one declared by the producer [1]. Assume that the programmer of the service Components decides to add more information to the output about a bicycle. It forces programmers for the Bicycle Shop and the Customer services to change the interfaces too, even though the Customer service does not directly interact with the Components service.

In our approach reusability of services is improved by introducing *term variables*. Such variables provide support for *parametric polymorphism*. If a term variable is used as part of the interface, then the service declares that any format is acceptable in place of it. A variable can also be present in a record as a "tail". This is a form of *row polymorphism* that implements *flow inheritance* essential for pipelined data processing. Thus, the variable `$p` in the example is present in the input and output interfaces of the Bicycle Shop service. When Bicycle Shop does not require this element, it still can be forwarded directly to the Customer service using flow inheritance.

Web services are typically tightly coupled, which contradicts to the concept of service-oriented architecture. This behaviour is caused by an *impedance mismatch* problem between objects and tree structures in XML and JSON [30]. Due to differences in the data models and the type systems between the object-oriented paradigm and structural one, it is difficult to preserve object properties after serialisation and deserialisation. Our example shows that nevertheless *subtyping* would be useful for Web service interoperability [12,32,35]. The existing Web service models, such as SOAP and REST, fail to preserve the subtyping property while respecting the loose coupling principle [1].

Another problem may arise in the verification of communication safety. Assume that the Accessories service is unavailable. At first glance it may seem to be the case that both networks in Fig. 1 are inconsistent, because the Bicycle Shop service cannot provide accessories to the customer if the latter wishes to buy them. However, in Fig. 1b the interfaces keep track of operation dependencies using the Boolean variables `x` and `y`. Since the interface for the service Customer declares that the client buys only bicycles and not accessories, the variable `x` will be set to true, which automatically requires the presence of the operation `comp` in the input interface of the Bicycle Shop service. The variable `y` is automatically required to be set to false. The analysis of the network in Fig. 1b can infer which operations are not used in the context, while such analysis cannot be applied to the network in Fig. 1a.

Interface variables provide facilities similar to `C++` templates. Services can specify a generic behaviour that is compatible with multiple contexts and input/output data formats. Given the context, the compiler configures the interfaces based on the requirements and capabilities of other services.

4 Message Definition Language

The common way to discuss the properties of constraint types is the mathematical theory of relations and their associated algebras [39]. A term algebra called Message Definition Language (MDL) is defined below as well as seniority relation on well-formed terms.

4.1 Overview

The purpose of the MDL is to describe flexible service interfaces. WSDL is an XML-based *de facto* standard for specifying the interfaces. On the other hand, in order to be able to keep the MDL interfaces short and concise, we decided not to use the XML for specification. Instead, we use an algebraic notation which is short and conventional. Although we use a syntax for MDL terms that is different from the appearance of standard WSDL-based interfaces, it can easily be rewritten as a WSDL extension.

Furthermore, the MDL is part of the type language for a language of combinators introduced in Section 7. We define a service network as a set of services that are connected by communication channels. The interconnection is specified in the language of combinators (a combinator declares a serial, parallel, or a wrap-around connection) with a type associated with each combinator. The type consists of a set of constraints on the MDL terms associated with the given combinator. The intention of the term is to represent

- a standard atomic type such as int, string, etc.; or
- an inextensible data collection such as a tuple; or
- an extensible data record, where additional named fields can be introduced without breaking the match between the producer and the consumer;
- a data-record variant, where generally more variants can be accepted by the consumer than the producer is aware of, and where such additional variants can be inherited from the output back to the input of the producer.

Term variables are used for supporting polymorphism and flow inheritance [22, 23]. Term variables facilitate the reusability of services by enabling generic interfaces. Furthermore, we introduce Boolean variables as part of a term. They are used to specify dependencies between elements of the interfaces. Finally, we define a seniority relation on terms that corresponds to the relation between the data producer and the data consumer.

4.2 Terms

Each term is either atomic or a collection in its own right. Atomic terms are *symbols* that are identifiers used to represent standard types such as `int`, `string`, etc. To account for subtyping[2] we include three categories of collections:

- *tuples* that have to be of the same size and thus admit only depth structural subtyping;
- *records* that are subtyped covariantly (a larger record is a subtype); and
- *choices* that are subtyped contravariantly using set inclusion (a smaller choice is a subtype).

[2] We use "subtyping" to refer to a relation on algebraic terms, where covariance is a feature which allows to substitute a subtype with supertype, and contravariance is a feature which allows to substitute a supertype with subtype. More details on covariance and contravariance can be found, for example, in [13].

Informally, a *tuple* is an ordered collection of terms and a *record* is an extensible, unordered collection of guarded labelled terms, where *labels* are arbitrary symbols that are unique within a single record. A *choice* is a collection of alternative terms. We use choices to represent polymorphic messages and service interfaces at the top level. The syntax of a choice is the same as that of a record except for the delimiters, and the difference is in width subtyping [7]. Briefly, the subtyping on records is defined so that a record with *more* elements is a subtype of a record with *less* elements. Similarly, a choice with *less* elements is a subtype of a choice with *more* elements.

We introduce Boolean variables (called b-variables later) as part of the term interfaces. They provide functionality similar to intersection types [38] and increase the expressiveness of function signatures. More specifically, this kind of variables serve the following purposes:

- b-variables allow to define configurable interface, and
- b-variables are used to specify dependencies between input and output data formats.

A Boolean expression g is a guard. It is defined by the following grammar, where selection of the Boolean operators is justified by the practical needs:

$\langle guard \rangle$ ::= true | false | $(\langle guard \rangle \wedge \langle guard \rangle)$ | $\neg\langle guard \rangle$ | $(\langle guard \rangle \vee \langle guard \rangle)$ | $\langle guard \rangle \rightarrow \langle guard \rangle$ | *b-variable*

For coercion of interfaces we distinguish between two term categories: *down-coerced* and *up-coerced* terms. The former ones include symbols, tuples and records, and the letter ones only choices. Informally, for two down-coerced terms, a term associated with a structure with "more data" is a subtype of the one associated with a structure that contains less; and vice versa for up-coerced terms.

In order to support parametric polymorphism and inheritance in interfaces, we introduce term variables, called later *t-variables* that are similar to type variables. Similar to terms, we distinguish between two variable categories: down-coerced variables that can be instantiated with symbols, tuples and records, and up-coerced variables that can only be instantiated with choices (up-coerced terms). We use $v^{\downarrow}$ and $v^{\uparrow}$ for down-coerced and up-coerced variables respectively, and v if the coercion sort is not important.

MDL terms are built recursively using the constructors: tuple, record, choice and switch, according to the following grammar:

$\langle term \rangle$ ::= $\langle symbol \rangle$ | $\langle tuple \rangle$ | $\langle record \rangle$ | $\langle choice \rangle$ | t-variable

$\langle tuple \rangle$ ::= `(`$\langle term \rangle$ [$\langle term \rangle$]*`)`

$\langle record \rangle$ ::= `{`[$\langle element \rangle$[`,`$\langle element \rangle$]*[`|`down-coerced t-variable]]`}`

$\langle choice \rangle$ ::= `(:`[$\langle element \rangle$[`,`$\langle element \rangle$]*[`|`up-coerced t-variable]]`:)`

$\langle element \rangle$ ::= $\langle label \rangle$`(`$\langle guard \rangle$`):`$\langle term \rangle$

$\langle label \rangle$ ::= $\langle symbol \rangle$

The empty record { } has the meaning of unit type and represents a message with no data. Similarly, the empty choice can be used for specifying service interfaces that cannot send or receive any messages. Note that in the following we use

1. nil to denote the empty record { }; and
2. none to denote the empty choice (: :).

Now nil is the maximum element for down-coerced terms, and none is the minimal element for up-coerced terms.

Records and choices are defined in the so-called *tail form*, where the tail is denoted by a t-variable that represents a term of the same kind as the construct in which it occurs. For example, in the term

$$\{\mathsf{l}_1(\mathsf{true})\colon t_1, \ldots, \mathsf{l}_\mathsf{d}(\mathsf{true})\colon t_d \,|\, v^{\downarrow}\}$$

the variable $v^{\downarrow}$ represents the tail of the record, that is its members with the labels l_i such that $\mathsf{l}_i \neq \mathsf{l}_1, \ldots, \mathsf{l}_i \neq \mathsf{l}_d$.

A switch is an auxiliary construct used for building conditional terms and specified as a set of unlabelled guarded alternatives. It allows to define an arbitrary interface as a function of Boolean values. Formally, it is defined as

$\langle switch \rangle$ ::= <$\langle guard \rangle : \langle term \rangle$[, $\langle guard \rangle : \langle term \rangle$]*>

If the guard of an element is equal to false, then the element can be omitted. For example,

$$\{\mathsf{a}(x \wedge y)\colon \mathtt{string}, \mathsf{b}(\mathsf{false})\colon \mathtt{int}, \mathsf{c}(x)\colon \mathtt{int}\} = \{\mathsf{a}(x \wedge y)\colon \mathtt{string}, \mathsf{c}(x)\colon \mathtt{int}\}.$$

If the guard of an element is equal to true, then the guard can be syntactically omitted. For example,

$$\{\mathsf{a}(x \wedge y)\colon \mathtt{string}, \mathsf{b}(\mathsf{true})\colon \mathtt{int}, \mathsf{c}(x)\colon \mathtt{int}\} = \{\mathsf{a}(x \wedge y)\colon \mathtt{string}, \mathsf{b}\colon \mathtt{int}, \mathsf{c}(x)\colon \mathtt{int}\}.$$

4.3 Seniority Relation

The central notion in the study of constraints and constraint satisfaction problems is the notion of a relation [39]. In this section for the purpose of structural subtyping we introduce a seniority relation on terms denoted by $R_{\sqsubseteq}$ (or for simplicity just $\sqsubseteq$). In practice it means the following: If a term t describes the input interface of a service, then the service can process any message described by a term t', such that $t' \sqsubseteq t$.

Given a term t and a guard g, let $\mathsf{V}^b(t)$ and $\mathsf{V}^b(g)$ denote the sets of the b-variables in t and g respectively, and $\mathsf{V}^{\uparrow}(t)$ and $\mathsf{V}^{\downarrow}(t)$ denote the sets of the up-coerced and down-coerced t-variables in t.

Definition 1 (Semi-ground and ground terms) A term t is called semi-ground if $\mathsf{V}^{\uparrow}(t) \cup \mathsf{V}^{\downarrow}(t) = \emptyset$. A term t is called ground if it is semi-ground and $\mathsf{V}^b(t) = \emptyset$.

For example, the record $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{string}\}$ is a ground term, and the record $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}(x)\colon \mathtt{string}\}$ is a semi-ground term. Finally, the record $\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}(x)\colon v\}$ is neither ground nor semi-ground term.

In the following we use $\mathscr{T}^{\downarrow}$ to denote the set of all down-coerced ground terms, and $\mathscr{T}^{\uparrow}$ to denote the set of all up-coerced ground terms. Now $\mathscr{T} = \mathscr{T}^{\downarrow} \cup \mathscr{T}^{\uparrow}$ is the set of all ground terms.

Similarly, $\mathscr{T}_m^{\downarrow}$, $m \geq 0$, denotes the set of all vectors of down-coerced ground terms of length m and $\mathscr{T}_n^{\uparrow}$, $n \geq 0$, denotes the set of all vectors of up-coerced ground terms of length n.

Definition 2 (Well-formed terms) A term t is well-formed if it is ground and exactly one of the following holds:

1. t is a symbol;
2. t is a tuple $\{t_1 \ldots t_d\}$, $d > 0$, where t_i, $1 \leq i \leq d$, is well-formed;
3. t is a record $\{\mathsf{l}_1(g_1)\colon t_1, \ldots, \mathsf{l}_\mathsf{d}(g_d)\colon t_d\}$ or a choice $\{\mathsf{l}_1(g_1)\colon t_1, \ldots, \mathsf{l}_\mathsf{d}(g_d)\colon t_d\}$, $d \geq 0$, where for $1 \leq i \neq j \leq d$, $g_i \wedge g_j \implies l_i \neq l_j$; and t_i is well-formed provided that g_i is true;
4. t is a switch $\langle (g_1)\colon t_1, \ldots, (g_d)\colon t_d \rangle$, $d > 0$, where there is an i, $1 \leq i \leq d$, such that $g_i = \mathsf{true}$ and t_i is well-formed and $g_j = \mathsf{false}$ for any $j \neq i$.

We define the *canonical form* of a well-formed collection as a representation that does not include false guards, and we omit the true guards anyway. The canonical form of a switch is its (only) term with a true guard, hence any term in canonical form is switch-free.

Definition 3 (Seniority relation) The seniority relation $R_{\sqsubseteq}^S$ (or $R_{\sqsubseteq}$ for simplicity) on a set of well-formed terms in canonical form S is defined recursively as follows:

1. $(\mathsf{none}, t) \in R_{\sqsubseteq}^S$ if t is a choice;
2. $(t, \mathsf{nil}) \in R_{\sqsubseteq}^S$ if t is a symbol, a tuple or a record;
3. $(t,t) \in R_{\sqsubseteq}^S$;
4. $(t_1,t_2) \in R_{\sqsubseteq}^S$, if for some $d,e > 0$ one of the following holds:
   (a) $t_1 = \{t_1^1, \ldots, t_1^d\}$, $t_2 = \{t_2^1, \ldots, t_2^d\}$ and $(t_1^i, t_2^i) \in R_{\sqsubseteq}^S$ for each $1 \leq i \leq d$;
   (b) $t_1 = \{\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_1^\mathsf{d}\colon t_1^d\}$ and $t_2 = \{\mathsf{l}_2^1\colon t_2^1, \ldots, \mathsf{l}_2^\mathsf{e}\colon t_2^e\}$, where $d \geq e$ and for each $j \leq e$ there is $i \leq d$ such that $l_1^i = l_2^j$ and $(t_1^i, t_2^j) \in R_{\sqsubseteq}^S$;
   (c) $t_1 = \{\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_1^\mathsf{d}\colon t_1^d\}$ and $t_2 = \{\mathsf{l}_2^1\colon t_2^1, \ldots, \mathsf{l}_2^\mathsf{e}\colon t_2^e\}$, where $d \leq e$ and for each $i \leq d$ there is $j \leq e$ such that $l_1^i = l_2^j$ and $(t_1^i, t_2^j) \in R_{\sqsubseteq}^S$.

In the following we will typically use $t \sqsubseteq t'$ for simplicity to denote that $(t,t') \in R_{\sqsubseteq}$. Moreover, if $\mathbf{t_1}$ and $\mathbf{t_2}$ are vectors of terms $(t_1^1, \ldots, t_d^1)$ and $(t_1^2, \ldots, t_d^2)$ of size d, then $\mathbf{t_1} \sqsubseteq \mathbf{t_2}$ denotes that $t_i^1 \sqsubseteq t_i^2$ for any $1 \leq i \leq d$.

Proposition 1 *Let S be some set of well-formed terms in canonical form. Then the seniority relation $R_{\sqsubseteq}^S$ is a partial order on S.*

Proposition 2 *$(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$ and $(\mathscr{T}_n^{\uparrow}, \sqsubseteq)$ is a pair of meet and join semilattices, that is*

$$\forall \mathbf{t_1}, \mathbf{t_2} \in \mathscr{T}_m^{\downarrow} : \mathbf{t_1} \sqsubseteq \mathbf{t_2} \text{ if and onl if } \mathbf{t_1} \sqcap \mathbf{t_2} = \mathbf{t_1},$$
$$\forall \mathbf{t_1}, \mathbf{t_2} \in \mathscr{T}_n^{\uparrow} : \mathbf{t_1} \sqsubseteq \mathbf{t_2} \text{ if and only if } \mathbf{t_1} \sqcup \mathbf{t_2} = \mathbf{t_2}.$$

Although the seniority relation is straightforwardly defined for ground terms, terms that are present in the interfaces of services can contain t-variables and b-variables. Finding ground term values for the t-variables and Boolean values for the b-variables satisfying seniority relation represents the constraint satisfaction problem for the MDL language, which is formally introduced in the next section.

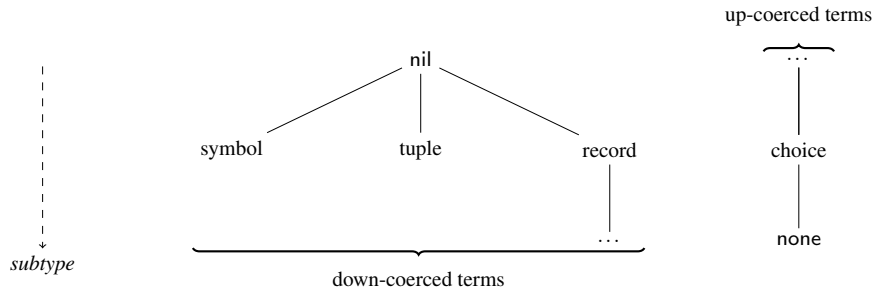


Fig. 2: The semilattices $(\mathscr{T}^{\downarrow}, \sqsubseteq)$ and $(\mathscr{T}^{\uparrow}, \sqsubseteq)$ represent the seniority relation for terms of different categories

5 Constraint Satisfaction Problem for Message Definition Language

The notion of constraint satisfaction problem (or CSP) was introduced by Montanari in 1974 [36] and has been extensively studied ever since. While finite-domain CSPs are a major focus of graph theory, artificial intelligence, and finite model theory [29, 44], a later generalisation to infinite domains enhanced significantly the range of computational problems that can be modelled as a CSP [5].

A CSP represents a problem as a homogeneous collection of constraints formally defined by

1. variables $x_1, \ldots, x_n$ with $n \geq 1$, where each variable x_i has a nonempty domain D_i of possible values; and
2. constraints $C_1, \ldots, C_m$ with $m \geq 1$, where each constraint C_j involves a subset of the variables and constrains the allowable values that can be simultaneously assigned to the variables.

Then the CSP seeks to find an assignment of values $v_1, \ldots, v_n$ to the variables $x_1, \ldots, x_n$ such that it does not violate any constraint.

In general, numerous problems can be expressed as a CSP with the help of a constraint language, including the standard propositional satisfiability problem [16]. The MDL constraint language is defined by a set of terms, where the unknown format information is captured by term variables and the seniority relation on the set of well-formed terms. The domains of the variables are the sets of well-formed terms of the relevant category. The set of constraints is given as a set of pairs of terms $\{(t_1^1, t_2^1), \ldots (t_1^m, t_2^m)\}$. The problem we consider is to find an assignment of variables (provided that one exists) satisfying the seniority relation for each pair of terms.

In the sequel we will use the notation $t \sqsubseteq t'$ which will mean one of the following: (a) a seniority relation on well-formed terms t and t', namely $(t, t') \in R_{\sqsubseteq}$; (b) a constraint on two terms t and t'. Also, by substitution we will mean a syntactic transformation on terms and guards, where a vector of b-variables is replaced by a vector of Boolean values or a vector of t-variables is replaced by a vector of ground or semi-ground terms. That is, for any a, which is either a term or a guard, we use

$a[\mathbf{f}/\mathbf{v}]$ to denote the result of the substitution, where f_i is replaced by v_i for some $\mathbf{f} = (f_1, \ldots, f_k)$ and $\mathbf{v} = (v_1, \ldots, v_k)$. Furthermore, we use $a[\mathbf{f}_1/\mathbf{v}_1, \ldots, \mathbf{f}_m/\mathbf{v}_m]$ as a shortcut for $a[\mathbf{f}_1/\mathbf{v}_1] \ldots [\mathbf{f}_m/\mathbf{v}_m]$.

Now we are in a position to formally define a CSP for the MDL language abbreviated as CSP-MDL.

Definition 4 (CSP-MDL) Let $\mathscr{C}$ be a set of constraints of the form $t_1 \sqsubseteq t_2$, where t_1 and t_2 are terms, defined on the vectors of b-variables $\mathbf{f} = (f_1, \ldots, f_l)$, down-coerced variables $\mathbf{v}^{\downarrow} = (v_1^{\downarrow}, \ldots, v_m^{\downarrow})$ and up-coerced variables $\mathbf{v}^{\uparrow} = (v_1^{\uparrow}, \ldots, v_n^{\uparrow})$.

Then a CSP for MDL, abbreviated as CSP-MDL, asks to find a vector of Boolean values $\mathbf{b} = (b_1, \ldots, b_l)$, and vectors of well-formed terms $\mathbf{t}^{\downarrow} = (t_1^{\downarrow}, \ldots, t_m^{\downarrow})$ and $\mathbf{t}^{\uparrow} = (t_1^{\uparrow}, \ldots, t_n^{\uparrow})$, if ones exist, such that for a constraint $t_1 \sqsubseteq t_2 \in \mathscr{C}$, the following holds

$$t_1[\mathbf{f}/\mathbf{b}, \mathbf{v}^{\downarrow}/\mathbf{t}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{t}^{\uparrow}] \sqsubseteq t_2[\mathbf{f}/\mathbf{b}, \mathbf{v}^{\downarrow}/\mathbf{t}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{t}^{\uparrow}].$$

The tuple $(\mathbf{b}, \mathbf{t}^{\downarrow}, \mathbf{t}^{\uparrow})$ is called a solution of CSP-MDL for $\mathscr{C}$. If $\mathscr{C}$ has no solution, then we say that it as an unsatisfiable set of constraints.

A solution to the CSP-MDL is not necessarily unique, and in the context of Web services, multiple solutions correspond to multiple interface configurations. Deciding whether or not a given CSP instance has a solution is in general NP-complete [33]. We note that due to the presence of arbitrary Boolean constraints on b-variables, the CSP-MDL is NP-complete.

A CSP on a finite domain can typically be solved by constraint satisfaction methods using a form of search, where knowledge about easy problems could possibly serve as a heuristic in the solution of difficult problems [18]. The tools for analysis of a CSP on an infinite domain are based on approaches rooted in various fields of mathematics, including universal algebra, logic, graph theory, and Ramsey theory.

In the next section we discuss our approach to solving the CSP-MDL, including its complexity.

6 Solving CSP-MDL

In this section we present at first instance an algorithm for solving the CSP-MDL without Boolean variables and consecutively for the whole language.

The problem we solve is similar to type inference problem; however, it has large combinatorial complexity, which arises from the presence of Boolean variables in general form. Another problem is potential cyclic dependencies in the network, which prevent the application of a simple forward algorithm.

6.1 General Idea

As a first step, we consider the CSP-MDL when all b-variables are instantiated. This allows us to focus on the resolution of t-variables.

Our algorithm for solving CSP-MDL is a modification of the fixed-point algorithm by Kildall introduced in [28] to globally analyse the program structure in order to perform compile time optimization of object code generated for expressions. The algorithm requires terms to be structured in a lattice or a bounded semilattice. Therefore, we extend the pair $(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$ and $(\mathscr{T}_n^{\uparrow}, \sqsubseteq)$ of the meet and join semilattices, to the lattices $(\widetilde{\mathscr{T}_m^{\downarrow}}, \sqsubseteq)$ and $(\widetilde{\mathscr{T}_n^{\uparrow}}, \sqsubseteq)$ by introducing the top and bottom elements[3] $\underline{\bot}$ and $\overline{\top}$. That is,

$$\widetilde{\mathscr{T}_m^{\downarrow}} = \mathscr{T}_m^{\downarrow} \cup \{\underline{\bot}\} \text{ and } \widetilde{\mathscr{T}_n^{\uparrow}} = \mathscr{T}_n^{\uparrow} \cup \{\overline{\top}\}.$$

Similar to the empty clause in Propositional Logic, both elements $\underline{\bot}$ and $\overline{\top}$ represent absence of a solution and are used later to encode an unsatisfiable set of constraints. That is, if the set of constraints is unsatisfiable, the algorithm returns a pair of the lattices' elements $(\underline{\bot}, \overline{\top})$, which are formally defined as satisfying the following conditions:

$$\underline{\bot} \sqsubseteq \mathbf{a}^{\downarrow} \text{ for any } \mathbf{a}^{\downarrow} \in \widetilde{\mathscr{T}_m^{\downarrow}} \text{ and } \mathbf{a}^{\uparrow} \sqsubseteq \overline{\top} \text{ for any } \mathbf{a}^{\uparrow} \in \widetilde{\mathscr{T}_n^{\uparrow}}.$$

Our algorithm operates on the extended semilattices and performs the following steps:

1. Select the initial approximation (the iteration $i = 0$) as

$$(\mathbf{a}_0^{\downarrow}, \mathbf{a}_0^{\uparrow}) = ((\mathsf{nil}, \ldots, \mathsf{nil}), (\mathsf{none}, \ldots, \mathsf{none})),$$

   where $(\mathsf{nil}, \ldots, \mathsf{nil})$ is the top element of the join semilattice $(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$, and $(\mathsf{none}, \ldots, \mathsf{none})$ is the bottom element of the meet semilattice $(\mathscr{T}_n^{\uparrow}, \sqsubseteq)$[4].
2. Using the iterated function introduced in Section 6.2, compute the next approximation $(\mathbf{a}_{i+1}^{\downarrow}, \mathbf{a}_{i+1}^{\uparrow})$ such that
   (a) $\mathbf{a}_{i+1}^{\downarrow} \sqsubseteq \mathbf{a}_i^{\downarrow}$ and $\mathbf{a}_i^{\uparrow} \sqsubseteq \mathbf{a}_{i+1}^{\uparrow}$, and
   (b) for every constraint $t_1 \sqsubseteq t_2 \in \mathscr{C}$,

$$t_1[\mathbf{v}^{\downarrow}/\mathbf{a}_{i+1}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_i^{\uparrow}] \sqsubseteq t_2[\mathbf{v}^{\downarrow}/\mathbf{a}_i^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_{i+1}^{\uparrow}].$$

   As we show later, these conditions are sufficient to show monotonicity of the approximations.
3. The solution is found if $(\mathbf{a}_{i+1}^{\downarrow}, \mathbf{a}_{i+1}^{\uparrow}) = (\mathbf{a}_i^{\downarrow}, \mathbf{a}_i^{\uparrow})$.

Note that the algorithm computes a series of approximations, which either leads to a solution or to the pair $(\underline{\bot}, \overline{\top})$ corresponding to unsatisfiability.

[3] Such elements are often used in data-flow algorithms. One of the examples is the algorithm for constraint propagation, which is presented in [10].

[4] Recall that nil denotes the empty record { }; and none denotes the empty choice (: :) as it was defined in Section 4.2. That is, nil is the maximum element for down-coerced terms, and none is the minimal element for up-coerced terms.

6.2 Iterated Function

In order to formally define computing the next approximation, we introduce the *iterated function*

$$\mathsf{IF} : \mathscr{C} \times \widetilde{\mathscr{T}_m^{\downarrow}} \times \widetilde{\mathscr{T}_n^{\uparrow}} \to \widetilde{\mathscr{T}_m^{\downarrow}} \times \widetilde{\mathscr{T}_n^{\uparrow}}$$

which maps a single constraint and the current approximation to the next tighter approximation. According to Kildall ([28]), the iterated function is homomorphic if for any constraint $t_1 \sqsubseteq t_2$, from

$$\mathsf{IF}(t_1 \sqsubseteq t_2, \mathbf{a}_1^{\downarrow}, \mathbf{a}_1^{\uparrow}) = (\mathbf{a}_1'^{\downarrow}, \mathbf{a}_1'^{\uparrow}) \text{ and } \mathsf{IF}(t_1 \sqsubseteq t_2, \mathbf{a}_2^{\downarrow}, \mathbf{a}_2^{\uparrow}) = (\mathbf{a}_2'^{\downarrow}, \mathbf{a}_2'^{\uparrow})$$

we can conclude that

$$\mathsf{IF}(t_1 \sqsubseteq t_2, \mathbf{a}_1^{\downarrow} \sqcap \mathbf{a}_2^{\downarrow}, \mathbf{a}_1^{\uparrow} \sqcup \mathbf{a}_2^{\uparrow}) = (\mathbf{a}_1'^{\downarrow} \sqcap \mathbf{a}_2'^{\downarrow}, \mathbf{a}_1'^{\uparrow} \sqcup \mathbf{a}_2'^{\uparrow}).$$

In order to show the homomorphism property it is sufficient to demonstrate that IF is monotonic (Lemma 1) and returns the tightest possible approximation (Lemma 2). The former is important for showing termination of Algorithm 1, and the latter is important for proving its correctness.

Now wee define the function IF for all the categories of terms, but choices as they are symmetrical to records, and switches which are reduced to other term categories. Moreover, we take into account that the following constraints are satisfiable by Definition 3:

- $\mathsf{none} \sqsubseteq t$, where t is a choice;
- $t \sqsubseteq \mathsf{nil}$, where t is a symbol, a tuple or a record; and
- $t \sqsubseteq t$, where t is an arbitrary term.

Furthermore, if both terms t_1 and t_2 belong to different categories (for example, t_1 is a record, and t_2 is a tuple), then any constraint of the form $t_1 \sqsubseteq t_2$ is unsatisfiable.

Let $(\mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow})$ denote the current approximation, where $\mathbf{a}^{\downarrow} = (\overline{a}_1, \ldots, \overline{a}_m)$ and $\mathbf{a}^{\uparrow} = (\overline{\overline{a}}_1, \ldots, \overline{\overline{a}}_n)$, and let $\mathbf{v}^{\downarrow} = (\overline{v}_1, \ldots, \overline{v}_m)$, and $\mathbf{v}^{\uparrow} = (\overline{\overline{v}}_1, \ldots, \overline{\overline{v}}_n)$ be vectors of the down-coerced and up-coerced variables respectively. Now we define the iterated function for the following types of constraints:

1. For constraints on **atomic terms** and **variables** we consider the following cases:
   (a) If t is a down-coerced term and $\overline{v}_l$ is a down-coerced variable, then

$$\mathsf{IF}(t \sqsubseteq \overline{v}_l, \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}) = \mathsf{IF}(t \sqsubseteq \overline{v}_l[\mathbf{v}^{\downarrow}/\mathbf{a}^{\downarrow}], \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}).$$

   (b) If t is an up-coerced term and $v_l^{\uparrow}$ is an up-coerced variable, then

$$\mathsf{IF}(t \sqsubseteq v_l^{\uparrow}, \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}) = (\mathbf{a}^{\downarrow}, (\overline{a}_1, \ldots, \overline{a}_l \sqcup t[\mathbf{v}^{\downarrow}/\mathbf{a}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}^{\uparrow}], \ldots, \overline{a}_n)).$$

   (c) If $\overline{\overline{v}}_l$ is an up-coerced variable and t is an up-coerced term, then

$$\mathsf{IF}(\overline{\overline{v}}_l \sqsubseteq t, \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}) = \mathsf{IF}(\overline{\overline{v}}_l[\mathbf{v}^{\uparrow}/\mathbf{a}^{\uparrow}] \sqsubseteq t, \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}).$$

(d) If $\overline{v}_l$ is a down-coerced variable and t is a down-coerced term, then $\overline{v}_l$ should not be higher than the ground term $t[\mathbf{v}^{\downarrow}/\mathbf{a}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}^{\uparrow}]$ in the meet semilattice:

$$\mathsf{IF}(\overline{v}_l \sqsubseteq t, \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}) = ((\overline{a}_1, \ldots, \overline{a}_l \sqcap t[\mathbf{v}^{\downarrow}/\mathbf{a}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}^{\uparrow}], \ldots, \overline{a}_m), \mathbf{a}^{\uparrow}).$$

2. For a constraint $t_1 \sqsubseteq t_2$, where $t_1 = (t_1^1 \ldots t_k^1)$ and $t_2 = (t_1^2 \ldots t_k^2)$ are **tuples** we have to ensure that the following holds:

$$\mathsf{IF}((t_1^1 \ldots t_k^1) \sqsubseteq (t_1^2 \ldots t_k^2), \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}) = (\bigsqcap_{1 \le i \le k} \mathbf{a_i}^{\downarrow}, \bigsqcup_{1 \le i \le k} \mathbf{a_i}^{\uparrow}).$$

3. For a constraint $t_1 \sqsubseteq t_2$, where $t_1 = \{\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{p}^1\colon t_p^1\}$ and $t_2 = \{\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{q}^2\colon t_q^2\}$ are **records**, we consider two following cases:
   (a) If for all i, $1 \le i \le q$, there exists j such that $l_j^1 = l_i^2$, then the constraint $t_j^1 \sqsubseteq t_i^2$ has to be satisfied, that is :

$$\mathsf{IF}(\{\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{p}^1\colon t_p^1\} \sqsubseteq \{\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{q}^2\colon t_q^2\}, \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}) = (\bigsqcap_{1 \le i \le q} \mathbf{a_i}^{\downarrow}, \bigsqcup_{1 \le i \le q} \mathbf{a_i}^{\uparrow}).$$

   (b) Otherwise, the set of labels in t_2 is not a subset of the labels in t_1 and, therefore, $t_1 \sqsubseteq t_2$ is unsatisfiable:

$$\mathsf{IF}(\{\mathsf{l}_1^1\colon t_1^1, \ldots, \mathsf{l}_\mathsf{p}^1\colon t_p^1\} \sqsubseteq \{\mathsf{l}_1^2\colon t_1^2, \ldots, \mathsf{l}_\mathsf{q}^2\colon t_q^2\}, \mathbf{a}^{\downarrow}, \mathbf{a}^{\uparrow}) = (\bot, \top).$$

4. The seniority relation for **choices** is symmetric to the seniority relation for records, and, therefore, the iterated function for choices is defined similarly to records.

Example 1 A constraint for down-coerced terms, where the right part is a variable, is reduced by substitution to the constraint with the right part as a ground term:

$$\mathsf{IF}(\mathtt{int} \sqsubseteq \overline{v}_1, (\mathtt{int}), ()) = \mathsf{IF}(\mathtt{int} \sqsubseteq \mathtt{int}, (\mathtt{int}), ()) = ((\mathtt{int}), ()).$$

Example 2 Assume a constraint for records $\overline{v}_1 \sqsubseteq \{\mathsf{a}\colon \mathtt{int}\}$ and the approximation $a^{\downarrow} = (\{\mathsf{a}\colon \mathsf{nil}, \mathsf{b}\colon \mathtt{int}\})$ are given, where $v^{\downarrow} = (\overline{v}_1)$. The new approximation for $\overline{v}_1$ is computed as the greatest lower bound of the current approximation and the term $\{\mathsf{a}\colon \mathtt{int}\}$:

$$\begin{aligned}\mathsf{IF}(\overline{v}_1 \sqsubseteq \{\mathsf{a}\colon \mathtt{int}\}, (\{\mathsf{a}\colon \mathsf{nil}, \mathsf{b}\colon \mathtt{int}\}), ()) = \\ ((\{\mathsf{a}\colon \mathsf{nil}, \mathsf{b}\colon \mathtt{int}\} \sqcap \{\mathsf{a}\colon \mathtt{int}\}), ()) = \\ ((\{\mathsf{a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{int}\}), ()).\end{aligned}$$

Example 3 Consider the constraint for choices $(\mathsf{:a}\colon \mathtt{int}\mathsf{:}) \sqsubseteq \overline{\overline{v}}_1$ and the approximation $a^{\uparrow} = ((\mathsf{:a}\colon \mathsf{nil}, \mathsf{b}\colon \mathtt{int}\mathsf{:}))$ for a vector $v^{\uparrow} = (\overline{\overline{v}}_1)$ are given. The new approximation for $\overline{\overline{v}}_1$ is computed as the least upper bound of the current approximation and the grounded term:

$$\begin{aligned}\mathsf{IF}((\mathsf{:a}\colon \mathtt{int}\mathsf{:}) \sqsubseteq \overline{\overline{v}}_1, (), ((\mathsf{:a}\colon \mathsf{nil}, \mathsf{b}\colon \mathtt{int}\mathsf{:}))) = \\ (((\mathsf{:a}\colon \mathsf{nil}, \mathsf{b}\colon \mathtt{int}\mathsf{:}) \sqcup (\mathsf{:a}\colon \mathtt{int}\mathsf{:})), ()) = \\ (((\mathsf{:a}\colon \mathtt{int}, \mathsf{b}\colon \mathtt{int}\mathsf{:})), ()).\end{aligned}$$

6.2.1 Fixed-point Algorithm

Algorithm 1 below takes as an input a set of constraints $\mathscr{C}$ without b-variables (that is, for the case when $\mathsf{V}^b(\mathscr{C}) = \emptyset$) and returns a solution, provided that one exists; otherwise it returns $(\underline{\bot}, \overline{\top})$. It uses the function IF to compute a series of approximations. For simplicity we define the function $\mathsf{IF}_{\mathscr{C}}$ as a composition of IF functions that are sequentially applied to all constraints in $\mathscr{C}$. The order in which IF is applied to the constraints is not important due to the distributivity of the seniority relation. Note that the sequential composition preserves monotonicity for $\mathsf{IF}_{\mathscr{C}}$.

Algorithm 1 CSP-MDL($\mathscr{C}$), where $\mathsf{V}^b(\mathscr{C}) = \emptyset$

1: $i \leftarrow 0$
2: $(\mathbf{a}_0^{\downarrow}, \mathbf{a}_0^{\uparrow}) \leftarrow ((\mathsf{nil}, \ldots, \mathsf{nil}), (\mathsf{none}, \ldots, \mathsf{none}))$
3: **repeat**
4: $\quad i \leftarrow i+1$
5: $\quad (\mathbf{a}_i^{\downarrow}, \mathbf{a}_i^{\uparrow}) \leftarrow \mathsf{IF}_{\mathscr{C}}(\mathbf{a}_{i-1}^{\downarrow}, \mathbf{a}_{i-1}^{\uparrow})$
6: **until** $(\mathbf{a}_i^{\downarrow}, \mathbf{a}_i^{\uparrow}) = (\mathbf{a}_{i-1}^{\downarrow}, \mathbf{a}_{i-1}^{\uparrow})$
7: **if** $(\mathbf{a}_i^{\downarrow}, \mathbf{a}_i^{\uparrow}) = (\underline{\bot}, \overline{\top})$ **then**
8: $\quad$ **return** Unsat
9: **else**
10: $\quad$ **return** $(\mathbf{a}_i^{\downarrow}, \mathbf{a}_i^{\uparrow})$
11: **end if**

We observe that the function IF is monotonic. Lemma 1 below formally states that substitution of down-coerced variables is a decreasing function (indeed, the approximations for down-coerced variables can only coerce down in the lattice $\widetilde{\mathscr{T}_m^{\downarrow}}$). Similarly, we can prove that substitution of up-coerced variables is an increasing function.

Lemma 1 (Substitution monotonicity) *Let t be a term such that $|\mathsf{V}^b(t)| = \emptyset$, $\mathbf{v}^{\downarrow} = (v_1, \ldots, v_k)$ be a vector of down-coerced variables in t, and $\mathbf{s_1}^{\downarrow} = (s_1^1, \ldots, s_k^1)$ and $\mathbf{s_2}^{\downarrow} = (s_1^2, \ldots, s_k^2)$ be vectors of down-coerced ground terms such that $\mathbf{s_1}^{\downarrow} \sqsubseteq \mathbf{s_2}^{\downarrow}$. Then*

$$t[\mathbf{v}^{\downarrow}/\mathbf{s_1}^{\downarrow}] \sqsubseteq t[\mathbf{v}^{\downarrow}/\mathbf{s_2}^{\downarrow}].$$

Proof Substitution monotonicity follows from the structure of the seniority relation. Any term is covariant with respect to its subterms (see Definition 3). □

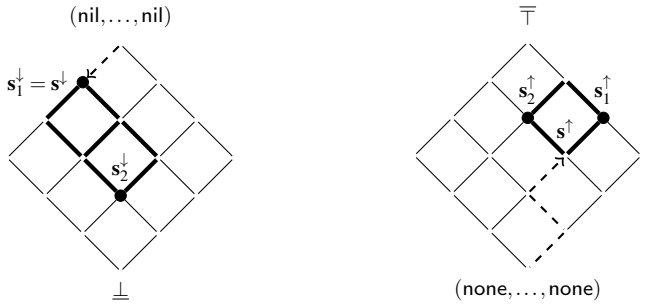
Lemma 2 states that the function IF produces at each step the tightest possible approximation.

Lemma 2 *Assume a constraint $t_1 \sqsubseteq t_2$ such that $\mathsf{IF}(t_1 \sqsubseteq t_2, \mathbf{a}_1^{\downarrow}, \mathbf{a}_1^{\uparrow}) = (\mathbf{a}_2^{\downarrow}, \mathbf{a}_2^{\uparrow})$ for some $(\mathbf{a}_1^{\downarrow}, \mathbf{a}_1^{\uparrow})$ and $(\mathbf{a}_2^{\downarrow}, \mathbf{a}_2^{\uparrow})$. If*

$$t_1[\mathbf{v}^{\downarrow}/\mathbf{a}_2^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_1^{\uparrow}] \sqsubseteq t_2[\mathbf{v}^{\downarrow}/\mathbf{a}_1^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_2^{\uparrow}],$$

then no approximation $(\mathbf{a}_3^{\downarrow}, \mathbf{a}_3^{\uparrow})$ exists such that $(\mathbf{a}_3^{\downarrow}, \mathbf{a}_3^{\uparrow}) \neq (\mathbf{a}_2^{\downarrow}, \mathbf{a}_2^{\uparrow})$, $\mathbf{a}_2^{\downarrow} \sqsubseteq \mathbf{a}_3^{\downarrow}$, $\mathbf{a}_3^{\uparrow} \sqsubseteq \mathbf{a}_2^{\uparrow}$ and

$$t_1[\mathbf{v}^{\downarrow}/\mathbf{a}_3^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_1^{\uparrow}] \sqsubseteq t_2[\mathbf{v}^{\downarrow}/\mathbf{a}_1^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_3^{\uparrow}]. \quad (1)$$



(a) the lattice $(\widetilde{\mathscr{T}_m^{\downarrow}}, \sqsubseteq)$ (b) the lattice $(\widetilde{\mathscr{T}_n^{\uparrow}}, \sqsubseteq)$

Fig. 3: Lemma 3 states that the chain of approximations (the dashed path) converges to the greatest fixed point $\mathbf{s}^{\downarrow}$ in $(\widetilde{\mathscr{T}_m^{\downarrow}}, \sqsubseteq)$ and the least fixed point $\mathbf{s}^{\uparrow}$ in $(\widetilde{\mathscr{T}_n^{\uparrow}}, \sqsubseteq)$.

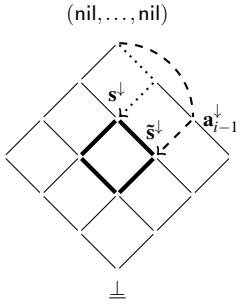


Fig. 4: Illustration of the reachability proof in Lemma 3.

Proof By definition, the function IF makes a coercion of the approximation $(\mathbf{a}_1^{\downarrow}, \mathbf{a}_1^{\uparrow})$ only if $(\mathbf{a}_1^{\downarrow}, \mathbf{a}_1^{\uparrow})$ is not a solution and the coercion is required for satisfaction of $t_1 \sqsubseteq t_2$. As a result, the function produces a coerced $(\mathbf{a}_2^{\downarrow}, \mathbf{a}_2^{\uparrow})$. The approximation $(\mathbf{a}_3^{\downarrow}, \mathbf{a}_3^{\uparrow})$, such that (1) holds, would exist only if IF performed excessive coercions, which is avoided by definition. □

Lemma 3 *Assume a set of constraints* $\mathscr{C}$, $\mathsf{V}^b(\mathscr{C}) = \emptyset$. *Let for* $k > 0$

$$(\mathbf{a}_0^{\downarrow}, \mathbf{a}_0^{\uparrow}), \ldots, (\mathbf{a}_k^{\downarrow}, \mathbf{a}_k^{\uparrow})$$

be a series of approximations such that $(\mathbf{a}_i^{\downarrow}, \mathbf{a}_i^{\uparrow}) = \mathit{IF}_{\mathscr{C}}(\mathbf{a}_{i-1}^{\downarrow}, \mathbf{a}_{i-1}^{\uparrow})$ *for any* $0 < i \leq k$, *and* $\mathbf{a}_0^{\downarrow} = (\mathsf{nil}, \ldots, \mathsf{nil})$ *and* $\mathbf{a}_0^{\uparrow} = (\mathsf{none}, \ldots, \mathsf{none})$. *Then for any fixed-point* $(\tilde{\mathbf{s}}^{\downarrow}, \tilde{\mathbf{s}}^{\uparrow})$

$$\tilde{\mathbf{s}}^{\downarrow} \sqsubseteq \mathbf{a}_k^{\downarrow} \text{ and } \mathbf{a}_k^{\uparrow} \sqsubseteq \tilde{\mathbf{s}}^{\uparrow}.$$

Proof The proof consists of two parts. First, we prove that a fixed-point $(\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow})$ exists, such that for any fixed point $(\tilde{\mathbf{s}}^{\downarrow}, \tilde{\mathbf{s}}^{\uparrow})$, $\tilde{\mathbf{s}}^{\downarrow} \sqsubseteq \mathbf{s}^{\downarrow}$ and $\mathbf{s}^{\uparrow} \sqsubseteq \tilde{\mathbf{s}}^{\uparrow}$ (existence). Then we show that $(\mathbf{a}_k^{\downarrow}, \mathbf{a}_k^{\uparrow}) = (\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow})$ (reachability).

Existence. By Knaster-Tarski theorem [45], the sets of fixed points of IF in $(\widetilde{\mathscr{T}_m^{\downarrow}}, \sqsubseteq)$ and $(\widetilde{\mathscr{T}_n^{\uparrow}}, \sqsubseteq)$ are lattices. Therefore, there exists the fixed-point $(\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow})$ such that for any fixed-point $(\tilde{\mathbf{s}}^{\downarrow}, \tilde{\mathbf{s}}^{\uparrow})$, $\tilde{\mathbf{s}}^{\downarrow} \sqsubseteq \mathbf{s}^{\downarrow}$ and $\mathbf{s}^{\uparrow} \sqsubseteq \tilde{\mathbf{s}}^{\uparrow}$ (see Fig. 3).

Reachability. Proof by contradiction (see Fig. 4). Assume that IF does not converge to $(\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow})$, that is $(\mathbf{a}_k^{\downarrow}, \mathbf{a}_k^{\uparrow}) = (\tilde{\mathbf{s}}^{\downarrow}, \tilde{\mathbf{s}}^{\uparrow})$, where $(\tilde{\mathbf{s}}^{\downarrow}, \tilde{\mathbf{s}}^{\uparrow}) \neq (\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow})$, and $\tilde{\mathbf{s}}^{\downarrow} \sqsubseteq \mathbf{s}^{\downarrow}$ or $\mathbf{s}^{\uparrow} \sqsubseteq \tilde{\mathbf{s}}^{\uparrow}$. Assume that $\tilde{\mathbf{s}}^{\downarrow} \sqsubseteq \mathbf{s}^{\downarrow}$ (the case $\mathbf{s}^{\uparrow} \sqsubseteq \tilde{\mathbf{s}}^{\uparrow}$ is similar).

Let $(\mathbf{a}_{i-1}^{\downarrow}, \mathbf{a}_{i-1}^{\uparrow})$ be the approximation that precedes $(\tilde{\mathbf{s}}^{\downarrow}, \tilde{\mathbf{s}}^{\uparrow})$ in the chain of approximations: $\mathsf{IF}(\mathbf{a}_{i-1}^{\downarrow}, \mathbf{a}_{i-1}^{\uparrow}) = (\tilde{\mathbf{s}}^{\downarrow}, \tilde{\mathbf{s}}^{\uparrow})$. Then for every constraint $t_1 \sqsubseteq t_2 \in \mathscr{C}$

$$t_1[\mathbf{v}^{\downarrow}/\tilde{\mathbf{s}}^{\downarrow}, \mathbf{v}^{\uparrow}/\tilde{\mathbf{a}}_{i-1}^{\uparrow}] \sqsubseteq t_2[\mathbf{v}^{\downarrow}/\mathbf{a}_{i-1}^{\downarrow}, \mathbf{v}^{\uparrow}/\tilde{\mathbf{s}}^{\uparrow}].$$

Since $\mathbf{s}^{\downarrow}$ is a fixed point, then

$$t_1[\mathbf{v}^{\downarrow}/\mathbf{s}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_{i-1}^{\uparrow}] \sqsubseteq t_2[\mathbf{v}^{\downarrow}/\mathbf{a}_{i-1}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{s}^{\uparrow}].$$

On the other hand, $\tilde{\mathbf{s}}^{\downarrow} \sqsubseteq \mathbf{s}^{\downarrow}$. Due to the substitution monotonicity (Lemma 1),

$$t_1[\mathbf{v}^{\downarrow}/\tilde{\mathbf{s}}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_{i-1}^{\uparrow}] \sqsubseteq t_1[\mathbf{v}^{\downarrow}/\mathbf{s}^{\downarrow}, \mathbf{v}^{\uparrow}/\mathbf{a}_{i-1}^{\uparrow}]. \tag{2}$$

Lemma 2 states that IF produces the "tightest" approximation of $\tilde{\mathbf{s}}^{\downarrow}$. On the other hand, it follows from (2) that $\mathbf{s}^{\downarrow}$ is "tighter" than $\tilde{\mathbf{s}}^{\downarrow}$. It leads to a contradiction.

□

Termination of the algorithm is based on the monotonicity of the iterated function, and the fact that each term category may expand (that is approximations of the down-coerced terms "go up" and approximations of the up-coerced terms "go down") only a finite number of times.

Theorem 1 (Termination) *For any set of constraints $\mathscr{C}$ with the empty set of b-variables ($\mathsf{V}^b(\mathscr{C}) = \emptyset$), Algorithm 1 terminates after a finite number of steps.*

Proof The algorithm computes the iterated function until the fixed-point is reached. It follows from Lemma 1 that IF is a monotonic function. Therefore, the algorithm terminates after a finite number of steps if both lattices have a finite height. We prove it by induction on the depth of a term.

We observe that by definition $(\widetilde{\mathscr{T}_m^{\downarrow}}, \sqsubseteq)$ and $(\widetilde{\mathscr{T}_n^{\uparrow}}, \sqsubseteq)$ have a finite height if the semilattices $(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$ and $(\mathscr{T}_n^{\uparrow}, \sqsubseteq)$ have a finite height. That is, it is sufficient to show the later.

We consider the semilattice $(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$ (the proof for $(\mathscr{T}_n^{\uparrow}, \sqsubseteq)$ is similar) for each of term categories (that is when an element of the semilattice is either a symbol, a tuple or a record on the top-level). We rely on the property that follows from the seniority relation (see Definition 3 and Fig. 2): the term category is constant and cannot be changed unless the term is nil.

Symbol. The semilattice $(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$ for a symbol consists of two elements (nil and the symbol itself). The symbol does not contain nested terms and, therefore, the semilattice for the symbol has a finite height.

Tuple. The "width" (the number of elements) of a tuple is constant. Therefore, the tuple cannot *expand* (by expansion we mean adding new elements to the tuple). The height of the semilattice for the tuple is finite providing that the semilattices for its nested terms is finite.

Record. For any given $\mathscr{C}$ the size of a record can only expand by adding elements with labels that are not yet present in the record. The set of labels in $\mathscr{C}$ is finite and the algorithm does not generate new labels. Therefore, the record can expand only a finite number of times. The height of the semilattice for the record is finite providing that the semilattices for its nested terms are finite.

Choice. The case for a choice term is considered similarly to the one for a record.

As a result, the semilattices $(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$ and $(\mathscr{T}_n^{\uparrow}, \sqsubseteq)$ have a finite height. Therefore, the lattices $(\widetilde{\mathscr{T}}_m^{\downarrow}, \sqsubseteq)$ and $(\widetilde{\mathscr{T}}_n^{\uparrow}, \sqsubseteq)$ have a finite height too. □

Theorem 2 (Correctness of Algorithm 1) *For any set of constraints $\mathscr{C}$ with the empty set of b-variables (that is, $\mathsf{V}^b(\mathscr{C}) = \emptyset$),* CSP-MDL *for $\mathscr{C}$ is unsatisfiable if and only if Algorithm 1 returns* Unsat.

Proof We give a proof by contradiction.

($\Rightarrow$) Let $\mathscr{C}$ be an unsatisfiable set of constraints and Algorithm 1 returns $(\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow})$ such that $(\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow}) \neq (\underline{\bot}, \overline{\top})$. Then by Lemma 3, $(\mathbf{s}^{\downarrow}, \mathbf{s}^{\uparrow})$ is the fixed point that contains values satisfying $\mathscr{C}$. This contradicts the initial hypothesis. Therefore, Algorithm 1 returns Unsat if $\mathscr{C}$ is unsatisfiable.

($\Leftarrow$) Let Algorithm 1 return Unsat and $\mathscr{C}$ has a solution. In this case the algorithm computes the chain of approximations. This is the fixed point, and by Lemma 3 no other fixed point $(\bar{\mathbf{s}}^{\downarrow}, \bar{\mathbf{s}}^{\uparrow})$ exists such that $\underline{\bot} \sqsubseteq \bar{\mathbf{s}}^{\downarrow}$ or $\bar{\mathbf{s}}^{\uparrow} \sqsubseteq \overline{\top}$. It means that no fixed points apart from $(\underline{\bot}, \overline{\top})$ exists. This contradicts the initial hypothesis. Therefore, $\mathscr{C}$ is unsatisfiable if Algorithm 1 returns Unsat. □

Note that our approach computes the tightest possible solution (as defined by the conditions of Lemma 2), provided that the set of constraints has at least one solution.

Example 4 The set of constraints $\{x \sqsubseteq \{a:1\}, x \sqsubseteq \{b:1\}\}$ has multiple solutions. For example, $x_1 = \{a:1, b:1, c:1\}$ is a solution, and $x_2 = \{a:1, b:1, d:1\}$ is also a solution. And the tightest solution is $x_{tight} = \{a:1, b:1\}$.

6.3 CSP-MDL Algorithm

In this section we extend the algorithm to support constraints that contain b-variables. By instantiating b-variables in all possible ways, in the worst case we obtain 2^l subproblems of the initial problem, where l is the number of b-variables.

A straightforward approach would be to call Algorithm 1 for each of the 2^l subproblems independently. In order to improve efficiency, we propose a heuristic implemented on the top of Algorithm 1. We construct a set of Boolean constraints, which specifies Boolean instantiations potentially leading to finding a solution by ensuring well-formedness of terms, and satisfaction of the seniority relation.

1. $\mathsf{WFC}(t) = \emptyset$ if t is a symbol;
2. $\mathsf{WFC}(t) = \bigcup_{1 \le i \le n} \mathsf{WFC}(t_i)$ if t is a tuple $(t_1 \ldots t_n)$;
3. $\mathsf{WFC}(t) = \{\neg(g_i \wedge g_j) \mid 1 \le i \ne j \le n \text{ and } l_i = l_j\} \cup \bigcup_{1 \le i \le n} \{g_i \to g \mid g \in \mathsf{WFC}(t_i)\}$ if t is a record $\{\mathsf{l}_1(g_1)\colon t_1, \ldots, \mathsf{l}_n(g_n)\colon t_n\}$ or a choice $(:\mathsf{l}_1(g_1)\colon t_1, \ldots, \mathsf{l}_n(g_n)\colon t_n:)$;
4. $\mathsf{WFC}(t) = \{\neg(g_i \wedge g_j) \mid 1 \le i \ne j \le n\} \cup \{\bigvee_{1 \le i \le n} g_i\} \cup \bigcup_{1 \le i \le n} \{g_i \to g \mid g \in \mathsf{WFC}(t_i)\}$ if t is a switch $\langle(g_1)\colon t_1, \ldots, (g_n)\colon t_n\rangle$.

Fig. 5: The set of Boolean constraints that ensures well-formedness of a term t

We use B to denote this set. If the set is a contradiction, then all instantiations are unsatisfiable. The algorithm returns Unsat accordingly. By $\mathsf{SAT}(\mathsf{B})$ we mean the set of Boolean vectors satisfying B.

6.3.1 Well-formedness Constraints

The first set of Boolean constraints is called *well-formedness constraints*. They follow directly from the definition of a well-formed term.

A symbol is a well-formed term and, therefore, no Boolean constraint is required. A tuple is a well-formed term if its nested terms are well-formed terms, too. Therefore, well-formedness constraints for a tuple is a union of well-formedness constraints for a tuple's nested terms.

If a term t is a record or a choice, then all labels of the collection must be distinct. Therefore, for any elements i and j with the same label the constraint $\neg(g_i \wedge g_j)$, where g_i and g_j are guards of element i and j, respectively, should be produced. In addition, well-formedness constraints for nested terms have to be taken into account, which is guaranteed by a constraint $g_i \to g$, where g_i is a guard for the element i and $g \in \mathsf{WFC}(t_i)$ is a well-formedness constraint for a nested term t_i.

If t is a switch, the well-formedness constraints ensure that only one element of a switch has a guard instantiated to true. Furthermore, $g_i \to g$, where g_i is a guard for the element i, ensure the constraints $g \in \mathsf{WFC}(t_i)$, which are well-formedness constraints for a nested term t_i.

6.3.2 Seniority Constraints

The other set of Boolean constraints is called *seniority constraints*. They naturally follow from the definition of the seniority relation.

Seniority constraints are denoted as $\mathsf{SC}(t_1 \sqsubseteq t_2)$ for a constraint $t_1 \sqsubseteq t_2$. They specify a set of constraints that ensures the seniority relation $t_1 \sqsubseteq t_2$, where t_1 and t_2 are well-formed terms. In other words, $\mathsf{SAT}(\mathsf{WFC}(t_1) \cup \mathsf{WFC}(t_2) \cup \mathsf{SC}(t_1 \sqsubseteq t_2)) \ne \emptyset$ guarantees that the seniority relation $t_1 \sqsubseteq t_2$ holds.

If t_1 and t_2 are equal symbols, then the seniority relation holds and no further Boolean constraints are required. If t_1 and t_2 are tuples of the same size, then the Boolean seniority constraints have to include Boolean seniority constraints for nested terms.

1. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \emptyset$, if t_1 and t_2 are equal symbols.
2. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \bigcup_{1 \le i \le d} \mathsf{SC}(t_i^1 \sqsubseteq t_i^2)$, if t_1 is a tuple $(t_1^1 \dots t_d^1)$ and t_2 is a tuple $(t_1^2 \dots t_d^2)$;
3. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \bigcup_{1 \le j \le d} \mathsf{SC}_j(t_j^2)$, if t_1 is a record $\{\mathsf{l}_1^1(g_1^1)\colon t_1^1, \dots, \mathsf{l}_\mathsf{d}^1(g_d^1)\colon t_d^1\}$, t_2 is a record $\{\mathsf{l}_2^1(g_1^2)\colon t_1^2, \dots, \mathsf{l}_\mathsf{e}^2(g_e^2)\colon t_e^2\}$ and $\mathsf{SC}_j(t_j^2)$ is one of the following:
   (a) $\mathsf{SC}_j(t_j^2) = \{(g_i^1 \wedge g_j^2) \to g \mid g \in \mathsf{SC}(t_i^1 \sqsubseteq t_j^2)\}$, if $\exists i : 1 \le i \le d$ and $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$;
   (b) $\mathsf{SC}_j(t_j^2) = \{\neg g_j^2\}$, otherwise;
4. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \bigcup_{1 \le i \le e} \mathsf{SC}_i(t_i^1)$, if t_1 is a choice $(:\mathsf{l}_1^1(g_1^1)\colon t_1^1, \dots, \mathsf{l}_\mathsf{d}^1(g_d^1)\colon t_d^1:)$, t_2 is a choice $(:\mathsf{l}_1^2(g_1^2)\colon t_1^2, \dots, \mathsf{l}_\mathsf{e}^2(g_e^2)\colon t_e^2:)$ and $\mathsf{SC}_i(t_i^1)$ is one of the following:
   (a) $\mathsf{SC}_i(t_i^1) = \{(g_i^1 \wedge g_j^2) \to g \mid g \in \mathsf{SC}(t_i^1 \sqsubseteq t_j^2)\}$, if $\exists j : 1 \le i \le e$ and $\mathsf{l}_\mathsf{i}^1 = \mathsf{l}_\mathsf{j}^2$;
   (b) $\mathsf{SC}_i(t_i^1) = \{\neg g_i^1\}$, otherwise;
5. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \{g_i^1 \to g \mid 1 \le i \le d$ and $g \in \mathsf{SC}(t_i^1 \sqsubseteq t_i^2)\}$, if t_1 is a switch $\langle (g_1^1)\colon t_1^1, \dots (g_d^1)\colon t_d^1 \rangle$ and t_2 is an arbitrary term.
6. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \{g_i^2 \to g \mid 1 \le i \le d$ and $g \in \mathsf{SC}(t_1 \sqsubseteq t_i^2)\}$, if t_1 is an arbitrary term and t_2 is a switch $\langle (g_1^2)\colon t_1^2, \dots (g_d^2)\colon t_d^2 \rangle$.
7. $\mathsf{SC}(t_1 \sqsubseteq t_2) = \{\mathsf{false}\}$, otherwise.

Fig. 6: The set of Boolean constraints that ensures the seniority relation $t_1 \sqsubseteq t_2$

If t_1 and t_2 are records (constraints for choices are generated in a similar way), then for every element $\mathsf{l}_\mathsf{j}^2(g_j^2)\colon t_j^2$ in t_2 the following set of Boolean constraints is produced:

1. if there exists an element $\mathsf{l}_\mathsf{i}^1(g_i^1)\colon t_i^1$ in t_1 such that l_i^1 is the same as l_j^2, then the Boolean seniority constraints have to include the seniority constraints for nested subterms providing that $g_i^1 \wedge g_j^2$;
2. otherwise, an element $\mathsf{l}_\mathsf{j}^2(g_j^2)\colon t_j^2$ has to be excluded from the collection: the constraint $\neg g_j^2$ is generated.

The Boolean seniority constraints for switches (t_1 or t_2 is a switch) ensure only the Boolean seniority constraints for nested terms. Finally, if $t_1 \sqsubseteq t_2$ does not match any of the above cases, then the constraint is unsatisfiable and false is generated.

6.3.3 Algorithm

Let $\mathsf{B}_0 \subseteq \mathsf{B}_1 \subseteq \dots \subseteq \mathsf{B}_s$ be sets of Boolean constraints, and $\mathbf{a}^\downarrow$ and $\mathbf{a}^\uparrow$ be vectors of semiground terms such that $|\mathbf{a}^\downarrow| = |\mathsf{V}^\downarrow(\mathscr{C})|$ and $|\mathbf{a}^\uparrow| = |\mathsf{V}^\uparrow(\mathscr{C})|$. Algorithm 2 seeks a solution of CSP-MDL($\mathscr{C}$) as the fixed point of a chain of approximations in the form

$$(\mathsf{B}_0, \mathbf{a}_0^\downarrow, \mathbf{a}_0^\uparrow), \dots, (\mathsf{B}_{s-1}, \mathbf{a}_{s-1}^\downarrow, \mathbf{a}_{s-1}^\uparrow), (\mathsf{B}_s, \mathbf{a}_s^\downarrow, \mathbf{a}_s^\uparrow),$$

where for every i, $1 \le i \le s$, and a vector of Boolean values $\mathbf{b} \in \mathsf{SAT}(\mathsf{B}_i)$,

$$\mathbf{a}_i^\downarrow[\mathbf{f}/\mathbf{b}] \sqsubseteq \mathbf{a}_{i-1}^\downarrow[\mathbf{f}/\mathbf{b}] \qquad \text{and} \qquad \mathbf{a}_{i-1}^\uparrow[\mathbf{f}/\mathbf{b}] \sqsubseteq \mathbf{a}_i^\uparrow[\mathbf{f}/\mathbf{b}].$$

The starting initialisation is $\mathsf{B}_0 = \emptyset$, $\mathbf{a}_0^\downarrow = (\mathsf{nil}, \dots, \mathsf{nil})$, $\mathbf{a}_0^\uparrow = (\mathsf{none}, \dots, \mathsf{none})$ and the chain of approximations terminates as soon as it finds $\mathbf{b} \in \mathsf{SAT}(\mathsf{B}_i)$ such that

$$(\mathbf{a}_i^\downarrow[\mathbf{f}/\mathbf{b}], \mathbf{a}_i^\uparrow[\mathbf{f}/\mathbf{b}]) = (\mathbf{a}_{i-1}^\downarrow[\mathbf{f}/\mathbf{b}], \mathbf{a}_{i-1}^\uparrow[\mathbf{f}/\mathbf{b}]).$$

Algorithm 2 CSP-MDL($\mathscr{C}$)

$c \leftarrow |\mathscr{C}|$
$i \leftarrow 0$
$\mathsf{B}_0 \leftarrow \emptyset$
$\mathbf{a}_0^{\downarrow} \leftarrow (\mathsf{nil}, \ldots, \mathsf{nil})$
$\mathbf{a}_0^{\uparrow} \leftarrow (\mathsf{none}, \ldots, \mathsf{none})$
repeat
  $i \leftarrow i+1$
  $(\mathbf{a}_i^{\downarrow}, \mathbf{a}_i^{\uparrow}) \leftarrow \mathsf{IF}_{\mathscr{C}}(\mathbf{a}_{i-1}^{\downarrow}, \mathbf{a}_{i-1}^{\uparrow})$
  $\mathsf{B}_i \leftarrow \mathsf{B}_{i-1} \cup \bigcup_{t_1 \sqsubseteq t_2 \in \mathscr{C}} (\mathsf{WFC}(t_1[\mathbf{v}/\mathbf{a}_i]) \cup \mathsf{WFC}(t_2[\mathbf{v}/\mathbf{a}_i]) \cup \mathsf{SC}(t_1[\mathbf{v}/\mathbf{a}_i] \sqsubseteq t_2[\mathbf{v}/\mathbf{a}_i]))$
until $\exists \mathbf{b} \in \mathsf{SAT}(\mathsf{B}_i) : (\mathbf{a}_i^{\downarrow}[\mathbf{f}/\mathbf{b}], \mathbf{a}_i^{\uparrow}[\mathbf{f}/\mathbf{b}]) = (\mathbf{a}_{i-1}^{\downarrow}[\mathbf{f}/\mathbf{b}], \mathbf{a}_{i-1}^{\uparrow}[\mathbf{f}/\mathbf{b}])$
if B_i is unsatisfiable **then**
  return Unsat
else
  return $(\mathbf{b}, \mathbf{a}_i^{\downarrow}[\mathbf{f}/\mathbf{b}], \mathbf{a}_i^{\uparrow}[\mathbf{f}/\mathbf{b}])$, where $\mathbf{b} \in \mathsf{SAT}(\mathsf{B}_i)$
end if

The adjunct set of Boolean constraints potentially expands at every iteration of the algorithm by inclusion of further logic formulas produced by the set of Boolean constraints that ensure well-formedness of the terms and satisfaction of the seniority relation. Moreover, if the original CSP-MDL is satisfiable, then so is $\mathsf{SAT}(\mathsf{B}_s)$ by construction. That is if the tuple of vectors $(\mathbf{b}_s, \mathbf{a}_s^{\downarrow}[\mathbf{f}/\mathbf{b_s}], \mathbf{a}_s^{\uparrow}[\mathbf{f}/\mathbf{b_s}])$ is a solution to the former, then $\mathbf{b_s}$ is a solution to $\mathsf{SAT}(\mathsf{B}_s)$. Note that Algorithm 2 runs Algorithm 1 as a subroutine and adds adjunct Boolean constraints that ensure well-formedness of terms and satisfaction of the seniority relation. That is, the main aim of constructing the set B is to limit the search space.

Theorem 3 (Correctness of Algorithm 2) *For any set of constraints $\mathscr{C}$,* CSP-MDL *for $\mathscr{C}$ is unsatisfiable if and only if Algorithm 2 returns* Unsat.

Proof The sets of Boolean constraints provided in Fig. 5 and Fig. 6 guarantee that the terms are wellformed and the seniority relation is satisfied. Now we give a proof by contradiction.

($\Rightarrow$) Let $\mathscr{C}$ be an unsatisfiable set of constraints and Algorithm 2 return

$$(\mathbf{b}, \mathbf{a}_i^{\downarrow}[\mathbf{f}/\mathbf{b}], \mathbf{a}_i^{\uparrow}[\mathbf{f}/\mathbf{b}]),$$

where $\mathbf{b} \in \mathsf{SAT}(\mathsf{B}_i)$. Then $(\mathbf{a}_i^{\downarrow}[\mathbf{f}/\mathbf{b}], \mathbf{a}_i^{\uparrow}[\mathbf{f}/\mathbf{b}]) \neq (\underline{\bot}, \overline{\top})$, where $(\mathbf{a}_i^{\downarrow}[\mathbf{f}/\mathbf{b}], \mathbf{a}_i^{\uparrow}[\mathbf{f}/\mathbf{b}])$ is the fixed point that contains values satisfying $\mathscr{C}$. This contradicts the initial hypothesis. Therefore, Algorithm 2 returns Unsat if $\mathscr{C}$ is unsatisfiable.

($\Leftarrow$) Let Algorithm 2 return Unsat and $\mathscr{C}$ have a solution. In this case the algorithm computes a chain of approximations. This is a fixed point and by Lemma 3 no other fixed point $(\mathbf{b}, \mathbf{a}_i^{\downarrow}[\mathbf{f}/\mathbf{b}], \mathbf{a}_i^{\uparrow}[\mathbf{f}/\mathbf{b}])$, where $\mathbf{b} \in \mathsf{SAT}(\mathsf{B}_i)$, exists such that $\underline{\bot} \sqsubseteq \mathbf{a}_i^{\downarrow}[\mathbf{f}/\mathbf{b}]$ or $\mathbf{a}_i^{\uparrow}[\mathbf{f}/\mathbf{b}] \sqsubseteq \overline{\top}$, which means that no fixed points apart from $(\underline{\bot}, \overline{\top})$ exists. This contradicts the initial hypothesis. Therefore, $\mathscr{C}$ is unsatisfiable if Algorithm 2 returns Unsat. □

In general, the set $\mathsf{SAT}(\mathsf{B}_s)$ can have more than one solution, and we select one of them. Heuristics that would make it possible to select a better solution for a given application require further research.

6.4 Complexity

Deciding whether or not a given CSP instance has a solution is in general NP- complete [33]. Therefore, one of the most fundamental challenges in constraint solving is to understand the computational complexity of problems involving constraints, and, in particular, to characterise exactly which types of constraints give rise to constraint problems which can be solved in polynomial time [8,9,15].

A CSP can be easier to solve than in the general case when the types of constraints are limited. So far the most successful method of studying the complexity of CSPs on finite domains has been the algebraic approach introduced by Jeavons, Cohen and Gyssens in [26]. It shows that if the constraints are preserved by a semilattice operation then the CSP can be solved in polynomial time. This result has been further extended by Bodinsky, Macpherson and Thapper to certain classes of CSPs for infinite domains [6].

Jeavons et al. define in [26] an ACI operation as an idempotent operation, which is also associative and commutative. Consequently they use well-known results from elementary algebra[5] to show that any set of relations over a finite domain D is solvable in polynomial time if this set is closed under some ACI operation.

Definition 5 (ACI operation) Let $\mathsf{Op} : D^2 \to D$ be an idempotent binary operation on the set D such that for all $d_1, d_2, d_3 \in D$,

- (Associativity) $\mathsf{Op}(\mathsf{Op}(d_1,d_2),d_3) = \mathsf{Op}(d_1,\mathsf{Op}(d_2,d_3))$; and
- (Commutativity) $\mathsf{Op}(d_1,d_2) = \mathsf{Op}(d_2,d1_)$.

Then Op is said to be an ACI operation.

Theorem 4 (Jeavons et al., 1997) *For any set of relations Γ over a finite domain D, if Γ is closed under some ACI operation, then CSP(Γ) is solvable in polynomial time.*

Clearly, the meet $\sqcap$ and join $\sqcup$ are dual to one another with respect to order inversion. Moreover, any (finite) nonempty set which is $\sqcap$-closed (alternatively $\sqcup$-closed) contains the least upper bound (alternatively the greatest lower bound) with respect to the respective partial order [26].

Lemma 4 *For any set of constraints $\mathscr{C}$ with the empty set of b-variables ($\mathsf{V}^b(\mathscr{C}) = \emptyset$), Algorithm 1 requires time polynomial in $|\mathscr{C}|$.*

Proof By Theorem 4 We note that we extended the pair $(\mathscr{T}_m^{\downarrow}, \sqsubseteq)$ and $(\mathscr{T}_n^{\uparrow}, \sqsubseteq)$ of the meet and join semilattices, to the lattices $(\widetilde{\mathscr{T}_m^{\downarrow}}, \sqsubseteq)$ and $(\widetilde{\mathscr{T}_n^{\uparrow}}, \sqsubseteq)$ by introducing the top and bottom elements $\bot$ and $\top$. As both lattices $(\widetilde{\mathscr{T}_m^{\downarrow}}, \sqsubseteq)$ and $(\widetilde{\mathscr{T}_n^{\uparrow}}, \sqsubseteq)$ are now trivially $\sqcap$- and $\sqcup$-closed, the lemma holds by Theorem 4. □

[5] See, for example, [34].

In general, an instance of the CSP-MDL problem can contain arbitrary Boolean constraints on b-variables. That is the problem is NP-complete as stated in Section 5, and, therefore, Algorithm 2 requires exponential time in the worst case.

7 Service Network Topology

Definition of service composition using the "algebraic" style facilitates formal reasoning about the services [19,40]. Following this approach, we propose a formal description of a service network in the form of a language of combinators. Then, we define a type system on top of the language that aggregates communication constraints throughout the network.

We regard a service-based application as a network N, and $\mathscr{C}(N)$ (or just $\mathscr{C}$ for simplicity) is the set of constraints that have to be satisfied by N. Given $\mathscr{C}(N)$, we use $\mathsf{V}^b(\mathscr{C})$ to denote the set of b-variables, $\mathsf{V}^{\downarrow}(\mathscr{C})$ to denote the vector of down-coerced t-variables and $\mathsf{V}^{\uparrow}(\mathscr{C})$ to denote the vector of up-coerced t-variables.

We associate with each service s the pair $\tau(s) = (\mathscr{I}_s, \mathscr{O}_s)$, where $\mathscr{I}_s$ and $\mathscr{O}_s$ are the sets of its input and output ports respectively. That is $\tau(s)$ represents the service properties related to its interface. They can either be provided explicitly or be automatically derived from the service.

Each port is formally a pair (l_p, t_p), where l_p is the name of the port and $t_p \in \mathscr{T}$ is the MDL term specifying the service interface associated with the port.

7.1 Wiring

Wiring is the act of connecting services with communication channels. Each service is identified by the label, which denotes its functionality. An application is represented by a *streaming network* defined by the following grammar:

⟨*network*⟩ ::= ⟨*label*⟩
| ⟨*network*⟩ **..** ⟨*network*⟩
| ⟨*network*⟩ || ⟨*network*⟩
| ⟨*network*⟩ \
| **(** ⟨*network*⟩ **)**

with wiring patterns **..**, ||, and \, where

- **..** denotes a serial connection of two networks. Informally, it wires the output ports of one service to the input ports of another one with communication channels.
- || denotes the parallel connection of two networks. It places two networks side by side without introducing additional channels. That is the parallel connection builds a composite network by taking the union of the input and output ports.
- \ denotes the wrap-around connection for a network. It creates a cycling connection by wiring output ports to input ports in the network.

Furthermore, parens are introduced for grouping subnetworks together. For example, $(A \,..\, B) \,||\, \mathrm{C}$ and $\mathrm{A} \,..\, (\mathrm{B} \,||\, \mathrm{C})$ specify different networks due to different subnetwork grouping.

7.2 Types

With each network N we associate a *type* that encodes the ports and a set of the seniority constraints. The type of N is a tuple

$$(\mathscr{I}_N, \mathscr{O}_N, \mathscr{C}_N),$$

where $\mathscr{I}_N$ and $\mathscr{O}_N$ are sets of the input and output ports in the network respectively, and $\mathscr{C}_N$ is a set of the seniority constraints, which guarantee communication safety. Sets $\mathscr{I}_\mathtt{L}$ and $\mathscr{O}_\mathtt{L}$ can be specified in a separate service configuration file or can automatically be derived from the service. Each constraint is a relation

$$t_p \sqsubseteq t_{p'} : \mathscr{T} \times \mathscr{T}$$

on the wired ports p and p', where $\mathscr{T}$ is the set of ground terms. The seniority relation specifies conditions on the service interface wired with the communication channel.

We propose typing rules which specify a mechanism for aggregating the wiring constraints from the network.

The typing rule for a single service also referred as a *singleton network*, where L is a service label, is below:

$$(\textsc{sing})\ \frac{\mathtt{L}}{\mathtt{L}\colon (\mathscr{I}_\mathtt{L}, \mathscr{O}_\mathtt{L}, \emptyset)}$$

The type associated with the serial connection is constructed as follows:

$$(\,..\,)\ \frac{N_1 : (\mathscr{I}_{N_1}, \mathscr{O}_{N_1}, \mathscr{C}_{N_1}) \qquad N_2 : (\mathscr{I}_{N_2}, \mathscr{O}_{N_2}, \mathscr{C}_{N_2})}{N_1 \,..\, N_2 : (\mathscr{I}', \mathscr{O}', \mathscr{C}')}$$

where

$$\begin{aligned}
\mathscr{I}' &= \mathscr{I}_{N_1} \cup \{(l_p, t_p) \in \mathscr{I}_{N_2} \mid \forall (l'_p, t'_p) \in \mathscr{O}_{N_1} : l_p \neq l'_p\},\\
\mathscr{O}' &= \mathscr{O}_{N_2} \cup \{(l_p, t_p) \in \mathscr{O}_{N_1} \mid \forall (l_{p'}, t_{p'}) \in \mathscr{I}_{N_2} : l_p \neq l_{p'}\},\\
\mathscr{C}' &= \mathscr{C}_{N_1} \cup \mathscr{C}_{N_2} \cup \{t_p \sqsubseteq t_{p'} \mid \exists (l_p, t_p) \in \mathscr{O}_{N_1}, (l_{p'}, t_{p'}) \in \mathscr{I}_{N_2} : l_p = l_{p'}\}.
\end{aligned}$$

That is, the serial connection wires the output ports of of N_1 with identically named input ports of N_2, and the set of constraints is the union of the constraints on the networks N_1 and N_2 and constraints that represent data relations on newly constructed channels.

We call $l_p = l_{p'}$ the *identity condition*: the channels always wire the identically named ports. Consequently, the wiring of the services in the network depends on service port names.

The wiring relation is completely generic, that is it can lead to one-to-one, one-to-many, many-to-one or many-to-many connections.The problem of excessive or deficient wiring can be prevented by renaming the ports. Note that a channel may wire a single output port to more than one input port and a single input port to more than one output port:

- The semantics of the former is *copying*: each message output on the port will be received by each of the input port that the output port is wired to.
- The semantics of the latter is *merging*: when more than one output port is wired to a single input port, the messages from the output port are transferred to a single input port in no particular order, that is nondeterministically. It is also possible for a port to merge several inputs and copy the stream to several outputs.

The type associated with the parallel connection is constructed as follows:

$$(\,||\,)\ \frac{N_1 : (\mathscr{I}_{N_1}, \mathscr{O}_{N_1}, \mathscr{C}_{N_1}) \qquad N_2 : (\mathscr{I}_{N_2}, \mathscr{O}_{N_2}, \mathscr{C}_{N_2})}{N_1 \,||\, N_2 : (\mathscr{I}_{N_1} \cup \mathscr{I}_{N_2}, \mathscr{O}_{N_1} \cup \mathscr{O}_{N_2}, \mathscr{C}_{N_1} \cup \mathscr{C}_{N_2})}$$

The parallel connection does not wire services by producing new channels. Instead, it "joins" services by combining their input/output ports as well as sets of the seniority constraints into a single set. Moreover, $\mathscr{I}_{N_1}$ and $\mathscr{I}_{N_2}$, as well as $\mathscr{O}_{N_1}$ and $\mathscr{O}_{N_2}$ may contain ports with the same name without having compatibility issues. Consider the following example.

Finally, consider the typing rule for the wrap-around connection. The type associated with the wrap-around connection is constructed as follows:

$$(\backslash)\ \frac{N : (\mathscr{I}_N, \mathscr{O}_N, \mathscr{C}_N)}{N\backslash : (\mathscr{I}'_N, \mathscr{O}'_N, \mathscr{C}_N \cup \mathscr{C}'_N)}$$

where

$$\begin{aligned}
\mathscr{I}'_N &= \{(l_p, t_p) \mid \exists (l_p, t_p) \in \mathscr{I}_N\ \forall (l_{p'}, t_{p'}) \in \mathscr{O}_N :\ l_p \neq l_{p'}\},\\
\mathscr{O}'_N &= \{(l_{p'}, t_{p'}) \mid \exists (l_{p'}, t_{p'}) \in \mathscr{O}_N\ \forall (l_p, t_p) \in \mathscr{I}_N :\ l_p \neq l_{p'}\},\\
\mathscr{C}'_N &= \{t_{p'} \sqsubseteq t_p \mid \exists (l_{p'}, t_{p'}) \in \mathscr{O}_N\ \exists (l_p, t_p) \in \mathscr{I}_N \wedge l_p = l_{p'}\}.
\end{aligned}$$

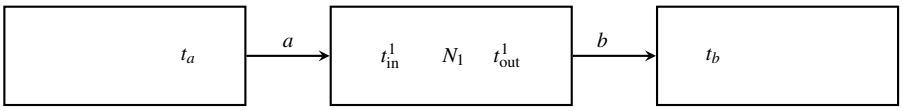
The wrap-around connection wires output ports of a service with identically named input ports and generates constraints on the port interfaces. Furthermore, the ports connected by a channel must be excluded from the sets of input/output ports $\mathscr{I}_N$ and $\mathscr{O}_N$.

The type of the top-level network contains a set of communication constraints, which serve as an input to the constraint satisfaction problem for Web services. The network topology is safe for communication if the constraints can be satisfied and unsafe otherwise.

7.3 Subtyping

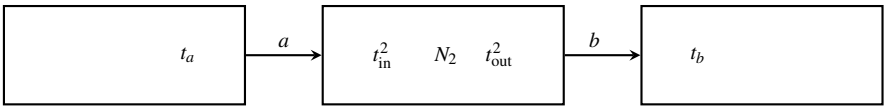
Next we introduce subtyping on types $(\mathscr{I}, \mathscr{O}, \mathscr{C})$. It defines the hierarchy of networks and hierarchy of partial solutions to the constraints $\mathscr{C}$ (see the example in Fig. 7). Subtyping can be formally defined as follows:

$$\begin{gathered}
\forall (l_p^2, t_p^2) \in \mathscr{I}_{N_2} \exists (l_p^1, t_p^1) \in \mathscr{I}_{N_1} : l_p^1 = l_p^2 \wedge t_p^1 \sqsubseteq t_p^2\\
\forall (l_p^2, t_p^2) \in \mathscr{O}_{N_2} \exists (l_p^1, t_p^1) \in \mathscr{O}_{N_1} : l_p^1 = l_p^2 \wedge t_p^2 \sqsubseteq t_p^1\\
\text{(S-SUB)}\ \frac{\mathscr{C}_{N_1} \neq \bot \qquad V(\mathscr{C}_{N_1}) \supset V(\mathscr{C}_{N_2}) \vee \mathscr{C}_{N_1} \to \mathscr{C}_{N_2} \qquad \mathscr{C}_{N_1} \models \mathscr{C}_{N_2}}{(\mathscr{I}_{N_1}, \mathscr{O}_{N_2}, \mathscr{C}_{N_1}) \leq (\mathscr{I}_{N_1}, \mathscr{O}_{N_2}, \mathscr{C}_{N_2})}
\end{gathered}$$



$$(N_1 : \{(a : t^1_{\text{in}})\}, \{(b : t^1_{\text{out}})\}, \mathscr{C})$$

(a) The port a of the network N_1 is connected to a service that provides the term t_a; the port b is connected to a service that expects the term t_b. The constraints $t_a \sqsubseteq t^1_{\text{in}}$ and $t^1_{\text{out}} \sqsubseteq t_b$ are produced



$$(N_2 : \{(a : t^2_{\text{in}})\}, \{(b : t^2_{\text{out}})\}, \mathscr{C})$$

(b) The port a of the network N_2 is connected to a service that provides the term t_a; the port b is connected to a service that expects the term t_b. The constraints $t_a \sqsubseteq t^2_{\text{in}}$ and $t^2_{\text{out}} \sqsubseteq t_b$ are produced

Fig. 7: An example illustrating subtyping: N_1 is a subtype of N_2 if $t^1_{\text{in}} \sqsubseteq t^2_{\text{in}}$ and $t^2_{\text{out}} \sqsubseteq t^1_{\text{out}}$. As a result, $t_a \sqsubseteq t^1_{\text{in}}$ implies $t_a \sqsubseteq t^2_{\text{in}}$ and $t^1_{\text{out}} \sqsubseteq t_b$ implies $t^2_{\text{out}} \sqsubseteq t_b$

where

- $V(\mathscr{C})$ denotes the set of all variables $\mathscr{C}$;
- $\mathscr{C}_{N_1} \to \mathscr{C}_{N_2}$ declares that $\mathscr{C}_{N_2}$ is a satisfiable set of constraints if $\mathscr{C}_{N_1}$ is a satisfiable set of constraints too;
- $\models$ is a logical entailment, which defines the following relation: if $\mathscr{C}_{N_1} \to \mathscr{C}_{N_2}$, then the solution to $\mathscr{C}_{N_1}$ is also a solution to $\mathscr{C}_{N_2}$.

Intuitively, a network $N_2 : (\mathscr{I}_{N_2}, \mathscr{O}_{N_2}, \mathscr{C}_{N_2})$ is a supertype of $N_1 : (\mathscr{I}_{N_1}, \mathscr{O}_{N_1}, \mathscr{C}_{N_1})$ if N_1 is less generic. Specifically, the sets $\mathscr{I}_{N_1}$ and $\mathscr{O}_{N_1}$ may contain more ports than the sets $\mathscr{I}_{N_2}$ and $\mathscr{O}_{N_2}$, and the number of solutions to $\mathscr{C}_{N_1}$ is less than the number of solutions to $\mathscr{C}_{N_2}$.

Furthermore, for input ports with the same labels (l_p, t^1_p) and (l_p, t^2_p) in $\mathscr{I}_{N_1}$ and $\mathscr{I}_{N_2}$ respectively, t^1_p must be a subtype of t^2_p. Similarly, for output ports with the same labels $(\widetilde{l}_p, \widetilde{t}^1_p)$ and $(\widetilde{l}_p, \widetilde{t}^2_p)$ in $\mathscr{O}_{N_1}$ and $\mathscr{O}_{N_2}$ respectively, $\widetilde{t}^2_p$ must be a subtype of $\widetilde{t}^1_p$.

The subtyping relation is reflexive, transitive and antisymmetric. There exists a unique type

$$(\mathscr{I}_{\text{top}}, \mathscr{O}_{\text{top}}, \mathscr{C}_{\text{top}}) = (\emptyset, \emptyset, \emptyset)$$

such that $(\mathscr{I}, \mathscr{O}, \mathscr{C}) \leq (\mathscr{I}_{\text{top}}, \mathscr{O}_{\text{top}}, \mathscr{C}_{\text{top}})$ for any $(\mathscr{I}, \mathscr{O}, \mathscr{C})$, where $\mathscr{C} = \emptyset$ denotes the set of tautological constraints. It leads us to the fact that the subtyping relation is a semilattice with $\mathscr{C}_{\text{top}}$ as the top element.

The subtyping relation defines a solution hierarchy for the CSP on $\mathscr{C}$. Indeed, any solution to $\mathscr{C}_{N_1}$ in a network $N_1 : (\mathscr{I}_{N_1}, \mathscr{O}_{N_1}, \mathscr{C}_{N_1})$ is also a solution to $\mathscr{C}_{N_2}$ in a network $N_2 : (\mathscr{I}_{N_2}, \mathscr{O}_{N_2}, \mathscr{C}_{N_2})$ providing that $(\mathscr{I}_{N_1}, \mathscr{O}_{N_1}, \mathscr{C}_{N_1}) \leq (\mathscr{I}_{N_2}, \mathscr{O}_{N_2}, \mathscr{C}_{N_2})$. However, other solutions to $\mathscr{C}_{N_2}$ may exist too.

Assume that we are looking for a solution to $\mathscr{C}_{N_2}$. The *tightest* solution is a vector of values to variables $V(\mathscr{C}_{N_2})$ such that the constraints $\mathscr{C}_{N_2}$ are satisfied and the vector is not a solution to $\mathscr{C}_{N_1}$, where N_1 is any subtype of N_2 and $\mathscr{C}_{N_1} \neq \mathscr{C}_{N_2}$.

Example 5 Consider the service-based application in Fig. 1. Assuming that the Accessories service is available, the program specifying the application in the language of combinators is the following:

```
(Components || Accessories) .. Bicycle Shop .. Customer
```

The basic types associated with each service are the following:

$$\begin{aligned}
\texttt{Components} &: (\{\}, \{a : t_{a_{\text{out}}}\}, \emptyset) \\
\texttt{Accessories} &: (\{\}, \{b : t_{b_{\text{out}}}\}, \emptyset) \\
\texttt{Bicycle Shop} &: (\{a : t_{a_{\text{in}}}, b : t_{b_{\text{in}}}\}, \{c : t_{c_{\text{out}}}\}, \emptyset) \\
\texttt{Customer} &: (\{c : t_{c_{\text{in}}}\}, \{\}, \emptyset)
\end{aligned}$$

a, b and c are names of the ports. The ports of connected services are explicitly called the same if they need to be connected by a communication channel. $t_{a_{\text{in}}}, t_{a_{\text{out}}}, t_{b_{\text{in}}}, t_{b_{\text{out}}}, t_{c_{\text{in}}}, t_{c_{\text{out}}}$ are terms that are defined in the MDL:

$$\begin{aligned}
t_{a_{\text{out}}} &= (:\mathsf{comp}\text{: } \{\mathsf{price}\text{: } \texttt{int}, \mathsf{frame}\text{: } \texttt{int}\}:) \\
t_{a_{\text{in}}} &= (:\mathsf{comp}(x)\text{: } \{\mathsf{price}\text{: } \texttt{int} \,|\, p^{\downarrow}\}:) \\
t_{b_{\text{out}}} &= (::) \\
t_{b_{\text{in}}} &= (:\mathsf{acc}(y)\text{: } \{\mathsf{price}\text{: } \texttt{int} \,|\, q^{\downarrow}\}:) \\
t_{c_{\text{out}}} &= (:\mathsf{bike}(x)\text{: } \{\mathsf{price}\text{: } \texttt{int} \,|\, p^{\downarrow}\}, \mathsf{acc}(y)\text{: } \{\mathsf{price}\text{: } \texttt{int} \,|\, q^{\downarrow}\}:) \\
t_{c_{\text{in}}} &= (:\mathsf{bike}\text{: } \{\mathsf{price}\text{: } \texttt{int}, \mathsf{frame}\text{: } \texttt{int}\}:)
\end{aligned}$$

The terms represent service interfaces, that is the data format that the service can receive and produce.

The type derivation tree for the service program is the following (`C`, `A`, `S` and `Cs` are used as shorthands for `Components`, `Accessories`, `Bicycle Shop` and `Customer` services, respectively):

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{C}}{\texttt{C} : (\{\}, \{a : t_{a_{\text{out}}}\}, \emptyset)} \quad \dfrac{\texttt{A}}{\texttt{A} : (\{\}, \{b : t_{b_{\text{out}}}\}, \emptyset)}}{\texttt{C} \,||\, \texttt{A} : (\{\}, \{a : t_{a_{\text{out}}}, b : t_{b_{\text{out}}}\}, \emptyset)} \quad \dfrac{\texttt{S}}{\texttt{S} : (\{a : t_{a_{\text{in}}}, b : t_{b_{\text{in}}}\}, \{c : t_{c_{\text{out}}}\}, \emptyset)}}{(\texttt{C} \,||\, \texttt{A}) \,..\, \texttt{S} : (\{\}, \{c : t_{c_{\text{out}}}\}, \{t_{a_{\text{out}}} \sqsubseteq t_{a_{\text{in}}}, t_{b_{\text{out}}} \sqsubseteq t_{b_{\text{in}}}\})} \quad \dfrac{\texttt{Cs}}{\texttt{Cs} : (\{c : t_{c_{\text{in}}}\}, \{\}, \emptyset)}}{((\texttt{C} \,||\, \texttt{A}) \,..\, \texttt{S}) \,..\, \texttt{Cs} : (\{\}, \{\}, \{t_{a_{\text{out}}} \sqsubseteq t_{a_{\text{in}}}, t_{b_{\text{out}}} \sqsubseteq t_{b_{\text{in}}}, t_{c_{\text{out}}} \sqsubseteq t_{c_{\text{in}}}\})}$$

As a result, the following set of constraints for the service network is derived from the program:

$$\mathscr{C} = \{t_{a_{\text{out}}} \sqsubseteq t_{a_{\text{in}}}, t_{b_{\text{out}}} \sqsubseteq t_{b_{\text{in}}}, t_{c_{\text{out}}} \sqsubseteq t_{c_{\text{in}}}\}.$$

8 Use Cases

We illustrate our approach with two scenarios: Three Buyer Use Case and Image Classification Use Case.

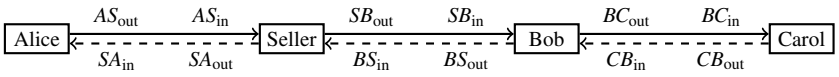


Fig. 8: Service composition in Three Buyer Use Case

8.1 Three Buyer Use Case

A simple but non-trivial example, known as the *three-buyer use case*, is often called upon to demonstrate the capabilities of session types such as communication safety, progress and protocol conformance [25]. Consider a system involving buyers called Alice, Bob and Carol that cooperate in order to buy a book from a Seller: (a) Each buyer is specified as an independent service that is connected with other services via a channel-based communication; (b) There is an interface associated with every input and output port of a service, which specifies the service's functionality and data formats that the service is compatible with.

Fig. 8 depicts the situation where Alice is connected to Seller only and can interact with Bob and Carol indirectly. *AS*, *SB*, *BC*, *CB*, *BS*, and *AS* denote interfaces associated with service input/output ports. AS_{out} declares the output interface of Alice, which declares functionality and the format of messages sent to Seller. The service has the following functionality:

- Alice can request a book's price from Seller by providing a title of an arbitrary type (which is specified by a term variable $tv^{\downarrow}$) that Seller is compatible with. On the other hand, Seller declares that only a title of type string is acceptable, which means that $tv^{\downarrow}$ has be instantiated to string.
- 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 Seller does not require the id, the interconnection is still valid (a description in standard WSDL interfaces would cause an error though) due to the subtyping supported in the MDL.
- Furthermore, Alice can offer to share a purchase between other customers. Although Alice is not connected to Bob or Carol and may even not be aware of their presence, our mechanism detects that Alice can send a message with "share" label to Bob by bypassing it implicitly through Seller.
- The mechanism sets a tail variable $ct1^{\uparrow}$ to (:share: {title: string, money: int}:) in order to enable flow inheritance in Seller's service. If Bob were unable to accept a message with "share" label, the mechanism would instantiate x with false, which automatically removes the corresponding functionality from the service.
- 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 y variable to false.

The collection elements contain guards x, y and z. as the self-configuration mechanism: Boolean variables control the dependencies between any elements of interface

collections (this can be seen as a generalized version of intersection types). The variables exclude elements from the collection if the dependencies between corresponding elements in the interfaces cannot be satisfied.

Parametric polymorphism is supported using interface variables such as $tv^{\downarrow}$, $ct1^{\uparrow}$ and $ct2^{\uparrow}$. Moreover, the presence of $ct1^{\uparrow}$ and $ct2^{\uparrow}$ in both input and output interfaces enables flow inheritance mechanism that provides delegation of the data and service functionality across available services.

Furthermore, the Boolean variable z behaves as an intersection type: Bob has "purchase sharing" functionality declared as an element $\mathsf{share}(z)$: $\{\dots\}$ in its input interface SB_{in} (used by Seller). The element is related to the element $\mathsf{share}(z)$: $\{\dots\}$ 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 Seller.

For brevity, we only provide AS, SB and BC (the rest of the interfaces are defined in the same manner) specified in the MDL:

$$AS_{\text{out}} = (:\mathsf{request}\colon \{\mathsf{title}\colon tv^{\downarrow}\}, \mathsf{payment}\colon \{\mathsf{title}\colon tv^{\downarrow}, \mathsf{money}\colon \mathsf{int}, \mathsf{id}\colon \mathsf{int}\},$$
$$\mathsf{share}(x)\colon \{\mathsf{title}\colon tv^{\downarrow}, \mathsf{money}\colon \mathsf{int}\}, \mathsf{suggest}(y)\colon \{\mathsf{title}\colon tv^{\downarrow}\}:)$$

$$AS_{\text{in}} = (:\mathsf{request}\colon \{\mathsf{title}\colon \mathsf{string}\}, \mathsf{payment}\colon \{\mathsf{title}\colon \mathsf{string}, \mathsf{money}\colon \mathsf{int}\} |\ ct1^{\uparrow}:)$$
$$SB_{\text{out}} = (:\mathsf{response}:\ \{\mathsf{title}\colon \mathsf{string}, \mathsf{money}\colon \mathsf{int}\} |\ ct1^{\uparrow}:)$$
$$SB_{\text{in}} = (:\mathsf{share}(z):\ \{\mathsf{quote}\colon \mathsf{string}, \mathsf{money}\colon \mathsf{int}\}, \mathsf{response}:\ \{\mathsf{title}\colon \mathsf{string}, \mathsf{money}\colon \mathsf{int}\} |\ ct2^{\uparrow}:)$$
$$BC_{\text{out}} = (:\mathsf{share}(z)\colon \{\mathsf{quote}\colon \mathsf{string}, \mathsf{money}\colon \mathsf{int}\} \mid ct2^{\uparrow}:)$$
$$BC_{\text{in}} = (:\mathsf{share}:\ \{\mathsf{quote}\colon \mathsf{string}, \mathsf{money}\colon \mathsf{int}\}:)$$

8.2 Image Classification Use Case

A deep neural network (DNN) is a neural network, which consists of several (two or more) hidden layers, each modifying the input data and sending the output to the next layer in the pipeline. DNNs used in production may contain more than 150 layers [24]. Deep learning frameworks provide a modular approach to designing DNNs as a composition of custom layers.

The Caffe framework [27] provides a layer catalogue, which a developer can use for DNN construction. Furthermore, the framework provides a mechanism for designing custom layers. In particular, the layer interface is defined in a Google Protocol Buffer (PB) format, which provides efficient data serialisation, a human-readable format and efficient implementation in multiple languages.

Despite its advantages, the PB interface format is non-flexible, that is the interfaces of adjacent layers should match, otherwise it will cause an error. As a result, in order to compose a DNN with high number of layers, developers have to modify the layer interfaces to make sure that they are compatible.

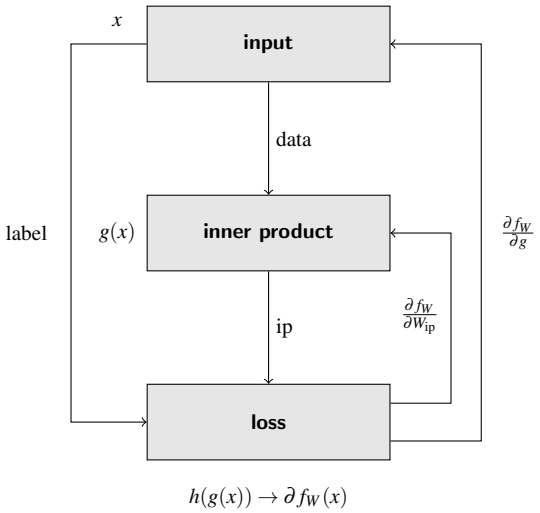


Fig. 9: Forward (from top to bottom) and backward (from bottom to top) passes through layers in neural network for image classification

In this example we demonstrate how the problem of interface compatibility can be overcome by using the MDL instead of PB as the specification language for layer interfaces.

Using the MDL as an interface definition language allows to avoid interconnections, due to flow inheritance support. Generic MDL interfaces declare *only* functionality, which they support and the data, which the layers can process. Then, using constraint satisfaction, they are contextualised to enable flow inheritance.

Fig. 9 depicts a simple logistic regression classifier[6], which is a neural network for classifying objects in images. The network consists of two passes: the forward pass computes the classification function $f_W(x)$ from a sequence of labelled input images. The image data is passed through an inner product layer $g(x)$ and then through a loss function $h(g(x))$ to compute $f_W(x)$. The backward pass takes the loss as the input and computes gradients layer-by-layer.

Fig. 10 shows composition of the neural network, where $IP_\star$, $PI_\star$, $PL_\star$ and $LP_\star$ specify layer interfaces as follows: IP_{out}, IP_{in}, PL_{out} and PL_{in} implement the forward pass, and LP_{out}, LP_{in}, PI_{out} and PI_{in} implement the back pass. In particular, IP_{out} declares that input layer exports an image in the form of a matrix and an expected classification label. The input interface IP_{in} of inner product layer is more generic, that is using polymorphism, it declares that the layer processes an image of a generic format and inherits the rest of the data in the variable $tv^{\downarrow}$:

- $tv^{\downarrow}$ is also present in the output interface PL_{out}, which implements flow inheritance for messages on the interface level.
- $tv^{\downarrow}$ contains label, which is inherited from input and is declared in PL_{in}.

[6] This example is used in Caffe tutorial: `http://caffe.berkeleyvision.org/tutorial/forward_backward.html`.

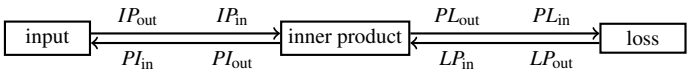


Fig. 10: Service composition for the Image Classification use case

Similarly to message inheritance, the MDL provides support for inheriting functionality across layers. We specify the interfaces in the MDL as follows:

$$IP_{out} = (:\mathsf{ip}: \{\mathsf{data}: \mathtt{matrix}, \mathsf{label}: \mathtt{bool}\}:)$$
$$IP_{in} = (:\mathsf{ip}: \{\mathsf{data}: t^{\downarrow} | tv1^{\downarrow}\}:)$$
$$PL_{out} = (:\mathsf{loss}: \{\mathsf{transformed}: t^{\downarrow} | tv^{\downarrow}\}:)$$
$$PL_{in} = (:\mathsf{loss}: \{\mathsf{transformed}: t^{\downarrow}, \mathsf{label}: \mathtt{bool}\}:)$$
$$LP_{out} = (:\mathsf{tune\_ip}: \{\mathsf{delta}: \mathtt{vector}\}, \mathsf{tune\_input}: \{\mathsf{delta}: \mathtt{vector}\}:)$$
$$LP_{in} = (:\mathsf{tune\_ip}: \{\mathsf{delta}: \mathtt{vector}\} \mid ct^{\uparrow}:)$$
$$PI_{out} = (: \mid ct^{\uparrow}:)$$
$$PI_{in} = (:\mathsf{tune\_input}: \{\mathsf{delta}: \mathtt{vector}\}:)$$

LP_{out} declares elements tune_ip and tune_input, which can be processed by inner product and input, respectively. tune_ip is matched with the corresponding element in the input interface LP_{in}. tune_input is matched by the variable $ct^{\uparrow}$ in LP_{in}, and, as a result, tune_input is automatically propagated to to the interface PI_{out} using flow inheritance. input expects tune_input in the interface PI_{in}, and, therefore, the layer interfaces are compatible.

This example demonstrates that a mechanism for configuration of flexible interfaces can be applied not only to Web services, but also to other domains with modular composition of long computational pipelines.

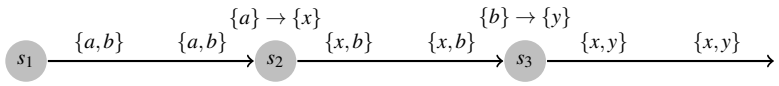
9 Interface Configuration Mechanism for Service-based Applications

In this section we present an interface configuration protocol, which supports flow inheritance in services coded in C++.

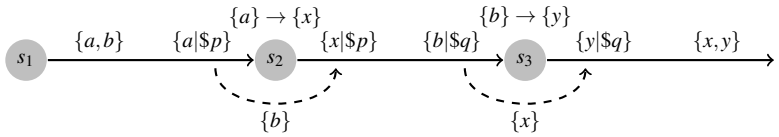
9.1 Configuration Protocol

Service loose coupling is the key principle of Web services, because a service-based application is designed in dispersed teams with diverse requirements. Each service is treated in the form of a black box, which exposes only its interface. That is interfaces have to be generic and should be adaptable to many contexts.

We define a fixed format for service interfaces, and represent an interface as a choice term at the top level. Flow inheritance can be supported then using tail variables in the choice. If choices in input and output interfaces contain the same tail



(a) A pipeline without support for flow inheritance



(b) A pipeline with support for flow inheritance

Fig. 11: A pipeline of services. $\{\ldots\}$ denotes a record as a message type

variable, then we guarantee that data that is matched by the tail variable is automatically propagated to the output as it is depicted Fig. 11.

The basic configuration mechanism is the following. Initially we assume that each service provider has a code of their C++ service. The protocol specifies a sequence of steps that need to be performed for service configuration (all steps are automatised in our toolchains).

1. First, a code preprocessing is performed for each service. In particular, the code is annotated with C++ macros. The macros are a placeholder for configuration parameters, which are generated from the CSP-MDL solution.
2. Then we derive the interfaces as MDL terms from the annotated services. This step is performed by analysing the service code.
3. Given the MDL terms, we construct a set of the seniority constraints by taking an application topology into account. This is a trivial step: for each pair of interacting services, we construct a seniority constraint from the MDL terms. This step must be performed in a centralised manner by a coordinator, because the interfaces from all services must be collected.
4. Then we solve the CSP-MDL.
5. If a solution to the CSP-MDL exists, we construct a header file that encodes the solution in the form of the C++ macro definitions. In other words, the solution contains configuration parameters for all services. The header file with its configuration is provided to all services.
6. By including the provided header files, each service provider compiles a service library that is configured specifically for the given context. Note that the providers do not need to expose the code. They only provide binaries to a runtime environment.

In Fig. 12, `.cpp` represents the service source files, where `.cpp*`* are source files augmented with preprocessor's directives, `.int` are files with the derived interfaces, `topology` is a file specifying the application communication graph, `.hpp` are header files for the augmented source files generated from the CSP-MDL solution, and `.lib` are generated libraries for the services.

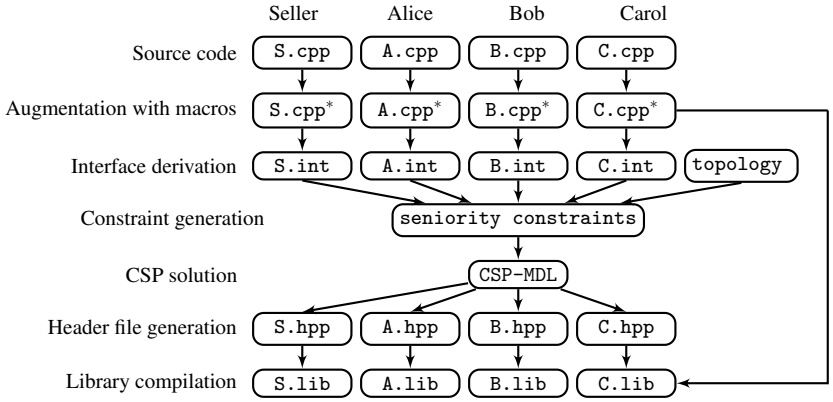


Fig. 12: An interface reconciliation workflow performed by the toolchain

9.2 Example Illustrating Configuration Protocol

Example provided in Fig. 13 represents implementation of the Seller service, where `request_1` and `payment_1` are the processing functions triggered when an input message arrives.

```
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());
    }
}
```

Fig. 13: The code of the Seller service

The processing functions are distinguished from the other functions by special return type `service`. Output messages are produced by calling special functions called *salvo*s. Salvos are declared by service developers and must have `salvo` as return type. If a processing function calls a salvo function, the salvo arguments are sent to

IN #1: (:request(x): {title: `string` $|a^{\downarrow}$}, payment(y): {title: `string`, money: `int` $|b^{\downarrow}$} | $c^{\uparrow}$:)

OUT #1: (:response(x): {title: `string`, money: `int` $|d^{\downarrow}$}, invoice(y): {id: `int` $|e^{\downarrow}$} | $c^{\uparrow}$:)

OUT #2: (:error($x \vee y$): {msg: `string` $|f^{\downarrow}$}:)

Fig. 14: One input and two output interfaces derived from the Seller service's code. Additional constraints $a^{\downarrow} \sqsubseteq d^{\downarrow}$, $a^{\downarrow} \sqsubseteq f^{\downarrow}$, $b^{\downarrow} \sqsubseteq e^{\downarrow}$, $b^{\downarrow} \sqsubseteq f^{\downarrow}$ must be generated

the output as a message. Using this mechanism, each service produces zero or more output messages as a response to a single input message.

The names of processing functions and salvos may have suffixes, such as `_1` or `_2`. The suffixes denote names of service input and output ports. The suffix in a function name specifies the name of the input port where input messages are received from; the suffix in a salvo name specifies the name of the output port where output messages are sent to.

Typically, port routing is specific to an application in which the service is used. Therefore, for service reusability we provide a mechanism, which allows to redefine a mapping between function/salvo names and ports.

The protocol provides a facility for renaming ports and message routing. This is performed from a service component called a *shell*. The shell is described in Section 9.4. In general, the shell is a service wrapper that provides facilities for port rerouting and function name renaming. Our configuration mechanism is flexible: the application designer can decide whether to accept and process error messages or not. If not, our configuration detects that the port is unwired and the salvo function `error_2` is not generated in the binary.

Fig. 14 illustrates MDL interfaces derived by our configuration toolchain. At the top level, an interface associated with a port is a choice-of-records term. Labels in the choice term of the input interface are equal to processing function names. A message is structured as a labelled data record, where the label specifies the function that processes the message. Compatibility of two communicating services is defined by the seniority relation. The interfaces can automatically be derived using the tool that we developed.

Boolean variables maintain relation between choice variants in input and output interfaces. Using Boolean variables, we can specify that salvos are present in the interface only if the functions producing the salvos can potentially receive some input.

Relation between variables in the input and output interfaces is maintained using auxiliary constraints

$$a^{\downarrow} \sqsubseteq d^{\downarrow}, a^{\downarrow} \sqsubseteq f^{\downarrow}, b^{\downarrow} \sqsubseteq e^{\downarrow} \text{ and } b^{\downarrow} \sqsubseteq f^{\downarrow}.$$

Essentially, they specify that $a^{\downarrow}$ must provide data that is required by both $d^{\downarrow}$ and $f^{\downarrow}$, $b^{\downarrow}$ must provide data that is required by both $e^{\downarrow}$ and $f^{\downarrow}$ (note that $f^{\downarrow}$ contains elements that are present in both $a^{\downarrow}$ and $b^{\downarrow}$). No other auxiliary constraints, such as $a^{\downarrow} \sqsubseteq e^{\downarrow}$ or $b^{\downarrow} \sqsubseteq d^{\downarrow}$ are needed. We know that the service cannot produce the response salvo as a response to the payment input message, and the invoice salvo as a response to the payment input message.

```
#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
```

Fig. 15: The code for the Seller service augmented with preprocessor's directives. `BV_...` and `TV_...` are macro names

Now we illustrate back-propagation of the CSP-MDL solution to services. It is the final step of the interface configuration mechanism. To simplify this task, we preliminary augment the code of each service with macro definitions as illustrated in Fig. 15, where `BV_x`, `BV_y`, `TV_a`, `TV_b`, `TV_d_decl`, `TV_e_decl`, `TV_f_decl`, `TV_d_use`, `TV_e_use` and `TV_f_use` are macros that are basically placeholders for the data that needs to be inherited.

Assume that $d^{\downarrow} = \{\mathsf{id}\colon \texttt{int}\}$, $f^{\downarrow} = \{\mathsf{rank}\colon \texttt{float}\}$, and $a^{\downarrow} = \{\mathsf{id}\colon \texttt{int}, \mathsf{rank}\colon \texttt{float}\}$ (recall that $a^{\downarrow} \sqsubseteq d^{\downarrow}$ and $a^{\downarrow} \sqsubseteq f^{\downarrow}$ must hold). Based on the solution, the header file that contains the following macro definitions will be generated:[7]

```
#define COMMA ,
#define TV_a COMMA int id
#define TV_d_decl COMMA int id
#define TV_d_use COMMA id
#define TV_f_decl COMMA float rank
#define TV_f_use COMMA rank
```

[7] In this example, "`,`" is replaced by the `COMMA` macro due to limitations of the C preprocessor

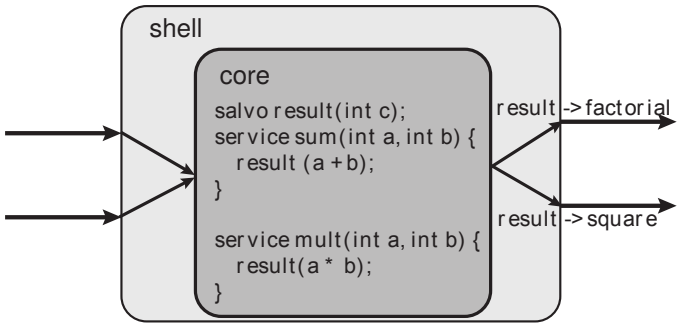


Fig. 16: The core and the shell of the service

Providing that the header file is included in the augmented source files, the above definitions replace their corresponding macro names as it is depicted in Fig. 15.

As a result, the data, which needs to be inherited will be included in processing functions and salvo functions before compilation. The configured service is compiled specifically to the given context and cannot be reused due to inherited context-specific data.

9.3 The Core

We now discuss the structure of individual services, in particular, the core and the shell. The core is the code of the service structured in the predefined format as shown in Fig. 16. The core is context-independent and can be seen as a reusable basic module. It is observed as a black box and its code is accessible only to the designer of the service. As a result, our mechanism supports proprietary services, which cannot publicly expose their source code.

The core is structured as a set of processing functions, each receiving one input message and producing a set of output messages at a time. After finishing input message processing, the internal state of the service is destroyed. Such design is useful for distributed processing, where services can be replicated to increase the throughput or migrated from one core to another without being afraid that the state is lost or unrecoverable.

The processing function is identified by the name, format of its input message and format of the output messages. The name denotes the purpose of the processing function and is equal to the function name by default. Furthermore, an MDL record, which represents a message, is mapped to C++ function arguments. It causes a problem, because elements of the record are ordered by inclusion, although C++ function arguments are identified by a position. The application layer, which triggers processing functions, reorders the record elements in a way they are specified in a processing function. On the other hand, the problem disappears in the languages with support for labelled arguments, such as Python or OCaml.

```
result -> 1, 2
1[result/factorial]
2[result/square]
```

Fig. 17: Illustration of the shell file. The salvo `result` is sent to two output ports. In the first port, the salvo name is renamed to `factorial` and in the second port the salvo name is renamed to `square`

In order to call the processing function, another service must provide a message with the same tag and compatible data format. Services developed by different enterprises rarely share tags and exact data formats. Our reconciliation mechanism solves this problem by checking compatibility of the formats.

The MDL supports depth and width subtyping, and, therefore, input messages can contain more data than the processing function requires. Assume that a consumer service requires $\{$a: `int`$\}$ as the input format of a processing function `f` and a producer service provides it with $\{$a: `int`, b: `float`$\}$. Consequently, the signature of function `f` is `service f(int a);`. The configuration mechanism must remove an element b: `float` from the input message since it is required for safe execution.[8]

Furthermore, the MDL interfaces support polymorphism, which allows to integrate the configuration mechanism with C++ templates.[9] Assume that a processing function contains an argument of type `vector<T>`, where `T` is an unitialised template argument. The type can be represented as $(\texttt{vector}\ t^{\downarrow})$, where $t^{\downarrow}$ is a free t-variable. After solving the CSP-MDL, $t^{\downarrow}$ is instantiated with a specific value, which can be propagated back to the service as specialisation of `T`.

Finally, the service can not only accept "more" data that required, but also propagate excessive data (the one that is provided, but not required by the consumer) to other services. A solution to CSP-MDL provides information about inherited record elements by analysing the application topology. Although the inherited data is specific to the context, the core of the service is reused in all contexts with different macro values (see Fig. 12, which illustrates configuration process in the context).

9.4 The Shell

Support of subtyping, polymorphism and flow inheritance provided in the MDL facilitates service reusability. On the other hand, fixed names of the processing functions breaks flexibility. Indeed, independent programming teams have different naming conventions. It is unlikely that the names of processing functions in a producer and a consumer would match. Furthermore, the services must be reusable: we want to avoid renaming names of processing functions in the code. For this purpose, we introduce the shell.

[8] Generation of such preprocessing function has not been done as part of this work though.

[9] The existing implementation of the configuration mechanism lacks such integration.

The shell is a layer that is used as an additional configuration mechanism that adapts services to the application topology. The shell is context-specific and is developed by an application designer. Transformation of processing function names can be specified in the shell as illustrated in Fig. 17.

Furthermore, the services are designed to have multiple input/output logical ports serving different purposes. For example, it is often natural to split the output into three ports: the first produces the result of computations, the second produces debug logs, and the third one produces errors. The environment may "connect" to one or another port or simply ignore it depending on the context.

There are two ways to specify port routing. First, the routing can be "hard-coded" in services as illustrated in Fig. 13. The names of processing functions and salvos contain suffixes, for example `_1` and `_2`. The suffixes specify port mapping that is used without the shell.

Developers can also specify port routing from the shell. The shell is a text file, which is associated with a service, that contains 1) processing function/salvo name transformation and 2) processing function/salvo routing to ports. Fig. 17 is the shell file that corresponds to name transformation and salvo routing from Fig. 16.

10 Conclusions

We proposed a constraint language called MDL for description of data formats in non-local Web-service communication. The language supports subtyping and flow inheritance. The language further supports Boolean configuration parameters, which help define the variety of modes a service can be used in.

Our constraint solving algorithms, has been shown correct, finds a solution to the global set of constraints defined by the MDL specifications received by the solver from a service network. The CSP-MDL is an NP-complete problem due to the presence of SAT; that is, the algorithm we presented would require exponential time in the worst case. On the other hand, the complexity of our algorithm that solves the problem without Boolean variables is polynomial as it has been shown.

Furthermore, the paper contains the description of a method for using the term assignment for configuring Web services for joint operation. and the language binding between the MDL and $C++$ has been sketched.

Our plans include developing a public domain implementation of our constraint solver and setting up a free higher-order Web service that reconciles the interfaces of individual Web services across a service network. We intend to collect use statistics to be able to draw conclusions about the average size and structure of terms and Boolean variables as well as any other quantitative or structural information of relevance. On this basis we hope to be able to answer the open questions raised in this paper in a practical way. As a result only relevant theoretical challenges will be set forward, and so any further advancements will be adequately supported with experimental base.

The existing configuration mechanism currently lacks error reporting that would be useful for application debugging. If the constraints cannot be satisfied, then the solver should provide feedback which will point an application designer towards the

interface for the particular service that is causing the problem. It would also be useful to add support for other data formats, such as generic collections, subtyped arrays [41], and functions with support for behavioural subtyping (that is subtyping on object methods), in the MDL.

The next major milestone will be the integration of the interface configuration mechanism with a full fledged service management systems such as, for example, Google Borg [46]. These systems can provide an execution environment for managing data and computations, but they lack a mechanism for checking compatibility of components. Our mechanism can be provided as an "out of the box" solution, because it does not rely on a computational model or on data management in the system.

References

1. D. Allam, H. Grall, and J.-C. Royer. From object-oriented programming to service-oriented computing: How to improve interoperability by preserving subtyping. In *WEBIST 2013-9th International Conference on Web Information Systems and Technologies*, pages 169–173. SciTePress Digital Library, 2013.
2. V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou. On the evolution of services. *IEEE Transactions on Software Engineering*, 38(3):609–628, 2012.
3. Apache. Apache Thrift. `https://thrift.apache.org/`, 2011.
4. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB standard — version 2.5, 2010.
5. M. Bodirsky. Complexity classification in infinite-domain constraint satisfaction. *CoRR*, abs/1201.0856, 2012.
6. M. Bodirsky, H. D. Macpherson, and J. Thapper. Constraint satisfaction tractability from semi-lattice operations on infinite sets. *ACM Transactions on Computational Logic*, 14(4):30:1–30:19, 2013.
7. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping constraints for incomplete objects (extended abstract). In *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*, pages 465–477, 1997.
8. A. A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. *Journal of ACM*, 53(1):66–120, 2006.
9. A. A. Bulatov. Constraint satisfaction problems over semilattice block mal'tsev algebras. *Information and Computation*, 268, 2019.
10. D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *ACM SIGPLAN Notices*, volume 21, pages 152–161. ACM, 1986.
11. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Programming Languages and Systems*, pages 2–17. Springer, 2007.
12. S. Carpineti and C. Laneve. A basic contract language for web services. In *Programming Languages and Systems*, pages 197–213. Springer, 2006.
13. G. Castagna. Covariance and controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *CoRR*, abs/1809.01427, 2018.
14. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(5):19, 2009.
15. D. A. Cohen, M. C. Cooper, P. Jeavons, and A. A. Krokhin. The complexity of soft constraint satisfaction. *Artificial Intelligence*, 170(11):983–1016, 2006.
16. N. Creignou, P. G. Kolaitis, and H. Vollmer, editors. *Boolean Constraint Satisfaction Problems: When Does Post's Lattice Help?*, pages 3–37. Springer Berlin Heidelberg, 2008.
17. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
18. R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.
19. A. Ferrara. Web services: a process algebra approach. In *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*, pages 242–251, 2004.

20. L. Fu, T. Kume, and F. Nishino. Web orchestration: Customization and sharing tool for web information. In *Human Centered Design, First International Conference, HCD 2009, Held as Part of HCI International 2009, San Diego, CA, USA, July 19-24, 2009, Proceedings*, pages 689–696, 2009.
21. Google. Protocol buffers. `https://developers.google.com/protocol-buffers`, 2008.
22. C. Grelck, S.-B. Scholz, and A. Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(02):221–237, 2008.
23. C. Grelck, S.-B. Scholz, and A. Shafarenko. Asynchronous stream processing with s-net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.
24. K. He, X. Zhang, Sh. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
25. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *SIGPLAN Notices*, 43(1):273–284, 2008.
26. P. Jeavons, D. A. Cohen, and M. Gyssens. Closure properties of constraints. *J. ACM*, 44(4):527–548, 1997.
27. Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.
28. G. A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages*, pages 194–206. ACM, 1973.
29. P. B. Ladkin and R. D. Maddux. On binary constraint problems. *Journal of ACM*, 41(3):435–469, 1994.
30. R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch. In *Datatype-Generic Programming*, pages 285–367. Springer, 2007.
31. C. Laneve and L. Padovani. The must preorder revisited. In *International Conference on Concurrency Theory*, pages 212–225. Springer, 2007.
32. Th. Y. Lee and D. W. Cheung. Formal models and algorithms for XML data interoperability. *Journal of Computing Science and Engineering*, 4(4):313–349, 2010.
33. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
34. R. N. McKenzie, G. F. McNulty, and W. F. Taylor. *Algebras, Lattices and Varieties*, volume 1. Wadsworth and Brooks, 1987.
35. G. Mentzas. *Semantic Enterprise Application Integration for Business Processes: Service-Oriented Frameworks: Service-Oriented Frameworks*. IGI Global, 2009.
36. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.
37. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.
38. B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(02):129–193, 1997.
39. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.
40. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. *IJBPIM*, 1(2):116–128, 2006.
41. S.-B. Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(06):1005–1059, 2003.
42. Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu. Web services composition: A decades overview. *Information Sciences*, 280:218–238, 2014.
43. M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.
44. A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.
45. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.
46. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.
47. Web Services Choreography Working Group and others. Web services choreography description language (WS-CDL), 2012. `https://www.w3.org/TR/ws-cdl-10/`.
48. P. Zaichenkov, O. Tveretina, and A. Shafarenko. A constraint satisfaction method for configuring non-local service interfaces. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, pages 474–488, 2016.